# Sortering i CPH STL

Jakob Sloth, Morten Lemvigh & Mads Kristensen

CPH STL rapport 2003-2 maj 2003; revidered november 2003

# Indhold

# 1 Indledning

Dette dokument er en besvarelse af et bachelorprojekt på DIKU, 2003. Projektet er udarbejdet af Jakob Sloth, Morten Lemvigh og Mads Kristensen.

## 1.1 Problemstilling

Opgaven går ud på at søge efter litteratur om, implementere og teste forskellige sorteringsalgoritmer. Dette gøres for at finde frem til de bedste sorteringsalgoritmer, der så kan implementeres i CPH STL.

Opgaven er opdelt i to dele:

- Første del beskæftiger sig med effektiv sortering i STL, og tager udgangspunkt i SGI STL's `sort` funktion. Algoritmen bag denne funktion beskrives, og det undersøges hvilke forbedringer der kan laves til denne funktion. Derefter implementeres disse forbedringer, og det testes om egentlige forbedringer har fundet sted.
- Anden del af opgaven beskæftiger sig med inplace sortering, altså sortering der ikke må bruge noget ekstra lagerplads i sin sortering. Da der ikke er implementeret nogen inplace sorteringsfunktion i STL, tager denne del istedet udgangspunkt i to artikler skrevet af Jyrki Katajainen. Algoritmerne der beskrives i disse artikler analyseres og forsøges implementeret. Herefter køres test for at finde frem til, hvilken algoritme der skal foreslås til CPH STL projektet.

## 1.2 Forudsætninger

Det forudsættes af læseren har grundliggende egenskaber indenfor algoritmik, svarende til Datalogi 2A kurset på DIKU. Dette betyder at ikke alle algoritmer forklares fuldt ud, da de forudsættes kendte, og kun der hvor det vurderes at en fuld redegørelse har berettigelse vil vi bringe den.

Det forudsættes yderligere at læseren har et grundliggende kendskab til C++ og STL, hvilket er påkrævet for at kunne læse kildekoden.

## 1.3 Testbilag

Alle kørte test er der enten tegnet grafer eller opstillet tabeller over. De egentlige uddata fra testkørslerne, som disse grafer og tabeller bygger på, har vi valgt ikke ar vedlægge rapporten. Dette er gjort da det ville fylde uhensigtsmæssigt meget. I stedet har vi under hver graf og tabel angivet, i hvilken fil uddata for denne kørsel ligger. Disse filer kan så findes på DIKU's system, hvor det ligger i `/net/skuld/home/disk24/di020276/bachelor/testdata`. Det samme gælder de kildekodefiler der kørte de enkelte tests. Disse er heller ikke vedlagt, men kan findes i et af underkatalogerne i kataloget

`/net/skuld/home/disk24/di020276/bachelor/code`, hvor al kildekoden for projektet ligger.

# 2 Sorteringsteori

Sortering er et nøgleemne indenfor datalogien, og det er således ikke tilfældigt, at Donald E. Knuth har dedikeret et helt bind til sortering og søgning i *The art of computer programming*. Ved sorteringsproblemer præsenteres man for en tabel af elementer. Hvert element har dels en nøgle, dels en ballast i form af en eller anden art information. Ønsket er at sortere elementerne efter stigende nøgle, så man hurtigt kan finde et givent element og tilgå informationen. I de to delopgaver, der følger her, vil der blive givet en række løsningsforslag til dette problem. Den første delopgave omhandler effektivitet set ud fra et tidsmæssigt synspunkt; hvor hurtigt kan man sortere en given tabel. I den anden delopgave er der sat en restriktion på sorteringen. Kravet er her at sorteringen skal være inplace; den må kun benytte $O(1)$ ekstra plads.

Før vi går til de enkelte funktioner, skal vi dog se på lidt generel sorteringsteori. For spørgsmålet må uværgeligt komme, når man sidder og optimerer funktionen: "Hvor godt kan det gøres?" Når vi vurderer effektiviteten af en sorteringsfunktion kigger vi typisk på, hvor mange sammenligninger af elementer den udfører, og hvor mange elementflytninger den foretager. Vi tager ikke højde for arbejde med indeks til tabellen eller lignende operationer, da disse typisk er negligible i forhold til de mere komplekse elementer. Men som vi skal se, så kan det dog have en betydning i praksis.

## 2.1 Flytninger

Vi vil i alle tilfælde se på, hvad vi bør forvente os i det værst tænkelige tilfælde. Det må være klart at vi skal bruge mindst $n$ flytninger i værste fald, idet alle elementer så skal flyttes. I [7] redegør Munro og Raman for hvad vi kan forvente som det bedste værstetilfælde: De benævner alle de elementer, der allerede er på deres rette plads, det vil sige elementer, der ikke skal flyttes under sorteringen, som trivielle kredse. Alle øvrige elementer optræder i ikke-trivielle kredse. En ikke-triviel kreds er en række elementer i tabellen, der skal flyttes indbyrdes for at komme på deres rette plads. Her ser vi at alle elementer i sådanne kredse skal flyttes mindst én gang, og for at kunne flytte elementerne frem og tilbage må det desuden gælde, at mindst ét element fra hver af de ikke-trivielle sekvenser kræver et ekstra flyt. Hvis vi benævner antallet af trivielle sekvenser med $x$ og antallet af de ikke-trivielle med $y$, så kan vi opskrive følgende udtryk for det mindste antal flytninger, der kræves: $n - x + y$. Bedste tilfældet opstår, når alle elementer allerede står på deres plads, dvs. $x = n$. Værstetilfældet opstår når $x = 0$ og vi maksimerer antallet af kredse. Vi har det største antal kredse,

Figur 1: Et sammenligningstræ til sortering af tre elementer.

når elementerne skal bytte plads to og to. Da vi højst kan have $\lfloor n/2 \rfloor$ ikke-trivielle kredse, så vil den øvre grænse for denne minimumsbetragtning være $\lfloor 3n/2 \rfloor$.

## 2.2 Sammenligninger

Når vi sorterer en sekvens prøver vi at bestemme den interne ordning mellem elementerne. Knuth sammenligner det i [5] med en "knald eller fald"-turnering mellem tallene. Hvor mange kampe skal der udspilles mellem elementerne, førend vi kan bestemme deres rækkefølge?

Hvis vi opstiller problemet i et binært træ, så kan det anskues mere systematisk. Alle interne knuder i træet står for en sammenligning mellem to elementer. De forskellige veje ned gennem træet fra rod til blad udgøres af de sammenligninger en given sorteringsalgoritme skal bruge for at sortere sekvensen. Til en given knude er det venstre og højre under træ de to løsningsrum, der opstår alt efter, hvilket af elementerne i sammenligningen, der er det største. I bladene optræder den permutation af inputsekvensen, der afspejler den sti, de ligger for enden af. Da en sorteringsalgoritme skal kunne konstruere alle tænkelige permutationer af sekvensen - sammenlignet med, at tal på alle positioner i listen kan hænde at være det mindste, næste mindste... osv., vil vi i bladene få alle tænkelige permutationer af startsekvensen, se evt. figur 1. Der vil derfor mindst være $n!$ blade i træet. En sorteringsalgoritme, der giver flere blade i træet, må udføre redundante sammenligninger; flere af bladene vil indeholde den samme permutation. Da vi er på jagt efter et minimum af sammenligninger vil vi forudsætte, at der ikke foretages redundante sammenligninger.

Antallet af sammenligninger vi skal udføre, svarer til længden af den sti vi skal følge for at nå til et blad, dvs. højden af træet. Lad os betegne antallet

af sammenligninger med $S(n)$. I et træ med højden $S(n)$ vil der maksimalt være $2^{S(n)}$ blade. Som det også ses af figur 1 behøver sorteringstræet ikke at være komplet, vi kan derfor generelt skrive:

$$n! \leq 2^{S(n)} \Leftrightarrow \log_2(n!) \leq S(n)$$

Da vi har at $\log(a) + \log(b) = \log(ab)$ kan vi omskrive udtrykket på følgende vis:

$$S(n) = \log_2(n!) = \sum_{i=1}^{n} \log_2(i) = O(n \log_2(n))$$

Argumentet for at sidste lighedstegn holder er, at

$$\sum_{i=1}^{n} \log_2(i) < \sum_{i=1}^{n} \log_2(n) = n \log_2(n)$$

For at få en bedre tilnærmelse kan $\log_2(n!)$ ifølge [5] omskrives med Stirlings approksimation:

$$S(n) = n \log_2(n) - \frac{n}{\log_2(n)} + \frac{\log_2(n)}{2} + O(1)$$

Hvilket er $O(n \log_2(n))$.

# Del I
# Effektiv sortering

## 3 Sortering i SGI STL

Den algoritme der er implementeret i SGI STL hedder *Introsort*, og er beskrevet af David Musser i artiklen [8]. Denne algoritme har, som vi skal se, en køretid på $O(n \log_2 n)$. Da vi skal sammenligne denne køretid med andre algoritmer, der har samme asymptotiske køretid, vil vi i mange effektivitetsberegninger kigge på, hvor mange sammenligninger og flytninger der udføres i en given algoritme. I dette afsnit vil vi forklare, hvordan Introsort virker, og vise at den i SGI STL implementerede Introsort i værste tilfælde benytter $3.5n \log_2 n$ sammenligninger.

Introsort er baseret på to andre populære sorteringsalgoritmer, nemlig *Quicksort* og *Heapsort*. Quicksort er den mest anvendte sorteringsalgoritme, på grund af den meget gode gennemsnitlige køretid, der er $O(n \log_2 n)$, med en meget lille konstant. Quicksort har dog den uheldige egenskab, at den værste køretid er $O(n^2)$, hvilket gør, at den ikke kan bruges, når det er vigtigt, at sorteringen *altid* skal ske indenfor et vist tidsrum. I de tilfælde, hvor man ikke kan tillade sig at benytte en algoritme med en så dårlig værste køretid, benytter man normalt Heapsort til sorteringen istedet. Heapsorts køretid er $O(n \log_2 n)$ både i bedste og værste tilfælde. Heapsort er dog i praksis 4-5 gange langsommere end Quicksort, og man betaler derfor en stor pris, for at undgå de få tilfælde, hvor Quicksort kører ud i en kvadratisk køretid.

Dette forsøger Introsort at forbedre på, ved som første prioritet at benytte Quicksort, som har den gode gennemsnitlige køretid. Under kørslen af Quicksort overvåger man sorteringen, så man finder ud af, om kørslen er på vej ind i en kvadratisk køretid. Hvis dette er tilfældet, skiftes der til Heapsort, så man undgår den kvadratiske køretid.

### 3.1 Overvågning af Quicksort

Det er vigtigt, at man under overvågningen af Quicksort sørger for at skifte til Heapsort, inden al for meget arbejde er udført, da vi stadigvæk er interesserede i at have en værstetilfældeskøretid på $O(n \log_2 n)$. Den andel arbejde der er udført inden skiftet til Heapsort, må højst være $cn \log_2 n$, hvis hele algoritmens køretid skal holde sig inden for den angivne grænse.

Introsort overholder naturligvis denne grænse, og i den beskrevne algoritme er konstanten $c$ sat til 2. Denne værdi opnås pga. den afgrænsningsstrategi der benyttes i algoritmen. Måden Quicksort overvåges i Introsort er ved at kigge på den dybde, de rekursive kald når til. Quicksort er som bekendt en *Divide-And-Conquer* algoritme, og den kalder sig selv

rekursivt på mindre og mindre dele af inddata, indtil yderligere opdeling ikke er mulig, og sorteringen derfor er forbi[1]. Måden Quicksort kan ramme sin værste køretid er ved at vælge et uheldigt *pivot* element, der gør at man kun får fraskilt ganske få elementer i hver rekursion. Hvis man på denne måde kun får fraskilt et konstant antal elementer hver gang, vil køretiden blive de førnævnte $\Omega(n^2)$. Dette sker da man kan fraskille et konstant antal elementer $\Omega(n)$ gange, og for hver gang dette gøres, skal der partitioneres omkring et pivot, hvilket også tager $\Omega(n)$. Når dette værstetilfælde rammes, vil rekursionsdybden af Quicksort nærme sig $O(n)$, og det er ved at kigge på denne dybde, man forudser et eventuelt værstetilfælde.

I Introsort vælges det at skifte til Heapsort, hvis rekursionsdybden overstiger $2\log_2 n$, hvor der ifølge det overstående ræsonnement er udført arbejde svarende til $2n\log_2 n$ sammenligninger.

## 3.2 Den valgte Heapsort

Der findes et par variationer af Heapsort, og nogle af deres kendetegn skal kort nævnes her. I Mussers artikel nævnes det ikke, hvilken version af Heapsort der bør anvendes i algoritmen, men i SGI STL implementationen er der naturligvis truffet et valg. Valget er her faldet på *Bottom-Up-Heapsort*, der er beskrevet i [10]. Det er dog ikke præcist den version af Bottom-Up-Heapsort, der er beskrevet i [10], der benyttes i SGI STL, men en lidt anden variant som vi vil beskrive senere.

### 3.2.1 Den oprindelige Heapsort

Kort beskrevet virker den oprindelige version af Heapsort på følgende måde:

1. Opbyg en MAX-Hob af den data der skal sorteres.
2. I en løkke der udføres $n-1$ gange gøres følgende: Ombyt det forreste element med det bagerste, indskrænk hobstørrelsen med een og genopret hoben.

Operationen, der genopretter hobbetingelsen (*reheap*), tager $O(\log_2 n)$ tid, og ved hjælp af denne opbygges hoben. Man starter med at sørge for, at hobbetingelsen er opfyldt i de nederste dele af træet, og løber derefter opad for til sidst at sørge for, at hobbetingelsen er overholdt i hele træet. I dette gennemløb starter man midt i dataen ved position $\frac{n}{2}$, da alt under dette punkt er blade i træet, og derfor trivielt opfylder hobbetingelsen. Umiddelbart tager denne kørsel altså $\frac{n}{2}\log_2 n = O(n\log_2 n)$ tid for at opbygge hoben. Dette er dog ikke tilfældet, hvis man kigger på det faktum, at reheaps

[1] Den mest brugte implementation af Quicksort kalder sig selv rekursivt på den mindre del af data'en og sørger for at iterere over den større del. På denne måde holdes stakdybden på $\Theta(\log_2 n)$.

køretid er afhængig af, i hvilket niveau af hoben man befinder sig. Hvis man tager dette med i sine beregninger, viser det sig, at hobopbygningen tager $O(n)$ tid. Et bevis for dette kan ses i [1, s. 135].

Derefter sker selve sorteringen i 2. skridt, hvor man tager det største element og placerer det på sin endelige plads bagerst i hoben. Dette skift er naturligvis i konstant tid, men efterfølgende skal man sørge for, at hobbetingelsen stadigvæk er overholdt, hvilket tager $O(\log_2 n)$ tid. Skridt 2 tager altså $O(n \log_2 n)$ tid, og hele sorteringen bruger derfor $O(n \log_2 n)$ tid.

Hvis man kigger på antallet af sammenligninger og flytninger der udføres i sorteringen, hvilket er det mest brugte mål, når man skal sammenligne to algoritmer, der har samme asymptotiske køretid, kan man finde følgende for Heapsort: Opbyggelsen af hoben er ikke interessant at kigge på, da den er lineær, og derfor vil være ubetydelig i forhold til køretiden af de $n$ kald til reheap, der forekommer i skridt 2 af sorteringen. I en udførelse af reheap vil man sammenligne det element, man kigger på, med sine to børn, og hvis elementet er mindre end mindst et af sine børn, vil der også ske en flytning. Dette betyder at man i hver reheap kald udfører højst $2 \log_2 n$ sammenligninger og $\log_2 n$ flytninger. Der udføres derfor i hele sorteringen højst $2n \log_2 n$ sammenligninger og $n \log_2 n$ flytninger.

### 3.2.2 Bottom-Up-Heapsort

I den oprindelige Heapsort lader man i skridt 2 et element flyde fra toppen af hoben og nedad, for at det kan finde sin plads i træet. Da det element, der på denne måde skal flyde ned på sin plads, er taget fra bunden af træet, er elementet meget lille, og der er derfor en god chance for, at det skal flyde hele vejen ned til bunden igen. Dette betyder, at man næsten hver eneste gang rammer værstetilfældet, og derfor skal udføre $2 \log_2 n$ sammenligninger. Antallet af sammenligninger kan bringes ned, hvis man indser, at elementerne for det meste skal helt ned til de nederste niveauer, før de finder deres plads.

Det er denne observation, der er taget højde for i Bottom-Up-Heapsort. I stedet for, som i den oprindelige version af Heapsort, i hvert niveau at checke om det nuværende element er mindre end begge børn i træet, og derefter følge det mindste barn nedad, finder man fra starten den mindste sti i træet[2]. Dette gøres med blot een sammenligning for hvert niveau, hvor man sammenligner sine to børn, og derefter går videre nedad imod det mindste barn. På denne måde findes det, Ingo Wegener kalder *special leaf*, hvilket er det blad, den mindste sti fører til. Herfra dette blad begynder man så at gå opad langs den mindste sti, og for hvert skridt sammenligner man rodelementet (der skal flyde ned i træet), med det aktuelle element i den

[2]Den *mindste sti* betyder her stien der går fra rodknuden og igennem de mindste børn i hvert niveau.

mindste sti. Når man finder et element, der er større end rodelementet, indsætter man rodelementet på denne plads, og lader alle elementer over dette i den mindste sti tage sine forældres plads. Da vi kan regne med, at elementet skal indsættes som et blad, eller i det mindste på et meget lavt niveau i træet, bringer dette os ifølge [10] ned på højst $1.5 \log_2 n$ sammenligninger i værste tilfælde, istedet for de $2 \log_2 n$ den originale Heapsort brugte. Antallet af flytninger er ikke ændret i denne version af Heapsort, og de ligger altså stadigvæk på $n \log_2 n$ i værste tilfælde.

### 3.2.3 Den implementerede variant af Bottom-Up-Heapsort

Den Bottom-Up-Heapsort, der er beskrevet i det foregående afsnit, har en enkelt ulempe, og det er, at den har en meget dårlig cacheudnyttelse. Når man leder efter det specielle blad, starter man fra den ene ende at tabellen, og søger igennem indtil man finder et blad, der ligger på en af de sidste $\frac{n}{2}$ pladser i tabellen. Derefter søger man op igennem træet, for at finde den plads rodelementet skal indsættes på. Når man har fundet denne plads, begynder man igen fra toppen, med at rykke elementerne i den mindste sti op på deres forældres plads. Man løber altså fra toppen til bunden af træet, og derefter fra toppen til bunden igen, hvilket betyder at de elementer der ligger i toppen af træet, ikke længere er i cache når man skal til at flytte dem, og der sker derfor en masse cache misses i denne operation.

For at undgå alle disse cache misses er der implementeret en anden version af Bottom-Up-Heapsort i SGI STL. Denne version ombytter elementerne samtidig med, at man finder den mindste sti i hoben. Det vil sige, at man udtager rodelementet, og i hvert niveau finder det mindste barn, som med det samme bliver skubbet op på sin forælders plads. Når man når ned til bunden af træet, skubber man nu elementerne ned, imens man leder efter den rigtige plads, hvor elementet skal placeres. Det giver flere flytninger af data, da man nu flytter lige så mange elementer som man laver sammenligninger. Dette betyder at flytningerne er steget til $1.5n \log_2 n$, men alligevel vil denne model være hurtigere, da det giver en meget bedre cache udnyttelse at sammenligne og flytte elementerne på samme tid, så man ikke skal springe frem og tilbage i tabellen.

## 3.3 Den valgte Quicksort

Den Quicksort algoritme der er implementeret i SGI STL, er en *Median-of-3* Quicksort, der som navnet antyder vælger sit pivot element som medianen af tre elementer. Når man udvælger denne median, sker det på baggrund af det første, det midterste og det sidste element i tabellen. Dette giver en god ydelse for allerede sorteret data, da man på den måde vil partitionere omkring det midterste element i tabellen. For ikke-sorteret data vil det også give en god ydelse, da udvælgelsen af 3 elementer giver en bedre sandsyn-

lighed for at vælge et godt pivot element. Det kan dog stadigvæk lade sig gøre at konstruere data, som får Median-of-3 Quicksort til at køre kvadratisk.

Der er også implementeret en meget brugt optimering til Quicksort i SGI STL. Denne optimering tager fat i det faktum, at det for meget små inddata ikke kan betale sig at kalde Quicksort rekursivt for at sortere, når en simplere algoritme som f.eks. *Insertionsort*, har meget bedre praktisk køretid ved disse instanser. Insertionsorts køretid er $O(n^2)$, men ved så små inddata som f.eks. 16 elementer, er den i praksis meget hurtigere end Quicksort. Dette er den fordi, den udfører sorteringen i eet funktionskald, istedet for de mange funktionskald Quicksort bruger, og man sparer derfor den tid, det tager at kalde en ny funktion, der skal opbygge en ny stakramme osv.

Der er to måder at benytte denne optimering på:

1. Hver gang en sekvens af en længde mindre end en bestemt grænseværdi mødes, køres Insertionsort på denne sekvens.

2. Når man møder en sekvens af en mindre længde end den fastsatte grænseværdi, dropper man blot denne sekvens, og kører i stedet Insertionsort til sidst på hele tabellen.

Insertionsort har den egenskab, at hvis den kaldes på en tabel, der består små individuelt sorterede grupper, er dens køretid $O(n)$. Dette betyder, at man i det andet tilfælde vil få en fornuftig køretid ved det sidste kald til Insertionsort, og aldrig vil kunne komme ud i en $O(n^2)$ køretid.

I det første tilfælde vil de enkelte kald af Insertionsort være af konstant tid $O(1)$, da størrelsen af inddata er højst den valgte grænse og derfor konstant. Disse kald er der $O(n)$ af, så denne model tilføjer også kun $O(n)$ til køretiden.

I SGI STL har man valgt at implementere den sidstnævnte løsning, hvor man kører Insertionsort en enkelt gang til sidst i sorteringen. Det er ikke nødvendigvis den bedste løsning, men dette vender vi tilbage til i et senere afsnit. Den grænseværdi, der er valgt i SGI STL, er fastsat til 16 elementer, hvilket heller ikke nødvendigvis er den bedste løsning, så også dette vil vi vende tilbage til i et senere afsnit.

## 3.4 Analyse af Introsort

I afsnit 3.1 fandt vi frem til, at Quicksortdelen af Introsort i et værstetilfælde vil bruge $2n \log_2 n$ tid, før der skiftes til Heapsort. Da Quicksorts *partition* operation udfører $n$ sammenligninger, hver gang den køres, betyder dette også, at der udføres $2n \log_2 n$ sammenligninger i denne kørsel.

I afsnit 3.2 blev det så beskrevet, at den valgte Heapsorts antal sammenligninger er $1.5n \log_2 n$. Samlet giver dette Introsort $3.5n \log_2 n$ sam-

menligninger i det værste tilfælde, hvilket var det resultat vi præsenterede i introduktionen til dette afsnit.

Da man i sammenligningssortering ikke kan komme under $\Omega(n \log_2 n)$ i køretid, hvilket bevises i afsnit 2.2, er vi i denne opgave ikke ude efter at ændre den asymptotiske køretid for sorteringen. Vores opgave handler derimod om at prøve på at finde teoretiske forbedringer, der kan nedbringe konstanten 3.5 i den ovenstående køretid. Efter dette er undersøgt, vil vi også eksperimentere med praktiske forbedringer til sorteringen, hvilket vil være bakket op af grundige test.

# 4 Forbedring af Introsort

Hvis man vil forbedre den teoretiske køretid for Introsort, skal man som nævnt prøve på at få sænket konstanten i antallet af sammenligninger. Dette kan gøres på flere måder:

1. Ved at forbedre Median-of-3 Quicksort.
2. Ved at forbedre Bottom-Up-Heapsort.
3. Ved at kigge på hvordan Introsort benytter Quicksort og Heapsort.

John D. Valois har i artiklen "Introsort Revisited" [9] beskrevet, hvordan man kan forbedre Introsort. Valois benytter i denne artikel den tredje mulighed, hvor han kigger på, hvornår der skiftes til Heapsort i Introsort.

Valois foreslår 2 ændringer til Mussers version af Introsort. For det første vil Valois have en anden måling på, hvornår man skal skifte til Heapsort, dette kalder han "fine-grained introspection", hvilket vi beskriver i afsnit 4.1. For det andet vil Valois tilføje randomisering til Introsort, så man, før man skifter til Heapsort, prøver at randomizere sit input lidt, for at se om man ikke kan klare sorteringen med Quicksort. Denne optimering beskriver vi i afsnit 4.2.

Udover de forbedringer Valois foreslår, vil vi også forsøge os med nogle praktiske forbedringer af Quicksorts køretid. Disse vil blive beskrevet i afsnit 4.3, hvor vi specielt kigger på optimering af de kald til Insertionsort, der er i slutningen af Quicksort.

## 4.1 Fine-grained introspection

### 4.1.1 Algoritmen

Afgørelsen af, hvornår man skal skifte til Heapsort, er lavet om i Valois' Introsort. I stedet for at skifte til Heapsort når rekursionsdybden når $2 \log_2 n$ skal man kigge på, hvor mange skridt det tager at halvere inddatastørrelsen. Når man på den måde taler om at halvere sin inddatas størrelse, menes det her, at vi altid regner ud fra den største del af tabellen efter en partitionering.

Valois opstiller en simpel lineær funktion $f(x) = \lfloor mx \rfloor + b$, der beskriver, hvor mange partitioneringsskridt man forventer der skal tages, inden tabellen har reduceret sin størrelse til $2^{-x}$ gange sin originale størrelse.

Konstanten $m$ er fastlagt til 2, hvilket vises i [9, s. 3-4], og variablen $b$ bruges til at bestemme følsomheden af algoritmen, hvilket er den procentdel af inddata, der har brug for flere end det forventede antal partitioneringer. Hvis man ønsker at finde hvor mange partitioneringer, der skal bruges for at halvere inddata, skal man indsætte et 1 på $x$'s plads, og hvis man ser bort fra $b$, får man resultatet 2. Indenfor 2 partitioneringer skal tabellen altså være halveret. For ikke at skifte til Heapsort hele tiden, skal man sætte $b$ til andet end 0, så man får en lidt mindre følsom algoritme, der tolererer et vist udsving. Valois skriver at hvis $b = 4$, svarer det til en følsomhed på ca. 2%, og det er denne værdi, han bruger i sine tests. I alt giver det altså 6 partitioneringer, inden inddatas størrelse skal være halveret. Hvis inddata ikke er halveret efter dette antal partitioneringer, vil man med det samme skifte til Heapsort der kan klare resten af sorteringen. Er inddatas størrelse derimod halveret, giver man algoritmen endnu $mx$ partitioneringer til at lave en yderligere halvering af inddataen.

### 4.1.2 Køretidsanalyse

Hvis man rammer et værstetilfælde for Quicksort, som f.eks. en Median-of-3 killer sekvens[3], vil man med denne algoritme allerede efter $\lfloor mx \rfloor + b$ skridt skifte til Heapsort, og man har derfor kun udført $(\lfloor mx \rfloor + b)\, n$ sammenligninger inden skiftet. Da både $x$, $m$ og $b$ er konstanter i algoritmen, vil der derfor kun blive udført $O(n)$ sammenligninger.

I Valois implementation er $x = 1$ og $b = 4$, og vi ved fra tidligere at $m = 2$ for sortering. Dette betyder at der i denne implementation af algoritmen i værste tilfælde udføres $6n$ sammenligninger, inden der skiftes til Heapsort.

Værstetilfældet for en median-of-3-killer er altså samlet $1.5n \log_2 n + 6n$ sammenligninger, hvilket er meget bedre end den originale Introsorts køretid for denne inddata.

Dette er dog ikke algoritmens værste køretid. Værstetilfældet rammes når algoritmen skiftevis vælger et dårligt og et godt pivotelement. Hvis man ser bort fra $b$ der indstiller følsomheden i algoritmen, er der kun 2 partitioneringer til at halvere inddatas størrelse. Hvis den første af disse partitioneringer altid kun frasorterer 2 elementer, hvilket er det mindst mulige i median-of-3 quicksort, og den anden partitionering altid halverer inddata, vil sorteringen gennemføres uden nogensinde at skifte til stopperalgoritmen heapsort. Det ses at denne kørsel vil tage $O(2n \log_2 n)$, og dette er den værste køretid for algoritmen.

[3] En Median-of-3 killer sekvens er en mængde data, der vil sende Median-of-3 Quicksort ind i en $O\left(n^2\right)$ kørsel. Hvordan man laver en sådan sekvens er beskrevet i [8].

## 4.2 Remedial randomization

### 4.2.1 Algoritmen

Den anden forbedring, Valois taler om i sin artikel, er remedial randomization. Denne forbedring forsøger at undgå, at man nogensinde skifter til Heapsort, og skal bruges i samarbejde med den allerede beskrevne fine-grained introspection.

Med remedial randomization skifter man ikke til Heapsort med det samme, når man løber ind i noget inddata, der ikke har halveret sin størrelse indenfor et vist antal partitioneringer. Istedet for at skifte til Heapsort randomiserer man noget af inddata, så man får en større sandsynlighed for at finde et godt pivot element til næste partitionering. Den måde randomiseringen foregår på, er ved at ombytte det første, midterste og sidste element med tilfældige andre elementer i inddata. Da det er disse elementer, man benytter i udvælgelsen af pivot i Median-of-3 Quicksort, vil det nye pivot blive valgt som medianen af noget tilfældigt udvalgt data, og der vil derfor være en god sandsynlighed for at finde et godt pivot.

For ikke at få en dårlig køretid i det værste tilfælde skal man begrænse antallet af gange man randomiserer, så man på et eller andet tidspunkt får skiftet til Heapsort. Dette antal skal være $O(\log_2 n)$, da man ellers vil ødelægge Introsorts køretid. I Valois artikel har han valgt at sætte denne grænse til $\lfloor \log_2 n \rfloor$ kørsler.

### 4.2.2 Køretidsanalyse

Analysen af den randomiserede Introsort læner sig op ad den foregående køretidsanalyse, der beskrev køretiden for Introsort med fine-grained introspection.

Imellem hvert randomiseringsskridt kan der udføres $(\lfloor mx \rfloor + b)\, n$ sammenligninger, og disse randomiseringer sker højst $\lfloor \log_2 n \rfloor$ gange, hvilket giver et samlet antal sammenligninger på højst $((mx + b)\, n) \log_2 n$. Da $mx+b$ er en konstant i algoritmen er dette altså stadigvæk $O(n \log_2 n)$, så køretiden for Introsort holder endnu.

Som tidligere nævnt er $m = 2$ for sortering, og Valois har i sin implementation valgt at $b = 4$ og $x = 1$. Dette betyder, at det samlede antal sammenligninger inden Heapsort bliver kaldt er $6n \log_2 n$.

Værstetilfældet for denne algoritme er altså samlet $7.5n \log_2 n$ sammenligninger, hvilket er noget dårligere end den originale Introsort. I den gennemsnitlige køretid vil den dog være meget bedre end den originale Introsort, da der er meget lille sandsynlighed for at løbe ind i den værste køretid. Valois skriver, at man i de fleste tilfælde aldrig vil kalde Heapsort, da tabellen vil være sorteret efter mindre end $\lfloor \log_2 n \rfloor$ randomiseringer. Dette betyder, at man får hele sorteringen udført med den noget hurtigere Quicksort, og hele sorteringen kører derfor hurtigere. Hvor meget man vinder ved at ran-

domisere, i forhold til kun at benytte fine-grained introspection, får vi at se under vores tests.

## 4.3 Praktiske forbedringer af Quicksort

### 4.3.1 Subdivision limit

I SGI STL er det, som nævnt i afsnit 3.3, valgt at skifte til Insertionsort når inddataens størrelse er mindre end eller lig 16 elementer. Hvorfor lige præcis 16 er valgt, ved vi ikke, og vi har derfor tænkt os at undersøge, hvordan Quicksorts ydelse ændrer sig, når man varierer størrelsen af denne konstant. Baseret på disse tests vil vi vælge den grænseværdi, der giver den bedste ydelse. Der er altså her tale om en rent praktisk forbedring, der ikke ændrer noget ved den teoretiske køretid.

### 4.3.2 En eller flere kørsler af Insertionsort

Som det er beskrevet i afsnit 3.3, kan man i Quicksort vælge, om man vil kalde Insertionsort med det samme, når man møder en sekvens med færre elementer end en fastsat grænseværdi, eller om man vil vente indtil Quicksorts løkke er færdig, og derefter køre Insertionsort på hele tabellen.

I SGI STL har man valgt den sidste løsning, og også dette vil vi lave praktiske tests med, for at se hvilken model der giver den bedste ydelse.

De to forskellige modeller har begge sine fordele som kort skal beskrives her. Hvis man, som i SGI STL, venter med at køre Insertionsort til sidst, vil man kun have eet kald til denne funktion, hvor man i den anden model vil kalde Insertionsort $O(n)$ gange. Dette overhead af funktionskald på stakken, burde give modellen med eet kald til Insertionsort den bedste køretid. Dette er dog ikke nødvendigvis hele sandheden, da man ved at kalde Insertionsort når man står med den lille sekvens, udnytter sin cache meget bedre, fordi man lige har partitioneret de elementer, der ligger i sekvensen, og de derfor alle ligger i cache.

Dette beskriver Anthony LaMarca og Richard E. Ladner i deres artikel, hvor de siger om SGI STL's måde at benytte Insertionsort: "While saving small subarrays until the end makes sense from an instruction count perspective, it is exactly the wrong thing to do from a cache performance perspective."[6, s. 7].

Hvorvidt denne optimering med flere kald til Insertionsort giver en bedre køretid, vil vise sig i vores tests, hvor vi implementerer Quicksort med begge modeller.

# 5 Test

## 5.1 Grundliggende strategi

I det følgende afsnit vil vi beskrive de test, der er blevet udført med Introsort og dens tilhørende algoritmer.

Resultaterne af de enkelte test kan ses i bilag A, hvor de er vist som grafer. Hvis man ønsker at se de tal, der ligger bag graferne, kan man kigge i de filer, hvis navne er nævnt under hver graf.

I de test, hvor algoritmerne skulle testes på tilfældigt genereret data, er hver test kørt 50 gange, og de angivne tal er et gennemsnit af disse kørsler. Som tilfældighedsgenerator har vi brugt funktionen `rand()`, der ligger i `stdlib.h`.

Alle test er kompileret med GCC 3.2.2, og der er benyttet kompileringsargumenterne `-O3 -pipe -march='arkitektur'`, hvor arkitektur er den type processor, der sidder i testmaskinen. Enkelte test er udført på flere forskellige maskiner for at give et bedre billede af, hvordan algoritmen yder under forskellige arkitekturer. Hvor det ikke er angivet hvilken maskine en test er udført på, er den kørt på en Athlon-XP 2400+ med 256 Mb DDR-RAM og Linux 2.4.20. Før der er blevet afviklet test er alle services på maskinerne stoppet, så det eneste maskinen lavede var at udføre testen.

## 5.2 Subdivision limit test

Som nævnt i afsnit 4.3.1 skal det testes, hvilken størrelse de små delsekvenser af tabellen skal have, inden man skifter til Insertionsort, altså hvilken grænseværdi der giver den bedste ydelse.

Inddata til denne test er en tabel med 1.000.000 tilfældigt genererede elementer, der skal sorteres, og der testes med grænseværdier fra 1 til 64.

I testen indgår både en version af Quicksort, der benytter mange kald til Insertionsort, og en model der kun benytter det ene afsluttende kald. Disse funktioner hedder helholdsvis `qsort_m3_many` og `qsort_m3_one`.

Da denne test er meget afhængig af cache, har vi valgt at køre den på fire forskellige arkitekturer. Disse testkørsler kan ses som grafer i bilag A.1.

Hvis man ser på de fire grafer i bilag A.1, er der en tydelig tendens at spore i de fleste af dem. Den eneste test, der ikke er nem at læse, er den der blev udført på en Mobile Pentium 4, der har nogle meget uventede udsving i forhold til de andre arkitekturer.

Ud fra graferne kan man se, at det for dem alle sammen gælder, at den bedste køretid ligger ved grænseværdier imellem 20 og 25. Dette er endda også tilfældet for Pentium 4 testen, hvis man ser bort fra det underlige udfald ved grænseværdierne 3 og 4. Ud fra disse test har vi valgt grænseværdien 22, der ligger som en median imellem de forskellige arkitekturers bedste værdi.

For at se om vores valgte værdi har givet en praktisk forbedring af køretiden, har vi kørt endnu en test, hvor vi har kaldt funktionerne `qsort_-`

`m3_many` og `qsort_m3_one` i to forskellige modeller, een hvor grænseværdien er 16 som i den nuværende SGI STL implementation, og een hvor grænseværdien er sat til de 22 vores test viser skulle være bedre. Denne test kan ses i bilag A.2.

Af denne graf kan man se, at 22 giver en bedre ydelse på Athlon-XP arkitekturen. Den funktion der benytter flere kald til Insertionsort og grænseværdi 22 er den bedste for alle inddatastørrelser i grafen, og vi vælger derfor at benytte 22 som grænseværdi i vores endelige implementation af Introsort.

## 5.3 Een eller flere kørsler af Insertionsort

I den foregående test, hvor vi fandt frem til hvor store delsekvenser skal være, før man skifter til Insertionsort, kørte vi alle disse test med både `qsort_m3_many` og `qsort_m3_one`. Ud fra disse test kan man altså se, hvilken model der fungerer bedst når inddata er 1.000.000 elementer.

Ved både Athlon-XP, Athlon og Mobile P4 arkitekturerne var det en klar fordel at benytte de mange kald til Insertionsort, hvor man får den bedste cache ydelse. Ved Mobile P3 arkitekturen var det dog det modsatte der gjorde sig gældende. Her var det en lille fordel at benytte det ene afsluttende kald til Insertionsort.

Fordelen ved at benytte det afsluttende kald til Insertionsort i en P3 arkitektur er dog så meget mindre end fordelen ved det modsatte på alle de andre arkitekturer, så vi vælger derfor at benytte optimeringen, og beholder de mange kald til Insertionsort i vores endelige sorteringsfunktion.

For at vise at fordelen ikke kun er tilstede ved 1.000.000 elementer, har vi kørt endnu en test, der viser `qsort_m3_many` og `qsort_m3_one`'s køretider over flere inddata størrelser. Grafen over disse testresultater kan ses i bilag A.3.

Som man kan se på denne graf, der er kørt på en Athlon-XP, er modellen med mange kald til Insertionsort altid bedre end den anden model, og jo større inddata bliver jo bedre bliver den. Forskellen imellem de to modeller er dog ikke særligt stor, f.eks. er forskellen imellem de to funktioner ved 5.000.000 elementer kun 0.02 sekund.

## 5.4 Test af Valois' Introsort

De næste optimeringer der skal testes, er de to modeller Valois beskriver i artiklen [9]. Her testes både en model, der benytter fine-grained introspection, og en model der benytter både fine-grained introspection og remedial randomization.

Disse to modeller benytter begge helt nye måder at udføre overvågningen af Quicksort, og vi forventer os derfor noget mere af disse end de to tidligere forbedringer. Fine-grained introspection uden remedial randomization giver

også en langt bedre værstetilfældeskøretid end den originale Introsort, så denne skulle gerne give en noget bedre køretid. Når man slår remedial randomization til, får man pludselig en dårligere værstetilfældeskøretid, men da algoritmen er randomiseret, er der en meget lille chance for at det værste tilfælde rammes. Lige præcis på grund af randomiseringen, kan man forvente at ydelsen for Median-of-3 killer sekvenser er betragteligt bedre for denne algoritme, men dette vil vi få at se i de følgende test.

I de følgende test vises de tre funktioner `introsort_many`, `introsort_rev_many` og `introsort_rev_many_norandom`, der henholdsvis implementerer Mussers Introsort som den er i SGI STL i dag, Valois' Introsort med begge optimeringer og Valois' Introsort uden remedial randomization.

### 5.4.1 Tilfældigt genererede grafer

I bilag A.4 er der tre grafer, der viser de testkørsler der er lavet med tilfældigt genereret data. Den første graf viser en tidsmåling, og de to andre måler henholdsvis sammenligninger og elementflytninger.

I den første graf ser man straks to ting: 1) Valois varianter af Introsort er en del hurtigere end Mussers, og 2) der er ingen forskel i køretid imellem den randomiserede og den ikke randomiserede variant af Valois Introsort.

Ud fra disse observationer kan man slutte flere ting. For det første ser det ud til, at fine-grained introspection giver en meget bedre indikator på, hvorvidt Quicksort er på vej ind i et værstetilfælde, end Mussers rekursionsdybdemåling.

Grunden til at de to forbedrede funktioner yder bedre end Mussers Introsort er, at de ikke er så hurtige til at melde at et værstetilfælde er ramt, og derfor ikke får kaldt Heapsort så mange gange som Mussers Introsort. Dette ses i tabel 1, hvor vi har talt hvor mange gange, de enkelte algoritmer skifter til Heapsort for forskellige inddata størrelser.

| n | Introsort | Introsort Rev. (RR+FI) | Introsort Rev. (FI) |
|---|---|---|---|
| 100 | 0.02 | 0 | 0 |
| 1000 | 1.92 | 0 | 0 |
| 10000 | 30.04 | 0 | 0.24 |
| 100000 | 554.92 | 0 | 2.12 |
| 1000000 | 6444.26 | 0 | 12.84 |
| 10000000 | 52861.8 | 0 | 17.8 |

Tabel 1: Kald af stopperalgoritmen i Introsort varianterne ved tilfældigt genereret inddata. Introsort er Mussers originale algoritme, Rev. er de to varianter Valois beskriver. Forkortelsen RR betyder *Remedial Randomization* og FI står for *Fine-grained Introspection*. Data for denne tabel kan findes i filen `stopper_test`.

I denne tabel kan man se, at der ikke skiftes nær så mange gange

til Heapsort, i de to modeller Valois har beskrevet. I Mussers Introsort skiftes der f.eks. ved 1.000.000 elementer gennemsnitligt til Heapsort 6444.26 gange, hvor Valois Introsort med fine-grained introspection for samme inddatastørrelse, kun gennemsnitligt skifter til Heapsort 12.84 gange. Yderligere kan man se, at den variant, der både udnytter fine-grained introspection og remedial randomization, aldrig skifter til Heapsort for de inddatastørrelser der er testet med.

Umiddelbart kunne dette lyde som om, at modellen med randomisering er den bedste af de tre, men som man kan se på grafen, er der stort set ingen forskel på køretiden for `introsort_rev_many` og `introsort_rev_many_norandom`, de ligger nærmest på samme linie igennem hele grafen. Dette skyldes, at det relativt lille antal gange Heapsort kaldes i `introsort_rev_many_norandom`, opvejes af de gange `introsort_rev_many` skal randomisere sin data. Det skyldes højst sandsynligt også, at de få kald der er til Heapsort i `introsort_rev_many_norandom`, er på meget små delsekvenser af data.

De to andre grafer, der viser sammenligninger og flytninger, viser den samme tendens som den første. Mussers Introsort er den dårligste i begge grafer, og Valois' to varianter opfører sig stort set ens.

For at opsummere kan det altså siges, at Valois' varianter af Introsort er en del bedre end Mussers originale version, så vi vil helt sikkert anbefale at benytte en af disse varianter i CPH STL. Der er næsten ingen forskel på ydelsen imellem Valois' to varianter, så på baggrund af denne test vil vi umiddelbart tilråde, at man benytter den uden randomisering, da den har en meget bedre værstetilfældeskøretid.

### 5.4.2 Median-of-3 killers

De forskellige versioner af Introsort skal selvfølgelig også testes med Median-of-3 killer sekvenser, for at se hvordan de opfører sig, når den Quicksort de overvåger løber ind i et værstetilfælde. Resultatet af disse test kan ses i bilag A.5. Igen viser den første graf tidsmåling, og de to andre måler henholdsvis sammenligninger og elementflytninger.

I den første graf har der sneget sig en ekstra funktion ind. Det er funktionen `qsort_m3_many`, der er en standard Median-of-3 Quicksort. Den er medtaget for at vise at Median-of-3 killer sekvenserne rent faktisk giver kvadratisk køretid for denne funktion.

Som forventet kan man se på graferne, at Valois' to varianter af Introsort er bedre end Mussers ved disse sekvenser. Der er dog to forskellige grunde til at de er bedre: for varianten uden randomisering, er grunden til den bedre ydelse, at der her skiftes til Heapsort tidligere end i Mussers Introsort, da man hurtigere finder ud af at et værstetilfælde er ramt. For varianten med randomisering skyldes den gode køretid, at man aldrig skifter til Heapsort overhovedet. Ved at randomisere udvælgelsen af pivot elementet, har man altså ændret Median-of-3 killer sekvensen, til en sekvens der ikke længere er

så slem at sortere. I tabel 2 kan man se, hvor meget der skiftes til Heapsort i disse kørsler.

| n | Introsort | Introsort Rev. (RR+FI) | Introsort Rev. (FI) |
|---|---|---|---|
| 100 | 1 | 0 | 1 |
| 1000 | 1 | 0 | 1 |
| 10000 | 1 | 0 | 1 |
| 100000 | 1 | 0 | 1 |
| 1000000 | 1 | 0 | 1 |
| 10000000 | 1 | 0 | 1 |

Tabel 2: Kald af stopperalgoritmen i Introsort varianterne ved Median-of-3 killer sekvenser. Introsort er Mussers originale algoritme, Rev. er de to varianter Valois beskriver. Forkortelsen RR betyder *Remedial Randomization* og FI står for *Fine-grained Introspection*. Data for denne tabel kan findes i filen `stopper_m3k_test`.

Her kan man se, at der aldrig skiftes til stopperalgoritmen i den randomiserede variant af Introsort, hvilket som nævnt må betyde at randomiseringen får omstruktureret sekvensen, så den ikke længere er en Median-of-3 killer.

Hvis man kigger på grafen der viser antallet af sammenligninger for de forskellige algoritmer, kan man se at Mussers algoritme bruger det højeste antal sammenligninger, mens Valois' variant uden randomisering bruger en del mindre. Dette skyldes at Mussers Introsort vil have udført $2 \log_2 n$ partitioneringer med Quicksort, der kræver $O(n)$ sammenligninger, inden der skiftes til Heapsort. Den anden algoritme udfører kun 6 partitioneringer på denne måde. Den randomiserede variant af Introsort varierer naturligvis meget i antallet af sammenligninger, da det er afhængigt af hvor godt den kommer til at partitionere, men gennemsnitligt ser den ud til at have lidt færre sammenligninger end Mussers algoritme.

På grafen der viser antal flytninger kan man se, at den randomiserede variant af Introsort klart benytter færrest flytninger, hvilket skyldes at den aldrig kalder Heapsort. Efter den kommer Mussers Introsort, der benytter lidt færre flytninger end Valois' sidste variant der kun benytter fine-grained introspection.

Vinderen af denne test blev den randomiserede variant af Introsort, der har en køretid der kun er ca. $\frac{1}{3}$ af Mussers Introsort, og ca. $\frac{1}{2}$ af den ikke-randomiserede variant. Den ikke randomiserede er dog også meget bedre end Mussers Introsort, og den har en køretid der er ca. $\frac{2}{3}$ af dennes.

## 5.5 Opsummering

Konklusionen på vores test er, at der kunne laves en del små og store forbedringer af Mussers Introsort, som den er implementeret i SGI STL. Derfor vil vi nu opsummere på, hvilke forbedringer der blev lavet, og ud fra dette finde frem til den implementation vi vil foreslå til CPH STL projektet.

Først og fremmest implementerede og testede vi Valois' to varianter af Introsort, der viste sig i praksis at være en hel del bedre end Mussers Introsort både i det gennemsnitlige og i det værste tilfælde. Ved tilfældigt genererede inddata var køretiden for Valois' to varianter stort set ens, mens der ved Median-of-3 killer sekvenser var stor forskel, da den randomiserede helt undgik at kalde Heapsort, og derfor var meget hurtigere.

Den variant der benyttede randomisering har dog en værstetilfældeskøretid, der er noget dårligere end Mussers Introsort, og køretiden kommer i det værste tilfælde helt op på $7.5n \log_2 n$, istedet for de $3.5n \log_2 n$ den originale har. Da det er meget usandsynligt, at man skulle ramme dette værste tilfælde, vil vi foreslå at man alligevel benytter varianten med randomisering i CPH STL. Vi så i afsnit 5.4.1 at de to algoritmer Valois beskriver, yder det samme ved tilfældigt genereret data, så i dette tilfælde er det lige meget, hvilken algoritme man vælger, og hvis man skulle vælge en algoritme udelukkende på dette grundlag, ville det være den ikke randomiserede variant valget faldt på, da den har det gode værste tilfælde $1.5n \log_2 n$.

Ved konstrueret data som f.eks. Median-of-3 killers og organpipe[4] sekvenser, er den randomiserede variant dog meget bedre end den ikke randomiserede, og derfor mener vi at man bør vælge denne, da den alt i alt giver en bedre praktisk køretid, og da det er meget usandsynligt at man skulle ramme værstetilfældet.

De to andre forbedringer vi testede, der var forbedringer af Quicksort, var af rent praktisk natur. Den ene var en test af hvor store delsekvenser må være, inden man skifter til Insertionsort, og den anden testede om man ikke ville få en bedre køretid, ved at kalde Insertionsort med det samme, når man mødte en lille sekvens, istedet for at benytte eet sidste kald til Insertionsort, der sorterede alle de små delsekvenser.

Begge disse optimeringer gav små praktiske forbedringer, som man kan se i de testkørsler der beskrives i afsnit 5.2 og 5.3, og vi vil derfor også foreslå at disse implementeres i CPH STL.

I vores optimeringer har vi prøvet at forbedre overvågningen af Quicksort vha. Valois' artikel, og at forbedre Quicksort selv med vores små praktiske forbedringer. Den eneste del af Introsort, vi ikke har forsøgt på at optimere, er stopperalgoritmen Heapsort. Man kunne her have prøvet at implementere en bedre Heapsort, og lede efter små praktiske forbedringer til denne algoritme. Dette er en mulig udvidelse, der kunne testes for at finde ud af om

[4] Organpipe sekvenser er tabeller med et lige antal elementer, hvor data'en er heltal der ligger på denne måde $\left[1, 2, 3, ..., \frac{n}{2}, \frac{n}{2}, ..., 3, 2, 1\right]$.

det giver en reel forbedring.

# Del II
# Inplace sortering

På trods af moderne maskiners stadig større kapacitet findes der situationer, hvor det kan være ønskværdigt at minimere den ekstra plads, der skal bruges under sorteringen. Ved ekstern sortering vil man typisk hente mest muligt data ind i arbejdshummelsen for at minimere mængden af bekostelig I/O. Hvis elementerne der sorteres er store, kan det yderligere være en bekostelig affære at flytte dem rundt i den interne hukommelse, heraf opstår ønsket om at minimere antallet af flytninger.

Vi så i afsnit 2.1, at det minimale antal flytninger vi behøver for at sortere en tabel er $\lfloor 3n/2 \rfloor$. Hvorvidt dette kan nås i praksis med inplace sorteringsalgoritme er uvidst, men mange bud er givet.

Hobsortering er formentlig den hurtigste familie af kendte inplace sorteringsalgoritmer. Den eneste ulempe er, at den udfører relativt flere flytninger end en tilsvarende flettesortering. Vi skal derfor i denne del af opgaven kigge på inplace flettesorteringer. I den følgende afprøvning vil vi dog sammenligne de behandlede flettesorteringsalgoritmer med en hobsortering.

Traditionelle inplace flettesorteringsalgoritmer fungerer på grundlag af en inplace flettefunktion. Med en sådan er det en simpel sag at konstruere en inplace flettesorterings-algoritme. Huang og Langstons bud på en inplace flettefunktion fremhæves i [3] som værende den mest effektive algoritme til dato med hensyn til flytninger; den har en øvre grænse på $5(m+n) + O(\sqrt{m+n}\log_2(m+n))$ flytninger, hvor $m$ og $n$ er længden af de sekvenser der flettes. Sammenlignet med konstanten på en ikke-inplace flettefunktion, hvor vi kan nøjes med $n+m$ flytninger, er der ladet meget tilbage at ønske. I [3] skitserer Katajainen et al., hvordan man kan konstruere en effektiv inplace flettesorteringsalgoritme uden inplace fletninger. I [4] foreslås det, hvordan man kan forbedre denne algortime med hensyn til antallet af flytninger.

Vi skal her gengive de to algoritmer og give en beskrivelse af vores implementation. Samtidig vil vi præsentere en analyse af det samlede antal flytninger og sammenligninger i begge algoritmer. Endelig vil vi i afsnit 8 præsentere en række prøvekørsler af de udviklede programmer.

## 6 Praktisk inplace flettesortering

I det følgende vil der ses på den ene af de to inplace sorteringsalgoritmer som denne opgave omhandler. Algoritmen er beskrevet i [3], og det er den avancerede algoritme fra denne artikel, der her vil blive gået i dybden med. Algoritmen bygger på fletning som gennemgående metode i sorteringen. Strukturen vil være følgende: Først vil alle algoritmens bestandele blive gen-

nemgået og beskrevet, hvorefter den teoretiske del, der omhandler køretid, antal flytninger og antal sammenligninger, vil blive fremlagt.

## 6.1 Algoritmen

Den grundlæggende ide ved denne inplace sorteringsalgoritme er at undgå at skulle benytte inplace fletning som bestandel af algoritmen. I stedet for at benytte inplace fletning, bygger metoden på at kunne udføre en rutine, der minder om en standard 2-vejs-fletning. Standard 2-vejs-fletning benytter nemlig kun lineært antal flyt og sammenligninger, men derimod er den ikke inplace, da den behøver ekstra plads, der svarer til længden af de to sekvenser tilsammen. For at undgå at bruge den ekstra plads kan man i stedet finde den nødvendige plads i tabellen der skal sorteres. Dette kan gøres ved at opdele listen op i to dele. Den der skal sorteres, der fremover vil blive kaldt sorteringszonen, og pladsen hvor de sorterede elementer bliver flyttet til under fletningen, denne plads kaldes fremover arbejdsområdet. På den måde kan elementerne, der skal flettes, blive flyttet til arbejdsområdet i sorteret orden og inplace fletning undgås. Metoden kan yderligere udvides så to sekvenser $m$ og $n$ kan flettes sammen med brug af $\min(n, m)$ arbejdsplads. Det er dog nødvendigt, i forhold til en standard fletning, at elementerne i arbejdsområdet flyttes, så de ikke bliver overskrevet. Overskrivelsen kan udgås, ved at elementerne i arbejdsområdet undervejs i fletningen bytter plads med elementerne i sorteringszonen.

### 6.1.1 Teknik til flyt af elementer

Når elementer skal flyttes mellem sorteringszonen og arbejdsområdet, vil man umiddelbart gøre dette i tre flyt. Dette er dog kun nødvendigt hvis flytningen skal være stabil. Man kan i stedet nøjes med to flyt og stadig holde den ene del sorteret. Teknikken består i, at der ved flyt af to tabeller bliver lavet et "hul" i den ene tabel, før der flyttes. I dette tilfælde bliver hullet lavet i arbejdsområdet, da denne del ikke behøver at blive flyttet stabilt. Hullet laves ved at gemme første element i en temporær variabel. Når hullet er lavet, kan det opretholdes igennem fletningen ved først at flytte et element fra sorteringszonen over i hullet og derefter lave et nyt hul, ved at flytte næste element i arbejdsområdet over i sorteringszonen. I sidste flyt bliver elementet i den temporære variabel sat ind i slutningen af sorteringszonen. Her ligger grunden til at teknikken ikke er stabil.

## 6.2 Partitionsprincippet

På baggrund af teknikken med brug af et arbejdsområde i fletningen, bygges sorteringen op omkring et partitionsprincip. Metoden er illustreret på figur 2, og det, der sker i sorteringen, kan groft deles op i to forskellige aktiviteter; en del hvor der sorteres, og en del hvor der flettes. Der ses på en tabel $A$

af størrelse $n$, der skal sorteres. Først deles tabellen $A$ op i to dele $Q$ og $P$, hvor $P = [1, \lfloor n/2 \rfloor]$ og $Q = [\lceil n/2 \rceil, n]$ og Q sorteres. Hvordan sorteringen foretages forklares i afsnit 6.3. Derefter deles $P$ op i to lige store dele $P_1$ og $P_2$. $P_1$ bliver derefter sorteret med samme metode som $Q$ blev sorteret med. Når de to deltabeller er sorterede, bliver de flettet sammen ved at benytte $P_2$ som arbejdsområde. Dette kan gøres, idet $P_2$ har samme størrelse som $P_1$ og derfor er lig $\min(P_1, Q)$. Hvordan fletningen foregår beskrives i afsnit 6.3.3. Efter fletningen er den sorterede deltabel $Q$ blevet forøget med $P_1$ elementer, og denne mængde kaldes nu $Q'$ hvor $Q' = Q + P_1$. De elementer der før lå i $P_2$, og som blev brugt som arbejdsområde, skiftede under fletningen plads med $P_1$. $P_2$ er stadig usorteret, og sorteringen fortsætter nu på denne del, nu kaldet $P_2'$. Sorteringen af $P_2'$ bliver udført på samme måde som for $P$, så $P_2'$ og $Q'$ tildeles samme roller som $P$ og $Q$ og den beskrevne metode gentages:

- $P$ deles i to lige store dele.
- Første del af $P$ sorteres.
- Første del af $P$ flettes med $Q$ over i arbejdsområdet, der består af den anden usorterede del af $P$.

Figur 2: Det grundlæggende princip i sorteringen.

Sorteringen af elementerne ved brug af partitioneringsprincippet fortsætter således indtil et givent punkt. Hvordan og hvornår der stoppes bliver beskrevet i afsnit 6.3.4, der viser hvordan sorteringen afrundes.

## 6.3 Sorteringsdelen

Sorteringen der sker af både $Q$ og $P_1$ før disse to sekvenser flettes sammen bliver udført ved hjælp af en 4-vejs-fletning. Denne fletning er en udvidelse af 2-vejs-fletning, da der flettes mellem 4 sekvenser ad gangen i stedet for 2 sekvenser. Proceduren i 4-vejs-fletning er grundlæggende den samme som

i 2-vejs-fletning. Det vil derfor for overskuelighedens skyld først blive gennemgået hvordan 2-vejs-fletningen bliver udført.

2-vejs-fletningen skal udover sorteringszonen have et arbejdsområde til rådighed, der har samme størrelse som sorteringszonen. Dette er nødvendigt fordi fletningen skal være inplace. På grund af strukturen af partitionsprincippet vil dette område altid være til rådighed. I første iteration kan $Q$ benytte $P$ som arbejdsområde, og $P_1$ kan benytte $P_2$ og så fremdeles.

Sorteringszonen vil fremover blive kaldt $A$, og $A$ har størrelse $n$. Fra starten opfattes sorteringszonen som en mængde af $n$ delsekvenser, dvs. hver delsekvens er een lang. Fordi fletningen er 2-vejs flettes to delsekvenser ad gangen, hvilket fremover vil blive kaldet fletning af en gruppe. Gruppen består i 2-vejs-fletning altid af to sekvenser.

Figur 3: Viser 2-vejs-fletning med et tilhørende arbejdsområde til rådighed. $s$ på figuren er sekvenserne gennem forløbet.

Efter opdelingen af $A$ i grupper er sket, flettes alle sekvenser i sorteringszonen. I første iteration vil der i sorteringszonen være $n/2$ grupper der flettes derefter $n/2/2$ grupper osv., indtil der kun er een delsekvens på størrelse med $n$. Grunden til at antallet af grupper der skal flettes bliver halveret i hver iteration er, at de sorterede sekvensers størrelse bliver fordoblet ved hver fletning. Fletningen er illustreret på figur 3.

Som det kan ses på figuren skifter sorteringszonen og arbejdsområdet plads efter hver iteration. Sorteringszonen skal slutteligt ligge på dens udgangsposition, det er derfor nødvendigt at flytte elementerne til den rette plads, når sorteringen er slut, hvis den ikke allerede ligger der.

Selve fletningen sker, ved at de mindste elementer fra hver sekvens i en gruppe sammenlignes, og det mindste heraf bliver flyttet til arbejdsområdet. Denne procedure fortsætter, indtil begge sekvenser er tomme. På den måde bliver de to sekvenser i en gruppe sorteret, og gruppen vil ved næste iteration være en sekvens. Sorteringen afsluttes, når der kun er en sekvens, da denne

vil være fuldt sorteret.

Som før nævnt benytter denne algoritme sig af 4-vejs-fletning. Forskellen mellem 2-vejs og 4-vejsfletning er, at hver gruppe ikke består af 2 sekvenser som ved 2-vejs-fletning, men i stedet 4 sekvenser. Derudover er metoden den samme. Motivationen for at udvide fletningen fra en 2-vejs-fletning til 4-vejs-fletning er, at antallet af flyt i fletningen bliver reduceret med en faktor 2. Dette skyldes, at grupperne har den dobbelte størrelse. Hvor grupperne i 2-vejs-fletning vokser med en faktor to per iteration, vokser de i 4-vejs-fletning med en faktor fire per iteration, og vi skal derfor udføre halvt så mange iterationer.

Mere generelt kan man sige, at antallet af flytninger i en $d$-vejs fletning vil være mindre end ved en 2-vejs flytning med $d > 2$ selvfølgelig. Dette skyldes, at antallet af niveauer i fletningen formindskes fra $\log_2(n)+O(1)$ til $\log_d(n)+O(1)$. Som ved 2-vejsfletning flyttes hvert element i $n$ een gang per niveau, så derved er antallet af flytninger reduceret til $(2/\lceil \log_2(d) \rceil) n \log_2(n) + O(n)$ i modsætning til 2-vejs fletnings $n \log_2(n) + O(n)$ flyt.

Vi har i denne opgave implementeret to varianter af 4-vejs-fletsortering, fordi der i [3] viste sig at være forskel på ydelsen på de to. Forskellen på de to variater ligger i den måde, det mindste element fra hver gruppe findes på.

### 6.3.1 4-vejs-fletning med selektionstræ

I 4-vejs-fletningen risikeres det, at antallet af sammenligninger forøges, da der umiddelbart må foretages flere sammenligninger, hver gang det mindste element i gruppen skal findes. Ved blot at sammenligne de fire elementer fra hver gruppe, vil dette kræve 3 sammenligninger, i modsætning til en enkelt sammenligning i 2-vejs-fletning. Det er netop denne forøgelse af sammenligninger den første variant af algoritmen, der benytter et selektionstræ, forsøger at undgå.

Der benyttes et selektionstræ til at holde styr på det mindste element i hver af de fire sekvenser. Det er også fra træet, at det mindste element i gruppen bliver fundet. For at generalisere vil opbygningen af træet blive forklaret ud fra en $d$-vejs-fletning. Ideen blev introduceret i [5], og fungerer på følgende måde:

Selve træet er opbygget af pointere, så det udgås at skulle flytte selve elementerne. Bladene i træet er pointere til det mindste element i hver sekvens. Træet har derfor $d$ blade, og samlet bliver træets størrelse derfor på $2 * d - 1$ pointere. Ideen er at roden peger på det blad, der indeholder pegeren til det mindste element i gruppen. På den måde kan det mindste element tages ud af gruppen ved at se på roden af træet. Selve arbejdet består derfor i at opdatere træet, hver gang et element bliver taget ud af gruppen. Roden skal igen pege på det mindste element i gruppen. Denne opdatering kan gøres bottom-up og kræver derfor træets dybde, $\lceil \log_2(d) \rceil$ sammenligninger, dvs. i 4-vejs-fletning 2 sammenligninger. Når en sektion ikke har flere elementer,

der skal flettes, bliver bladet, der peger på sekvensen, sat til -1. Fletningen er derfor færdig, når rodknuden peger på et blad med værdien -1.

Det kan vises, at antallet af sammenligninger ikke stiger, når der benyttes et selektionstræ til $d$-vejs-fletningen. I 2-vejsfletning er antallet af sammenligninger $n\log_2(n) + O(n)$. I $d$-vejsfletning vil der blive foretaget $O(n/d)$ $d$-vejsfletninger, og gennem algoritmen vil dette betyde O(n) ekstra sammenligninger for at oprette selektionstræerne. For alle elementer undtagen det mindste vil der som før nævnt blive foretaget $\lceil \log_2(d) \rceil$ sammenligninger. Fordi antallet af niveauer af fletninger er $\lceil \log_d(n) + O(1) \rceil$, vil antallet af sammenligninger holde sig på $n\log_2(n) + O(n)$ totalt set. Dette fremgår af nedenstående udregning

$$\begin{aligned} & n \cdot \log_2(d)(\log_d(n) + O(1)) \\ = \; & n \cdot \log_d(n) \cdot \log_2(d) + O(n) \cdot O(1) \\ = \; & n \cdot \log_2(n) + O(n). \end{aligned}$$

Det kan derfor konkluderes, at antallet af sammenligninger ved brug af selektionstræet ikke forøger antallet af sammenligninger. Men da antallet af niveauer i fletningen nu kun vil være $\log_d(n)$, vil det samlede antal flytninger blive reduceret til $2/\lceil \log_2(d) \rceil) n\log_2(n) + O(n)$. Katajainen et al. giver i [3] en grundig analyse af både antallet af sammenligner, flytninger og den samlede køretid, den interesserede læser henvises hertil for fordybelse.

### 6.3.2 4-vejs-fletning med array

I denne variant bliver der i stedet for et selektionstræ benyttet et array til udvælgelsen af det mindste element. Det bliver gjort ved at have et array af $d$ pointere der peger til hver sin sekvens. Hver gang det mindste element i gruppen tages ud findes det ved sammenligning af de $d$ elementer som arrayet peger på. Pointeren der pegede på det element, der blev taget, bliver opdateret ved at pege på næste element fra den samme sekvens. Denne procedure fortsættes, indtil alle sekvenser er tomme.

Fordi der bliver anvendt pointere i arrayet, bliver der ikke foretaget unødige flytninger af elementer. Det kan derudover ses, at der, hver gang det mindste element udtages, bliver lavet højst $d - 1$ sammenligninger. Det kræver ikke nogen sammenligninger at oprette arrayet af pointere, som det gjorde for selektionstræet. Antallet af niveauer i fletningen er $\lceil \log_d(n) \rceil + O(1)$, og antallet af elementer, der skal tages ud per niveau er antallet af elementer der flettes. Antallet af sammenligninger vil derfor være:

$$(d-1)\cdot(\lceil \log_d(n) \rceil + O(1))\cdot n = (d-1)\cdot\lceil \log_d(n) \rceil \cdot n + O(n) = \frac{(d-1)}{\log_2(d)} \cdot n\log_2(n) + O(n).$$

Dette betyder at antallet af sammenligninger forøges når et array benyttes, i forhold til $d$-vejsfletning med selektionstræ. Som det ligeledes kan ses, er

forøgelsen af sammenligninger bestemt af $d$, hvilket sætter en begrænsning for hvor stor $d$ kan være, før antallet af sammenligninger har fatal indflydelse på ydelsen af algoritmen. Ved en 4-vejs fletning er konstanten sammenligningerne forøges med lig $(4-1)/\log_2 4 = 1.5$.

### 6.3.3 Fletningsdelen

Når de to deltabeller $Q$ og $P$ er sorteret med den netop beskrevne metode, skal de flettes sammen. Denne fletning kunne gøres ved simpel 2-vejsfletning, men idet $P$ igennem algoritmen halveres, og $Q$ konstant forøges, er dette ikke den bedste løsning. I stedet er det bedre at benytte en binær fletterutine. Denne flettemetode bygger på den kendsgerning, at den statistiske fordeling af elementernes værdier gør, at der for hvert element i $P$ vil være $|Q|/|P|$ elementer, der statistisk set svarer til denne værdi. I stedet for at sammenligne de to mindste elementer ved hver fletning, ville det være bedre at sammenligne første element i $P$ med det element der ligger på den $(|Q|/|P|)$'ende plads i $Q$. På den måde ville der optimalt blive flyttet $|Q|/|P|$ elementer fra $Q$ og et element fra $P$ efter hver sammenligning.

Den binære fletning udnytter overstående iagttagelse for to givne deltabeller $X$ med størrelse $n$ og $Y$ med størrelse $m$ der skal flettes; det antages at $n \leq m$. Det første element i $X$ der tages ud, kaldet $x_1$, sammenlignes nu med elementet i $Y$ bestemt udfra $m/n$, kaldet $y_{m/n}$. Hvis $x_1 \geq y_{m/n}$ kan alle elementer mindre end $y_{m/n}$ flyttes til arbejdsområdet, og dette omfatter elementerne $[1, m/n]$ i $Y$. Herefter gentages fletterutinen, men nu med to deltabeller $X$ der ligger mellem $[1, |X|]$ og $Y$ der nu er blevet $(m/n)$ elementer mindre. Hvis derimod at $x_1 < y_{m/n}$ skal pladsen til $x_1$ findes i de $[1, (m/n) - 1]$ elementer i $Y$. Dette bliver gjort med binær søgning. Når pladsen er fundet, lad os sige f.eks. $y_{k-1} \leq x_1 < y_k$, vil først $[1, k-1]$ elementer i $Y$ blive flyttet til arbejdsområdet og derefter $x_1$. Fletningen fortsætter derefter med $[2, |X|]$ elementer af $X$ og $[k-1, |Y|]$ elementer af $Y$.

Ved nærmere analyse kan det ses, at det største antal sammenligninger, der bliver foretaget i den binære fletning af to deltabeller af størrelse $m$ og $n$ er $O\left(n \log_2 (m/n)\right)$. Dette er tilfældet, hvis alle $n$ elementer er mindst. Da der vil blive foretaget en binær søgning for hvert element i $X$ gennem de $m/n$ første elementer i $Y$, hvilket forårsager $O(log_2 \left(\frac{m}{n}\right))$ sammenligninger. I den binære fletning, bliver hvert element i de to deltabeller flyttet een gang. Hvis der benyttes den før omtalte flytteteknik, der benytter et "hul" i flytningen, kan det samlede antal flyt derved blive holdt nede på $2 \cdot (m+n+1)$ flyt.

Det samlede antal sammenligninger, som fletningsdelen bevirker igennem hele algoritmen, kan udregnes på følgende måde: Fletningen bliver udført mellem først $\lfloor \frac{n}{2} \rfloor$ og $\lfloor \lceil \frac{n}{2} \rceil / 2 \rfloor$ elementer, i næste iteration bliver den kaldt med $\lfloor \frac{n}{2} \rfloor + \lfloor \lceil \frac{n}{2} \rceil / 2 \rfloor$ og $\lfloor \lceil \lceil \frac{n}{2} \rceil / 2 \rceil / 2 \rfloor$. Dette kan skrives som en talrække og sammenfattes til en summering således:

$$O\left(\frac{n}{4}\log_2(2) + \frac{n}{8}\log_2(6) + \cdots + \frac{n}{2^i}\log_2(2^i - 2) + \cdots\right).$$

Hvilket kan udtrykkes som:

$$O\left(n\sum_{i=1}^{log_2 n}\frac{i}{2^i}\right) = O(n).$$

Der kan på samme måde opstilles en talrække for antallet af flytninger.

$$2((\frac{n}{2} + \frac{n}{4}) + (\frac{n}{2} + \frac{n}{4} + \frac{n}{8}) + \cdots) + O(n).$$

Hvilket tilsammen giver $2n\log_2(n) + O(n)$.

### 6.3.4 Afrunding af sorteringen

Man kunne vælge at afslutte sorteringen ved blot at lade partitioneringen fortsætte, indtil der kun var eet element tilbage i $P$. Denne metode vil fremover kaldes simpel inplace fletsortering. Dette er dog ikke hensigtsmæssigt da meget små deltabeller derved vil blive flettet med den store deltabel $Q$. For at undgå dette bliver partitioneringen afsluttet inden deltabellen bliver lille. I [3] vælges der at fortsætte partitioneringen indtil $P$ bliver mindre end $n/\log_2(n)$. Når $P$ når denne størrelse bliver $P$ sorteret for sig selv. Sorteringen af $P$ sker med simpel inplace fletsortering.

Med denne stopklods på flettedelen, gør Katajainen et al. opmærksom på, at der nu kun bliver udført $\log_2(\log_2(n))$ fletninger. Det totale antal flytninger under fletningen bliver derfor reduceret til $2n\log_2(\log_2(n))$. Når fletningen stopper, kan sorteringen af $P$ efterfølgende gøres i lineær tid. Til den simple inplace flettesortering skal $P$ benytte et arbejdsområde, derfor bliver pladsen der skal sorteres $2n/\log_2(n)$. Den linære køretid skyldes at den simple inplace flettesortering sorterer i $n\log_2 n + O(n)$ og $|P'| = 2n/\log_2(n)$. Tiden er derfor:

$$\frac{n}{\log_2(n)}\log_2\frac{2n}{\log_2(n)} = \frac{n\log_2(2n) - n\log_2(\log_2(n))}{\log_2(n)} = O(n).$$

På figur 4 er illustreret den sidste procedure i sorteringen. På figuren ses først situationen på tidspunktet hvor partitioneringen stopper. Derefter bliver den venstre deltabel, her kaldet $X$, sorteret som netop beskrevet.

De to deltabeller $X$ og $Y$ mangler nu kun at blive flettet sammen. Problemet er dog, at der ikke som før er noget arbejdsområde til rådighed, for at fletningen kan udføres uden brug af inplace fletning. Arbejdsområdet bliver derfor skabt ved at finde de $n/\log_2 n$ mindste elementer i $X$ og $Y$ tilsammen, elementerne bliver her kaldet $x$ og $y$. Herefter skifter $x$ og deltabellen

Figur 4: Hvorledes algoritmen afslutter sorteringen.

$[X - x, |X|]$ plads, derved ligger de $n/\log_2 n$ elementer ved siden af hinanden. På den måde er arbejdsområdet skabt, da vi ved at de $n/\log_2 n$ mindste elementer skal ligge i område $[1, n/\log_2 n]$ af de $n$ elementer. Og da det er her de bliver flyttet hen ved fletning af $X - x$ og $Y - y$ kan fletningen foretages. Efter fletningen er foretaget, er $x + y$ ikke sorteret på grund af, at flyttemetoden som før beskrevet ikke er stabil. Det er derfor nødvendigt at sortere denne del, hvilket bliver gjort med simpel inplace fletsortering.

## 6.4 Udredning af køretid

Det er ikke vanskeligt at se, at algoritmen har en asymptotisk køretid på $O(n \log_2(n))$, da delproblemerne i hver iteration halveres. En grundigere analyse af antallet af sammenligninger og flyt er dog interressant, og dette kræver lidt større arbejde. Løbende under fremlæggelsen af selve algoritmen er der blevet udregnet antal sammenligninger og flytninger for de enkelte delprocedurer. Disse dele mangler nu blot at blive kædet sammen.

For at starte med antallet af sammenligninger, så må denne beregning laves for begge varianter af algoritmen, nemlig den med selektionstræ og den med array. I algoritmen der benytter et selektionstræ, blev det i afsnit 6.3.1 vist, at antallet af sammenligninger i sorteringsdelen holder sig på $n \log_2(n) + O(n)$. I fletningsdelen blev det vist i afsnit 6.3.3, at antallet af sammenligninger blev holdt nede på $O(n)$. Da afslutningen af algoritmen også også tager $O(n)$ sammenligninger, er det samlede antal sammenligninger, der skal bruges i algoritmen, begrænset opadtil af $n \log_2(n) + O(n)$.

Ved benyttelse af et array i fletningen stiger antallet af sammenligninger til $\frac{(d-1)}{\log_2(d)} \cdot n \log_2(n) + O(n)$, og da vi arbejder med 4-vejsfletning: $\frac{3}{2} n \log_2(n) +$

$O(n)$. Derudover er antallet af sammenligninger identisk med varianten med selektionstræet. Det samlede antal af sammenligninger bliver bestemt af den ekstra konstant, der afhænger af $d$. Antallet af sammenligninger er derfor afgrænset opadtil af $\frac{3}{2}n\log_2(n) + O(n)$.

I forhold til flytninger er der ikke forskel på de to varianter, da antallet af flytninger i sorteringsdelen er ens. Antallet af flytninger kan derfor bestemmes generelt for dem begge. Som det er vist i afsnit 6.3, er antallet af flytninger for sorteringsdelen lig $(2/\lceil\log_2 d\rceil)n\log_2(n) + O(n)$ flytninger, hvor $d \geq 2$ er en konstant. I fletningsdelen er antallet af flytninger $2n\log_2\log_2(n) + O(n)$. Antallet af flytninger i hele algortimen er derfor domineret af antallet af flytninger i sorteringsdelen.

Når vi arbejder med $d = 4$, så vil antallet af flytninger blive $n\log_2(n) + O(n)$. Generelt kan vi sige, at større værdier af $d$ vil give os færre flytninger og en bedre køretid, sålænge $d \ll n$. Hvis $d$ bliver for stor i forhold til $n$ holder denne betragtning ikke længere, da flere led i de foregående beregninger vil få en helt anden vægt. Hvis $d$ er af størrelsesordenen $O(n)$ vil selektionstræet få en højde på $\log_2(n)$ og selve fletningen ville pludselig udgøre en sortering i sig selv.

### 6.5 Implementation af 4-vejs-flettesortering

Implementationen af de to 4-vejs-flettesorteringer er lavet på grundlag af Katajainen et al.'s implementation i forbindelse med deres udfærdigelse af [3]. Denne kode var skrevet i C, og vi har derfor omstruktureret koden, så den passer til STL. Som containere er der benyttet vectorer og der benyttet random access iteratore til at tilgå data i vectoren.

Udover modificeringen af kildekoden er der blevet tilføjet uddybende noter, da vi følte, at dette ville gøre læsningen af kildekoden mere overskuelig. Her vil vi blot gennemgå en enkelt detajle, som vi vil knytte an til senere.

I implementationen af sorteringen ved hjælp af arrayet i funktionen `i_four_way_merge` er der lavet en optimering i forhold til udtagelsen af det mindste element. Optimeringen består i, at der ikke laves unødige sammenligninger i tilfælde af, at en sekvens i fletningen er tom. Fra start er der fire sekvenser i fletningen, og der bruges tre sammenligninger for at finde det mindste element. Når en sekvens bliver tom, sammenlignes der nu kun på de tre resterende sekvenser; dette kræver nu kun to sammenligninger. På den måde skæres de tomme sekvenser væk til der kun er een sekvens tilbage, der kan flyttes direkte over i arbejdszonen.

## 7 Inplace flettesortering med færre flytninger

Denne algoritme blev først skitseret i af Katajainen & Pasanen i [4]. I forbindelse hermed blev den dog ikke implementeret. Her følger en redegørelse

for selve algoritmen og derefter vores implementation. Endelig vil vi afrunde afsnittet med en grundig analyse af algoritmens effektivitet.

## 7.1 Algoritmen

Den grundlæggende idé bag algoritmen er den samme som den funktion, der blev beskrevet i afsnit 6, men hvor denne funktion benyttede sig af 4-vejs merge er tilgangen her en mere generel $d$-vejs merge. Funktionen er overordnet splittet op i fire dele:

1. Først opdeles tabellen i to zoner: sorteringszonen og omkodningszonen — deres roller vil blive forklaret senere.

2. Dernæst sorteres sorteringszonen. Dette trin kan opdeles i to:

   **Sorteringsfasen:** Sorteringszonen opdeles i blokke, der sorteres individuelt med en $d$-vejs flettesortering.

   **Flettefasen:** De sorterede blokke flettes sammen med en inplace flettefunktion.

3. Herpå sorteres elementerne i omkodningszonen med en anden effektiv inplace sorteringsfunktion.

4. Endelig flettes de to zoner sammen med en inplace flettefunktion.

I denne algoritme må vi forudsætte, at et maksinord er $O(log_2 n)$ bits stort, for at kunne garantere, at den er inplace. I forhold til eksisterende maskiner sætter den en øvre grænse for, hvor store mængder tal algoritmen kan håndtere på $2^{32}$ eller måske $2^{64}$ elementer. Grænsen for en 64-bits maskine kan synes teoretisk, men det skal dog nævnes.

### 7.1.1 Opdeling i blokke

Første trin i processen er at splitte sorteringszonen op i blokke. Blokkene konstrueres, så deres længder bliver stigende potenser af to. De to første blokke har længden 1, den efterfølgende 2, så $4, 8, 16 \ldots$ Hvis $n$ ikke er en potens af 2, så udgør den sidste del af sorteringszonen en blok for sig —vi vil benævne den blok *restblokken* og de øvrige som lige blokke. Er $n$ en potens af 2, så udgør restblokken halvdelen af sorteringszonen og er dermed også en potens af 2, mens den anden halvdel deles op som beskrevet. Figur 5 viser opdelingen af sorteringszonen.

Med denne opdeling af sorteringszonen gælder der for enhver af de lige blokke, at dennes længde er lig summen af længden af alle de foregående, mens det gælder for restblokken, at den er højst lige så lang som den øvrige del af sorteringszonen, hvilket vi skal bruge senere.

Figur 5: Opdeling af sorteringszonen i lige blokke og restblokken.

### 7.1.2 $d$-vejssortering

Denne viden om blokkenes længde udnytter vi under sorteringen. Hvis vi starter med den største af de lige blokke, så kan vi bruge idéen fra afsnit 6 og sortere den ved at anvende den resterende del af sorteringszonen som arbejdsområde. Derefter kan vi tage den næststørste blok og gøre det samme osv.

Initielt består en blok af $m$ grupper med hver $d$ sekvenser af længden 1. Vi skal senere se på, hvordan vi bestemmer $d$, men indtil videre skal vi blot bemærke, at $d$ er en hel potens af 2. Sekvenserne flettes sammen, så blokken nu består af $m/d$ grupper med længden $d$. Da både længden, $l$, af de lige blokke og $d$ er toerpotenser, vil der være et helt antal grupper i $l$ så længe $m$ er større end 1, dvs.: $l\%(m \cdot d) = 0$ for $m > 1$. Det betyder at vi, sålænge antallet af grupper er større end 1, kan blive ved at flette sekvenserne sammen til større sekvenser. Når der kun er én gruppe tilbage, vil der være $d$ eller færre sekvenser. Disse flettes sammen, hvorefter blokken er sorteret. Når restblokken sorteres, så må vi i hver runde af sorteringen tage specielt hensyn til den sidste gruppe, da det ikke er sikkert, det er en fyldt gruppe.

### 7.1.3 Fletningen

Fletningen foregår, som det blev beskrevet i afsnit 6, ved hjælp af et træ. Hvor træet i det nævnte afsnit var af fast størrelse, afhænger størrelsen her af $d$. $d$ fastsættes som en potens af 2, der opfylder at $\log_2(n)/\log_2(\log_2(n)) \leq d < 2\log_2(n)/\log_2(\log_2(n))$. Da træets størrelse afhænger af $d$, kan vi nu se, at det også afhænger af $n$. Det kan umiddelbart lyde foruroligende mht. algoritmens egenskaber som inplace, at den kræver en datastruktur, hvis størrelse er betinget af $n$. Hvis bladene skulle indeholde hele forskydningen ned i en sekvens, så ville hvert blad skulle kunne repræsentere et tal af størrelsen $n$, og dermed udgøres af $\log_2(n)$ bits. Det ville igen betyde, at træet i værste fald sammenlagt ville kræve $\log_2(n)(2d-1) = 4\log_2(n)^2/\log_2\log_2 n - \log_2(n)$ bits.

For at komme omkring dette problem forslår Katajainen & Pasanen en måde, hvorpå man kan gemme noget af informationen i tabelen selv. Tanken er som følger: Når sekvenserne under fletningen bliver længere end $(\log_2(n))^2$ opsplittes de i moduler af $(\log_2(n))^2$ elementer. Vi fletter som før men nu med den ene forskel, at når et helt modul er blevet tømt, bytter den plads

Figur 6: Selektionstræ til brug ved fletning.

med det næste modul i rækken, så vi altid fletter fra en position i starten af sekvensen, det aktive modul. Samtidig bemærker vi, at da hvert blad i træet svarer til en sekvens og da alle sekvenser har samme længde, så kan vi finde starten af sekvensen ud fra bladnummeret og sekvenslængden. Figur 6 viser selektionstræet med de underliggende sekvenser og moduler. De stiplede linjer angiver implicitte indekser — dels fra bladene til starten af hver sekvens, dels mellem en barneknude og dens forældre. De fuldtoptrukne linjer i træet viser det princip, der blev beskrevet i afsnit 6, hvor alle bladene peger på det mindste element i det sekvensen, mens alle indre knuder peger på det blad, der peger på det mindste element. De fede pile indikerer det modul, der skal skiftes ind, når hele det aktive modul er flettet over i arbejdsområdet. I sekvensen til højre er der ingen ventende blokke, hvorfor pilen ikke peger nogen steder hen.

Med denne struktur bliver den information, der skal gemmes i hvert blad begrænset til en forskydning ned i et modul af størrelsen $O(\log_2(n)^2)$, hvilket kan repræsenteres med $\log_2(\log_2(n)^2) = 2\log_2(\log_2(n))$ bits. Med denne nye størrelse af knuderne bliver den samlede størrelse af træet i værste fald kun $2\log_2(\log_2(n))(2d-1) = 8\log_2(n)$. Da vi forudsatte, at et maskinord er $O(\log(n))$ kan hele træet således ligge i om ikke ét så et konstant antal maskinord.

Det eneste vi mangler for at kunne virkeliggøre denne løsning er et redskab til at holde styr på, hvilket modul i sekvensen, der er det næste, der skal skiftes ind. I trin 1 i sorteringsprocessen opdelte vi tabellen i 2 zoner: sorteringszonen og omkodningszonen, og det er her vi får brug for omkod-

ningszonen.

Når omkodningszonen konstrueres, så flyttes elementerne rundt, så de er parvist forskellige. Vi kan nu fortolke områder i zonen som en bitstreng; hvis det største tal i et talpar står forrest, så skal talparret fortolkes som 1, ellers som 0. Dette kan vi udnytte til at gemme indekset på et modul som en bitstreng i omkodningszonen; med et *felt* i zonen vil vi mene en bitstreng, der repræsenterer et indeks. Zonen kommer til at indeholde indekset på de moduler, der skal skiftes ind, når det aktive modul er tomt. For at kunne dække $d$ sekvenser i hele sorteringszonen, må omkodningszonen bestå af $d\lceil \log_2(n) \rceil$ bits, dvs. $2d\lceil \log_2(n) \rceil$ elementer. Sorteringszonen kommer så til at bestå af de resterende $n - 2d\lceil \log_2(n) \rceil$ elementer.

## 7.2 Afsluttende fletning og sortering

Når alle blokkene er blevet sorteret, skal de flettes sammen til én sorteret sekvens. Til denne opgave benyttes en effektiv inplace flettealgoritme. Vi skal senere se på, hvad effektiviteten af denne har at sige for det samlede antal flytninger i funktionen.

Efter endt fletning sorteres omkodningszonen med en inplace sorteringsfunktion. Til dette formål har vi benyttet algoritmen fra afsnit 6.

## 7.3 Implementation

I det følgende skal vores implementation af den skitserede algoritme blive beskrevet. Vi har langt hen ad vejen fulgt denne, men enkelte steder har vi valgt at gribe tingene lidt anderledes an. Det vil være for omfattende at gå i detaljer med implementationen, men vi vil prøve at give et overblik, der gør det muligt at forstå kildefilerne. Disse er rigt kommenterede, og kommentarerne skulle lægge sig op ad det følgende. Kildekoden for håndteringen af omkodningszonen findes i bilag C.4.2, mens kildekoden for den øvrige del af funktionen findes i bilag C.4.1

Det skal nævnes med hensyn til vores implementation, at den ikke sorterer korrekt for sekvenser på en længde under ca. 200 elementer. Det har ikke haft nogen praktisk betydning for vores test, da det ikke har haft interesse at sortere så små sekvenser. Som det vil fremgå, har funktionen i dens nuværende udformning ingen praktisk interesse, hvorfor denne begrænsning ikke er af større betydning.

### 7.3.1 Omkodningszonen

Omkodningszonen skal understøtte tre funktionaliteter: Oprettelsen af zonen, læsning af et nummer samt skrivning af et nummer. Læsningen og skrivningen er essentielt set bare at skrive eller læse nogle bits. Vi vil ikke komme nærmere ind på implementationen her, men for de interesserede henvises der til funktionerne `make_number` og `read_number` i bilag C.4.2.

Mht. oprettelsen af zonen foreslår Katajainen & Pasanen i sin artikel, at man først finder medianen af alle elementerne, hvorefter man partitionerer omkring dette element. Man har nu en let måde at kombinere talparrene på, da alle elementerne i den lave partition er mindre end elementerne i den høje partition. Der henvises til, at begge algoritmer kræver $O(n)$ sammenligninger og flytninger, men vi må også være opmærksomme på, at der udføres væsentligt flere end nødvendigt er. Ideelt set flyttede man bare tallene på plads, så man i værste fald skulle foretage $O(d\lceil \log_2(n) \rceil)$ flytninger.

Vi forsøger at opnå dette ideal ved hjælp af to pointere. Den ene pointer peger på det første element i omkodningszonen, mens den anden pointer gennemløber resten af tabellen for at finde et element, der er forskelligt fra det første og dette element skiftes ind ved siden af det første. Derpå udfører vi samme procedure for den næste tupel indtil omkodningszonen er fuld. På grund af den gentagne gennemløbning af tabellen, ser det umiddelbart ud som om, at der udføres sammenligninger i størrelsesordenen $O(nd\lceil \log_2(n) \rceil)$, det er dog ikke tilfældet; i værste fald skal der bruges $n-1$ sammenligninger på at konstruere zonen: Værstefaldet opstår, hvis det første element er lig de efterfølgende $n-(2d\lceil \log_2(n) \rceil - 1)$ elementer. Når den næste tupel skal konstrueres behøver vi imidlertid kun gennemsøge de sidste $2d\lceil \log_2(n) \rceil - 1$ elementer, da vi ved at de øvrige elementer alle er ens. Hvis de sidste $2d\lceil \log_2(n) \rceil - 1$ elementer alle er forskellige fra de øvrige elementer, så vil det tage yderlige $2d\lceil \log_2(n) \rceil - 1$ sammenligninger at konstruere omkodningszonen. Hvis det ikke er tilfældet, så har vi ikke nok elementer til at lave en omkodningszone. Til gengæld har vi en næsten sorteret tabel, idet mere end $n - d\lceil \log_2(n) \rceil$ elementer er ens, og de øvrige elementer vil alle være samlet i omkodningszonen. For at sortere tabellen behøver vi da blot at sortere omkodningszonen og derpå flytte den øvrige del af tabellen ind på den rette plads i den sorterede sekvens. Sorteringen skal selvfølgelig være inplace, så til formålet benytter vi os af algoritmen beskrevet i afsnit 6.

Antallet af flytninger i værste fald maksimeres i det tilfælde, hvor samtlige elementer i omkodningszonen er ens. I dette tilfælde findes der ingen gyldige tupler indenfor zonen, og vi må derfor foretage en ombytning af elementer for hver tupel. Det værste antal flytninger bliver derfor $3d\lceil \log_2(n) \rceil$.

Selve implementationen af den beskrevne metode findes i funktionen `make_encoding_zone`, se bilag C.4.2.

### 7.3.2 Fletningen

Som det blev antydet i forrige afsnit, så er fletteprocessen stort set den samme, hvad enten vi fletter med moduler eller ej. Når der flettes med moduler, skal der dog udføres nogle flere operationer end ellers. For at undgå de unødvendige operationer, der ville opstå, hver gang vi fletter sekvenser, der ikke er opdelt i moduler, har vi lavet to flettefunktioner; en der bruger moduler (`d_way_merge_blocks`), og en der klarer sig uden (`d_way_merge`). I

det følgende vil vi kun gennemgå den første, da funktionaliteten af den anden er en ægte delmængde af den første. Kildekoden til funktionerne findes i bilag C.4.1.

Essensen i fletningen er udvælgelsestræet, som vi bruger til at finde det mindste element blandt de $d$ sekvenser. Idéen er allerede beskrevet i afsnit 6, men vi vil her uddybe det lidt.

Der er $d$ blade i træet, og hvert blad er knyttet til en af sekvenserne. Et blad indholder et afsæt ind i sekvensen, der svarer til, hvor mange elementer, der allerede er flettet. De indre knuder på næstnederste niveau indeholder indekset på den af børnene, der peger på det mindste element. De øvrige indre knuder sættes til at pege på det samme blad som den af børnene, der peger på det blad, der svarer til det mindste element. Herved kommer rodknuden i træet til at blive en indirekte henvisning til det mindste element i de $d$ sekvenser -og dermed det næste element, der skal bruges i fletningen.

Når det mindste element er fundet, så flyttes det over i arbejdsområdet, i enden af den sorterede følge. Derpå opdateres forskydningen i det blad og den sekvens, hvor elementet stammede fra. Værdien af det nye mindsteelement i sekvensen bruges til at opdatere træet, så rodknuden igen vil pege på det mindste element i samtlige sekvenser.

Når en sekvens er tom, så lader vi det tilsvarende blad indeholde -1. Når nu blade med samme forælder indeholder -1, så vil det forplante sig op til forælderen. På denne måde vil -1'ere vandre op igennem træet efterhånden som sekvenserne udtømmes. Efter det sidste element er taget ud, så vil -1 have nået rodknuden i træet.

I opdateringen af træet behøver vi ikke fortsætte længere op, hvis værdien af en knude ikke ændrer sig under opdateringen —det betyder at ingen knuder på højere niveauer vil ændre sig. Løsningen ville kræve lidt ekstra forespørgsler på indeks i træet, men vi ville spare en række sammenligner af elementer i sekvenserne, se desuden afsnit 8.4 for en videreudvikling af denne idé på træer af fast størrelse.

Træet selv bliver gemt i et array. På denne måde kan vi bruge den metode, der bliver beskrevet i [1] til at finde henholdsvis børn og forældre til en given knude: Hvor $i$ er indekset på knuden, så vil det venstre barn altid befinde sig på position $2i$ og det højre på $2i+1$. Forælderknuden findes som $\lfloor i/2 \rfloor$.

Da $d$ er en toerpotens, vil træet være et balanceret binært træ. Vi behøver således ikke tage højde for ikke-eksisterende knuder eller blade. Når den sidste gruppe i en blok skal flettes, så kan vi dog risikere, at der ikke er $d$ sekvenser. For stadig at have et generelt træ i dette tilfælde, finder vi den nærmeste toerpotens, der er større end antallet af sekvenser, og så fylder vi alle de blade, der ikke svarer til en sekvens med -1.

I afsnit 7.1.3 så vi, at træet kunne ligge i et konstant antal maskinord. Vi har dog tilladt os ikke at pakke bitrepræsentationen af træet, og i stedet valgt at bruge et ord pr. knude. På denne måder er funktionen strengt taget

ikke inplace, men vi mener vi kan tillade os det, da pladsforbruget stadig vil være minimalt, $O(d)$. Der vil ligge en del ekstra arbejde i at lave en pakket repræsentation af træet, så den valgte løsning giver en nedre grænse for, hvor effektivt arbejdet med træet er.

Det første der skal ske er, at omkodningszonen skal initialiseres. Alle felter sættes til at pege på det andet modul i hver sekvens.

Når vi begynder at flytte elementer fra træet over i arbejdszonen, så må vi for hvert nyt element undersøge, om den sekvens, elementet kom fra, er tom. Dette er nødvendigt for at kunne opdatere træet; hvis sekvensen er tom, skal vi ikke prøve at finde et nyt element derfra. Da træet kun indeholder information om, hvorvidt det aktive modul er tom, skal denne oplysning hentes fra omkodningszonen. Hvis omkodningsfeltet indeholder et indeks til et ventende modul, så kan sekvensen ikke være tom, ellers markeres sekvensen som tom.

Denne tilgang til problemet betyder, at vi for hvert element, der flettes ind i arbejdszonen må læse et felt. For at afkode et felt, når sekvenslængden er $l$, skal der bruges $\log_2(l)$ sammenligninger. Det er en stor faktor at have i hver iteration, og en mere hensigtsmæssig løsning ville være at gemme denne oplysning i en enkelt bit i forbindelse med udvælgelsestræet. Det samlede antal bits ville kunne ligge i ét ord, og funktionen ville stadig være inplace. Vi må dog bemærke at denne uhensigtsmæssighed kun giver os flere sammenligner, ikke flytninger, og da det primære formål med denne algoritme er at reducere antallet af flytninger, har vi tilladt os at se bort fra problemet. Hvis man ville optimere køretiden for funktionen, så skulle det selvfølgelig rettes. Tanken vil blive taget op igen i afsnit 8.2.

Når det mindste element er fundet, så opdateres træet. Hvis den nye rodknude er -1, så er alle sekvenser tomme og fletningen kan stoppe. Vi kan nu lukke hullet i arbejdszonen med det sidst fundne element og kopiere det gemte element fra arbejdszonen ind på den tomme plads i sorteringszonen. Hvis træet ikke er dødt, så skal det fundne mindsteelement bytte plads med et element fra arbejdszonen. Dernæst må vi kontrollere om det aktive modul er tomt, og hvis den er, så må vi, hvis sekvensen ikke er tom, skifte et nyt modul ind. Med et nyt modul må træet opdateres igen, så det afspejler det nye mindsteelement i sekvensen. Med denne fremgangsmåde vil vi, når vi skifter moduler, komme til at foretage to opdateringer af træet. Fordelen er dog, at vi kun behøver at konsultere rodknuden for at finde ud af, om fletningen er færdig. I modsat fald ville vi være nødt til at undersøge hver sekvens hver gang, for at undersøge om alle sekvenser er tomme.

Efter træet er opdateret, kan vi uddrage et nyt mindsteelement fra rodknuden, og så kører forløbet forfra.

### 7.3.3 Flettesorteringen

Funktionen `d_way_sort` kaldes på blokkene i sorteringszonen. Den koordinerer fletningen ved at iterere henover blokken og kalde flettefunktionen på stadig større grupper. Når sekvenserne overstiger $(\log_2(n))^2$ så skifter den fra at bruge `d_way_merge` til at bruge `d_way_merge_blocks`. Da vi gentagne gange fletter frem og tilbage mellem blokken og den resterende del af sorteringszonen, er det vigtigt at holde styr på, om elementerne ligger på deres rette plads efter den sidste fletning. Hvis det ikke er tilfældet må vi foretage en ekstra flytning af alle elementerne, så vi kan bevare strukturen i sorteringszonen.

Da det er den samme funktion, der står for sorteringen af restblokken såvel som de lige blokke, så må der tages særligt hensyn til den sidste gruppe i hvert gennemløb, da denne gruppe ikke nødvendigvis består af $d$ sekvenser. At vi griber det således an betyder, at vi her, såvel som i flettefunktionen bliver nødt til at foretage en række test på, om vi har med den sidste blok at gøre. Flettefunktionen skal endvidere tage højde for, om det er den sidste gruppe i blokken der bliver sorteret. For at komme ud over alle disse forespørgsler kunne man konstruere yderligere instanser af de forskellige funktioner, der var specialiseret til hver deres situation. Opdelingen ville forbedre køretiden, men ville ikke ændre ved det primære mål; at nedbringe antallet af flytninger.

Inspireret af afsnit 3.3, og med baggrund i indsættelsessorteringens effektivitet for relativt små sekvenser, kunne man foreslå at benytte indsættelsessortering på grupperne i første gennemløb i stedet for at flette dem. Alt efter gruppestørrelsen kunne man endog sortere $d$ grupper sammen på denne måde, for herved at springe de to første omgange af fletningen over.

Vi har forsøgsvist prøvet os med denne model, og det viste sig, at den, ved sortering af 1 million elementer, forbedrede køretiden med 1-2%. I dette forsøg sorterede vi grupperne en ad gangen, men det ville givetvis give en endnu bedre køretid, hvis man kørte over $d$ grupper. Det kunne være interresant som en fortsættelse af projektet at undersøge, hvor mange runder med fletning der skulle erstattes af indsættelsessortering, for at give det optimale resultat. Ud fra analyserne i afsnit 5.2 kunne det tyde på, at det var 2. Vi har dog ikke forfulgt denne tanke yderligere.

### 7.3.4 Det ydre funktionskald

Med de skitserede funktioner kan vi nu håndtere sorteringen af blokkene i sorteringszonen. Som det fremgik af listen i afsnit 7.1 så mangler vi nu blot at flette blokkene sammen, sortere omkodningszonen og endelig flette de to zoner sammen.

I [4] henviser Katajainen & Pasanen til en assymptotisk optimal flettealgoritme, men som de selv bemærker, så er det antal flytninger og sammenligninger, der forgår under fletningen, negligibel i forhold til sorteringsdelen.

Vi har derfor ikke brugt tid på at implementere denne funktion, men blot anvendt den inplace flettealgoritme, der ligger i STL. Hvis funktionen skulle integreres i CPH STL , så skulle vi selvfølgelig anvende algoritmen herfra. Diskussionen vil blive taget op igen i afsnit 8.1.1.

Det er under alle omstændigheder ikke den optimale løsning blindt at bruge inplace fletning på blokkene. På dette tidspunkt i algoritmen skal vi ikke længere bruge omkodningszonen. Det betyder, at vi kan bruge denne zone på $2(\log_2(n))^2$ elementer som arbejdsområde under fletningen. Da man kan flette to sekvenser af længde $m$ og $n$ med kun $\min\{m, n\}$ plads [3], og da blokkene ligger i stigende potenser af 2, betyder det at blokke op til en størrelse af $(2\log_2(n)^2)^2$ kan flettes med omkodningszonen som arbejdsområde. Vi har ikke undersøgt, hvordan arbejdsområdet kan bruges til at nedbringe antallet af flytninger for længere sekvenser, men det må formodes, at man vil kunne udnytte det ekstra arbejdsområde.

Under fletningen fletter vi hele tiden den første blok med den næstfølgende. Den første blok vil således hele tiden blive fordoblet, og dermed have samme længde som den efterfølgende blok, hvorved vi statistisk set får minimeret antallet af sammenligninger i gennemsnitstilfældet.

Her kunne man igen vælge at bruge indsættelsessortering til at sortere de mindste blokke. Men da det kun vil ske én gang for hver blok, så vil gevinsten praktisk taget være ikke eksisterende.

Til sortering af omkodnings zone benyttes algoritmen fra afsnit 6 og endelig skal de to zoner flettes sammen, og dette kan kun gøres med en inplace flettefunktion.

## 7.4 Analyse

Vi har nu set på, hvordan algoritmen virker, og hvordan vi har konstrueret den i praksis. I det følgende skal vi prøve at samle op på alle de overvejelser vedrørende effektivitet, vi har gjort os i de foregående afsnit. Vi vil kigge på sammenligninger og flytninger hver for sig. I det følgende vil $l$ betegne sekvenslængden.

### 7.4.1 Sammenligninger

Lad os først se på antallet af sammenligninger, der bliver udført i $d$-vejssorteringen af en sekvens af længde $m$, da denne funktion må formodes at være determinerende for det samlede antal sammenligninger.

I hver runde af fletningen skal vi opbygge udvælgelsestræet, det kræver $d$ sammenligninger. Da der i hver iteration over sekvensen skal opbygges $m/(dl)$ træer, og da der ialt er $\log_d(m)$ iterationer, vil antallet af sammenligninger være $O(m \log_d(m))$

Under fletningen har vi to tilfælde, med og uden moduler. Når vi fletter uden moduler, så udfører vi $d - 1$ sammenligninger, når vi opretter udvæl-

gelsestræet, og $\log_2(d)$ sammenligninger, når vi opdaterer det. I hver runde af fletningen bliver der udført $m$ opdateringer af træet, da $m$ elementer fjernes fra træet og lægges over i arbejdsområdet. Der skal oprettes udvælgelsestræer for hver gruppe, dvs. $m/(dl)$. Når vi fletter med moduler, skal vi læse et ord i omkodningszonen for hvert skift. En læsning koster $\log_2(l)$ sammenligninger, og da der højst vil være $m/l$ blokke i hver itaration, bliver det samlede antal sammenligninger på den konto $(m/l)\log_2(l)$ I hver runde af fletningen bliver antallet af sammenlinger:

$$\frac{md}{dl} + m\log_2(d) + \frac{m\log_2(l)}{l} = \frac{m}{l} + m\log_2(d) + \frac{m\log_2(l)}{l}.$$

Der itereres ialt $\log_d(m)+O(1)$ gange henover sekvensen. Da sekvenslængden $l$ afhænger af, hvilken runde vi er i, bliver vi nødt til at udfolde dette udtryk. Det samlede antal sammenligninger bliver da:

$$\begin{aligned}
& \log_d(m)\frac{m}{l} + m\log_d(m)\log_2(d) + O(m\log_2(d)) && (1)\\
= \;& m\sum_{i=1}^{\log_d(m)} \frac{1}{d^i} + m\log_2(m) + O(m\log_2(d)) && (2)\\
\leq \;& m\frac{d}{d-1} + m\log_2(m) + O(m\log_2(d)) && (3)\\
= \;& m\log_2(m) + O(m\log_2(d)). && (4)
\end{aligned}$$

Omskrivningen i linje (2) kommer af den betragtning, at $\log_d(m)\frac{1}{l}$ angiver det samlede antal sekvenslængder over alle iterationer. Sekvenslængden bliver i hver runde $d$ gange længere indtil den øvre grænse nås. At denne sum har grænseværdien $d/(d-1)$, kan indses, hvis vi betragter kvotientrækken; $\sum_{n=0}^{\infty}(d^{-1})^n$. Summen af denne række vil være en øvre grænse til vores sum:

$$\sum_{n=0}^{\infty}(d^{-1})^n = \frac{1}{1-d^{-1}} = \frac{d}{d-1}$$

Da summen af længden af alle blokkene i sorteringszonen er $n$ vil det samlede antal sammenligninger i sorteringsfasen være $n\log_2(n)+O(n\log_2(d)) = n\log_2(n) + O(n\log_2(\log_2(n)))$.

Under flettefasen, skal vi flette alle blokkene sammen. Der findes flere flettealgoritmer, der fletter lineært i længden af de to sekvenser, der skal flettes. Der er ca. $\log_2(n)$ blokke og vi fletter hele tiden de mindste blokke sammen. Vi fletter først to blokke med længden 1, så 2, 4 osv. Vi kan summere sammenligningerne over alle blokke, idet $2^i$ angiver summen af de to delsekvenser i det $i$'te flet fås:

$$\sum_{i=1}^{\log_2(n)} c2^i \qquad (5)$$

$$
\begin{aligned}
&= c \sum_{i=1}^{\log_2(n)} 2^i && (6)\\
&= c \sum_{i=1}^{\log_2(n)} 2^i && (7)\\
&= c(2^{\log_2(n)+1} - 2) && (8)\\
&= 2cn - c = O(n) && (9)
\end{aligned}
$$

I linje 8 udnytter vi, at summen er et afsnit af en kvotientrække, se **A.5** i [1].

Den afsluttende fletning har således ingen større indflydelse på det samlede antal sammenligninger. Vi så tidligere, hvordan vi kunne oprette omkodningszonen med kun $n-1$ sammenligner, hvilket også bliver domineret af sorteringen. Det sidste bidrag til sammenligningerne kommer fra sorteringen af omkodningszonen. Da omkodningszonen er $O(\log_2(n)^2/\log_2(\log_2(n)))$ vil køretiden for sorteringen blive stærkt domineret af $\log_2(n)^2$, som er $O(n)$.

Vi har således vist, at det samlede antal sammenligninger er domineret af sorteringen og kan sættes til $n\log_2(n) + O(n\log_2(\log_2(n))) + O(n)$.

Vi har i denne analyse ikke taget højde for de ekstra sammenligninger vi foretager i hver iteration af fletningen, som er forbundet med læsningen i omkodningszonen. Som nævnt koster det $\log_2(l)$ sammenligninger at foretage en læsning. Da der sammenlagt læses $m\log_d(m)$ bliver der i sorteringen af hver blok udført:

$$m\log_d(m)\log_2(l) = m\log_2(m)\frac{\log_2(l)}{\log_2(d)}$$

sammenligninger. Vi kan således se, da vi typisk har $l \geq d$, at denne fejl i implementationen mere end fordobler det samlede antal sammenligninger.

### 7.4.2 Flytninger

I forrige afsnit så vi grundige udredninger af, hvor mange sammenligner der udføres i algoritmen. Da mange af udredningerne vil være de samme for antallet af flytninger vil vi her opsummere en smule mere uformelt.

Under $d$-vejssorteringen flytter vi i hver iteration af fletningen hvert element 1 gang over i arbejdsområdet. Derforuden kan der forekommer flytninger, hver gang et nyt modul skal skiftes ind.

Når vi skal skifte et modul ind, så kan vi, hvis vi bruger hulteknikken nøjes med $2l + O(1)$ flytninger. Da der højst er $\lceil m/l \rceil$ blokke bliver antallet af flytninger i hver iteration $2m + O(m/l)$. Da vi også bruger hulteknikken til at flytte elementer over i arbejdsområdet — og skal flytte elementerne fra arbejdsområdet ind i sorteringszonen kræver dette $2m + O(1)$ flytninger. I hver iteration har vi $4m + O(m/l)$ flytninger, og da vi ialt har $\log_d(m) +$

$O(1)$ iterationer, så vil det totale antal flytninger under sorteringen være $4m \log_d(m) + O(m)$.

Antallet af flytninger i de afsluttende flettefacer og sorteringen af omkodningszonen vil være lineært i $n$ ud fra samme argumenter som i forrige afsnit. Katajainen & Pasanen redegør i [4] for, at $(1/2)\log_2(\log_2(n)) \leq \log_2(d)$. Det samlede antal flytninger bliver derfor:

$$
\begin{aligned}
& 4n \log_d(n) + O(n) && (10)\\
= \; & 4n \frac{\log_2(n)}{\log_2(d)} + O(n) && (11)\\
= \; & 8n \frac{\log_2(n)}{\log_2(\log_2(n))} + O(n). && (12)
\end{aligned}
$$

# 8 Eksperimentielle kørsler af algoritmerne

I det følgende vil de tre sorteringsalgoritmer, der teoretisk blev beskrevet i afsnit 6 og 7.4 blive testet eksperimentielt. Udover de tre inplace flettesorteringer har vi valgt at teste `bottom_up_heapsort`, som er beskrevet i afsnit 3.2.3. Dette er gjort fordi denne algoritme også er inplace, og derfor giver et godt sammenligningsgrundlag til flettesorteringerne.

Testen vil fokusere på tidsforbruget, antallet af sammenligninger og antallet af flyt som de enkelte algoritmer benytter for et givent input. Som teststrategi har vi valgt at lave to typer test af de tre nævnte testområder. Den ene test giver et godt overblik over udviklingen af algoritmerne ved at dække et stort antal elementer. Den anden test ser på udviklingen af algoritmerne indenfor et lille interval med færre elementer. Denne test er hovedsaglig interessant for `fewer_moves`. Generelt for begge test bliver der testet på tilfældigt genererede heltal, der ændres for hver deltest.

Næsten alle test er udført på en Athlon-XP 2400+ 2GHz maskine, med 256Mb DDR-RAM og Linux 2.4.20. I det ene tilfældet hvor en anden maskine er benyttet, er det angivet.

For at give et godt overblik over udviklingen af algoritmens ydelse bliver der for både tid, sammenligninger og antal af flytninger kørt en test der strækker sig fra 50.000 til 1.000.000 elementer med et interval på 50.000. Hver deltest mellem de 50.000 til 1.000.000 elementer bliver udført 50 gange, og det er gennemsnittet af de 50 test der vises. Resultaterne af de udførte test er opstillet som grafer.

Den anden test strækker sig over et mindre antal elementer og er kørt mellem 5.000 og 200.000 elementer med et interval på 5.000 elementer. I denne test er hver deltest også kørt 50 gange.

Udover de to beskrevne test af tid, sammenligninger og flytninger, er der til testen udviklet et testredskab, der er blevet benyttet til at teste enkelte detaljer i de tre flettesorteringerne. Selve værktøjet er beskrevet i afsnit 8.1 og benyttelsen af redskabet er uddybet i de følgende testafsnit.

## 8.1 Et særligt stopur

Vi har til brug under testningen udviklet et specialværktøj. Da vi har at gøre med rimeligt komplekse funktioner, der består af mange funktionskald, er det ønskværdigt at kunne dekomponere analysen af køretid, sammenligninger og flytninger i bidrag fra de forskellige dele af algoritmen. Specielt i afprøvningen af inplacesorteringen med færre flytninger, vil det være interresant at se, hvor i funktionen flytningerne typisk forekommer.

Der er to forskellige tilstande: Hvis man i sin testfil skriver `#define LOCAL_TIME` bliver statistikken bygget over køretiden i funktionerne, og hvis man skriver `#define LOCAL_COUNT` er det sammenligninger og flytninger.

Til hver tilstand er der knyttet en række variable, der skal være defineret i den testfil, der benytter modulet:

| COUNT |
|---|
| `extern unsigned long comparisons` |
| `extern unsigned long assignments` |
| `extern unsigned long function_m[ ]` |
| `extern unsigned long function_c[ ]` |
| `extern unsigned long start_move[ ]` |
| `extern unsigned long start_compare[ ]` |
| TIME |
| `extern double function[ ]` |
| `extern struct timeval start_time[ ]` |

Når der tælles skal man, hver gang der bliver udført en sammenligning eller en flytning i funktionen, inkrementere henholdsvis `comparisons` og `assignments`. Ved indgangen til alle større funktioner kaldes `start_count (FUNNUM)`, hvor `FUNNUM` er et nummer der identificerer funktionen, og ved udgangen kaldes `stop_count(FUNNUM)`. Funktionerne finder differencen på antallet af flytninger og sammenligninger fra `start_count(FUNNUM)` til `stop_count (FUNNUM)` og opdaterer det samlede resultat, der ligger i henholdsvis `function_m [FUNNUM]` og `function_c[FUNNUM]`. Hver af tabellerne skal således have en størrelse, der modsvarer det antal funktioner, man vil måle over. Efter funktionskaldet kan det samlede antal sammenligninger og flytninger i hver delfunktion findes i `function_m[FUNNUM]` og `function_c [FUNNUM]`.

Tidstagningen fungerer på samme måde som optællingen. Ved `start_count(FUNNUM)` gemmes systemtiden i `start_time[FUNNUM]` og ved `stop_count(FUNNUM)` gemmes differencen mellem slut og starttiden i `function [FUNNUM]`.

Det er vigtigt med hensyn til testen, at man kalder stop før funktionskald og start igen efter, så man kun får målt bidraget fra den kaldende funktion.

Da selve tidstagningen tager tid, kan værktøjet ikke bruges til en egentlig test af funktionens ydeevne, men det kan bruges til at vise de indbyrdes

forhold mellem dele af funktionen. Det kan således være med til at finde flaskehalse i programmet. Endelig må vi konstatere, at der vil være nogen usikkerhed på tidstagning. Meget korte funktionskald vil køre så kort tid, at den registrerede tid vil blive rundet af til 0. En funktion der kaldes mange gange men kun kører i korte intervaller kan således forsvinde fra køretidsstatistikken.

Desuden fremgik det af prøvekørslerne, at der af og til forsvandt 2% af sammenligninger og flyt. Fejlen skyldes formentlig manglende funktionskald til start og stop — vi prøvede at spore den, men det lykkedes ikke.

### 8.1.1 Test af antallet af flytninger

Til testen af antallet af flytninger i de tre beskrevne flettesorteringsalgoritmer er der lavet en generel test, som beskrevet i teststategien. Resultatet af denne ses i bilag B.1. Derudover er der lavet en test der tager et mere detaljeret udsnit af sorteringen af elementer mellem 0 til 100.000 elementer denne test er også vist i bilag B.1. Til testen af flettesorteringen med færre flytninger, skal det nævnes, at der ikke er blevet talt flytninger i inplacefletningen, der sidst i funktionen fletter de sorterede sekvenser sammen. Dette er en mangel i testen, som der skal tages højde for. Det skal dog nævnes at fletningen tager liniær tid se afsnit 7.4, og antallet af flytninger for en inplace fletning kan udføres med en øvre grænse på $3(n+m)+o(m)$ flytninger, hvor $m$ og $n$ er de to dele de flettes, hvilket er vist i [2]. Idet inplace fletningen er begrænset af $3n$, vil `fewer_moves` ikke overstige antallet af flytninger for `bottom_up_heapsort`. For at vise en tilnærmelse til antallet af flytninger som `fewer_moves` ville have, hvis flytninger i inplace fletning var registret, er der på testgrafen anført de tilnærmede antal flytninger for `fewer_moves`. Denne er genereret ved at tilføje $3n$ flytninger til testresultatet for `fewer_moves` og kaldes `fewer_moves_with_pseudo_merge`.

Der kan, testens kvalitet taget i betragtning, ses at `fewer_moves` rent faktisk benytter færre flytninger i sorteringen end de tre andre algoritmer.

Det viser, at på trods af, at hobsorteringen er dårligere end flettesorteringerne i værste tilfælde, så har den en bedre gennemsnitlig ydelse.

Derudover har de to 4-vejs-flettesorteringer samme antal flytninger, hvilket også var som forventet idet forskellen på de to algoritmer ikke skulle ændre på antallet af flytninger.

I forhold til testen af flytninger mellem 5.000 til 200.000 elementer, afbildet på figur B.1, kan det tydeligt ses på grafen over `fewer_moves`, at der omkring 60.000 elementer sker et dramatisk fald i hældingen på linien. Hvor `fewer_moves` starter ud med en hældning der svarer til `bottom_up_heapsort` bøjer linien af netop omkring de ca. $2^{16}$ elementer. Som Katajainen & Pasanen skriver er `fewer_moves` teoretisk set først funktionel ved sortering af elementer $n \geq 2^{16}$.

## 8.2 Test af sammenligninger

I testen af sammenligninger, der ses i bilag B.2, er det første der springer i øjene den store mængde sammenligninger som `fewer_moves` benytter i sorteringen. Det skal derudover nævnes at antallet af sammenligninger i inplace fletningen i `fewer_moves` ikke er inkluderet i testen. Antallet af sammenligningerer er mere end dobbelt så stort som `4_way_mergesort*` og næsten tre gange så stort som `bottom_up_heapsort` og `4_way_mergesort`. Teoretisk set er dette resultat ikke som forventet idet `fewer_moves` som det ses i afsnit 7.4 er garanteret en værste grænse på $n\log_2(n)) + O(n\log_2(\log_2(n))$ sammenligninger. Da dette er ca. samme grænse som for `4_way_mergesort`, burde forskellen ikke være så stor. Grunden til dette meget store antal af flytninger er der allerede givet en begrundelse for i afsnit 7.3.2. Her nævnes det at der i funktionen bliver udført unødig mange læsninger af tal i omkodningszonen, og hovdekilden til de mange sammenligninger må ligge her. Der er i forbindelse med diskusionen i afsnit 7.3.2, også givet et bud på hvordan antallet af flytninger kunne bringes ned. Set ud fra denne test ville det være en nødvendighed, hvis `fewer_moves` skulle blive en praktisk effektiv algoritme, at nedbringe netop antallet af sammenligninger. I teoriafsnittet fandt vi frem til, at fejlen i implementationen betød mere end dobbelt så mange sammenligninger. Hvis vi tager det ind i vores betragtninger af grafen, så kan vi se, at algoritmen kunne bringes til at give et hæderligt resultat, hvis fejlen blev rettet.

Som det også ses i testen af flytninger og tid opfører `fewer_moves` sig som forventet omkring $2^{16}$ elementer. Der ses ved antallet af sammenligninger at der sker et knæk på grafen lige omkring dette antal elementer.

Hvis man skal se på antallet af sammenligninger mellem de to 4-vejs-flettesorteringer holder testen sig i højere grad til teorien. Det ses at 4-vejs-fletsortering med array benytter flere sammenligninger end den med selektionstræet, og det er ikke overraskende.

## 8.3 Tidsforbrug

Vi afviklede alle funktioner i testsættet på den nævnte data. Grafen i bilag B.3 viser de fire sorteringsalgoritmers køretid som funktion af antallet af elementer der sorteres.

Det første man bør bemærke er hobsorteringens køretid i forhold til de tre flettesorteringer. I [3], der grundlaget for vores implementation af de to 4-vejs flettesorteringer, fremgår det, at køretiden på heapsort er noget bedre end de to flettesorteringer. Al teori henviser da også til, at hobsorteringen skulle være den mest effektive. Den benyttede hobsortering er den, der blev implementeret i afsnit 3.2.3 og i forbindelse med afprøvningen blev det vist, at dens køretid i forhold til Introsort passede med teorien. Samtidig stemmer de tre flettesorteringer godt overens indbyrdes. Vi har ledt langt og længe

efter noget, der kunne forklare dette forhold, men det er ikke lykkedes. Vi har endog forsøgt at kalde funktionerne i forskellig rækkefølge for at cache-udnyttelsen ikke skulle spille en rolle.

Ser vi bort fra denne opførsel, kan vi bruge testen til at sammenligne de tre flettesorteringer indbyrdes. Forholdet mellem de to 4-vejssorteringer er som forventet ud fra [3]. Den sidste er, som det blev forudsagt i [4], dårligere end de to andre med en konstant faktor. Grunden til en stor del af dette kan som nævnt findes i, vores uhænsigtsmæssige læsning i omkodningszonen. Det kan være svært at gisne om, hvor meget det vil betyde, at få elimineret dette, men set ud fra vores analyse vil det formentlig kunne bringe algoritmen op på et niveau, hvor den kan have interesse som alternativ til 4-vejssorteringen.

## 8.4 Sammenligning af de to varianter af 4-vejs-fletning

Som det kan ses i udfra grafen i bilag B.3, viser den samlede tidstest at 4-vejs-fletning med array er hurtigere end algoritmen implementeret med et selektionstræ. Det samme resultat kom de i [3] frem til. Katajainen et al. giver ikke noget decideret bud på hvorfor dette resultat opstår. Det fremgår af afsnit 6.3 at algoritmen med selektionstræet er teoretisk overlegen, så det kan undre en at den praktiske test ikke afspejler dette. For at få en ide om hvad resultatet kan skyldes er der blevet lavet en lokal test i de to algoritmer. Testen måler den procent vise del af det samlede tidsforbrug ved sortering af 10 millioner elementer, som bliver brugt henholdsvis i selektionstræet og i arrayet[5]:

| | Samlet køretid | Flettetid | Flettetid / % | Resttid |
|---|---|---|---|---|
| Array | 8,48s | 5,16 | 60,8 | 3,32 |
| Træ | 10,93s | 7,67 | 70,2 | 3,26 |
| Indbyrdes forhold | 1,30 | 1,49 | 1,15 | 0,98 |

Denne test viser at programmet med arrayet bruger 61% af den samlede køretid med fletning, mens algoritmen med træet bruger 70%. Træalgoritmen bruger således noget mere tid med fletningen end array-algoritmen. Testen bekræfter således Katajainen et al.s resultater. Vi bør desuden ligge mærke til, at køretiden i resten af sorteringen er ens for de to funktioner, hvilket også var at forvente, da det er det eneste sted de adskiller sig fra hinanden.

Men for at vende blikket mod det teoretiske perspektiv, så lad os se på, hvordan fordelingen af sammenligninger (S) og flytninger (F) er for de to funktioner:

[5]Testene i dette afsnit er kørt over 10 millioner elementer på en Athlon 1.333 med compileflag: -O3 -pipe -march=athlon.

| | S | Flet / % | F | Flet / % | S i resten |
|---|---|---|---|---|---|
| Array | 321271 | 87,6 | 347458 | 60,8 | 39941 |
| Træ | 232032 | 82,8 | 347458 | 60,8 | 39936 |
| Indbyrdes forhold | 1,38 | | 1,00 | | 1,00 |

Som forventet udfører de to funktioner lige mange flytninger, mens der i arrayet bliver foretaget 1,38 gange flere sammenligninger i træet. Sammenholdt med at denne funktion ifølge forrige test er 1,3 gange hurtigere, bærer det vidne om, at der er andet ved sagen en blot antallet af sammenligninger. Den eneste mulige forklaring er, at vedligeholdelsen af træet udgør en for stor konstant i forhold til det antal sammenligninger der vindes. Det må formodes, at forskellen i køretid vil mindskes, hvis graden af fletningen bliver højere. Da differencen i sammenligninger ville forøges yderligere. Hvis elementerne der sorteres er andet og større end heltal vil arbejdsmængden forbundet med en sammenligning stige, hvorved træ-algoritmen ville have interesse.

Grundet træ-algoritmens teoretiske overlegenhed er det ønskværdigt at prøve at nedbringe omkostningerne forbundet med træets vedligeholdelse. Hvis denne difference kan elimineres, vil træ-algoritmen være den overlegne grundet de færre sammenligninger. Vi skal her fremlægge en mulig forbedring af træstrukturen.

### 8.4.1 Et nyt træ

Idet antallet af sammenligninger er større når der bruges et array, og antallet af flytninger er ens for de to algoritmer, må forskellen ligge i den konkrete implementation. Som det er nævnt i implementationsafsnittet 6.5 sørges der for at undgå unødige sammenligninger når en sektion er tom. Arrayet af pointere har variabel størrelse og antallet af sammenligninger formindskes derved. Denne metode fortsættes indtil der kun er een pointer tilbage, hvorved sammenligning helt undgås. I værste tilfælde vil denne optimering ikke have nogen effekt, men i praksis lader det til at den gør en forskel.

Den samme strategi er der også forsøgt udført i selektionstræet, hvor der ikke sammenlignes på blade der er -1. Problemet er blot at optimeringen er foretaget indenfor samme løkke, hvilket gør at det er nødvendigt at foretage enkelte sammenligninger af værdierne i træets kunder for hver iteration. Derudover kører løkken så længe der er en sekvens der har elementer tilbage. Derved bliver det ikke udnyttet at den sidste sekvens blot kan flyttes til arbejdsområdet. Selektionstræet kunne istedet implementeres ved at splitte løkken op i flere løkker på denne måde:

Udgangspunktet for algoritmen er den samme, træet opbygges med fire blade og rodknuden opdateres til at pege på bladet der peger på det mindste element. I stedet for at lade løkken køre ind til rodknuden indeholder $-1$, skal løkken istedes splittes op i fire løkker.

Figur 7: Ændringen af selektionstræet gennem fletningen

Som det ses på figur 7 bliver hver løkke kørt indtil een af sektionerne bliver tom, herefter ændres tilstand og en ny løkke startes. Tanken er at den "brødreløse" knude bliver flyttet op i træet efter første løkke er stoppet. Derved udgås det at man sammenligner med knuder der er "-1". I bedste tilfælde vil der når rodknuden skal opdateres kun blive lavet een sammenligning. Dette sker når det mindste element bliver taget fra den "brødreløse" knude. For at kunne flytte hvilken som helst knude op i træet, laves den regel at den brødreløse knude altid flyttes til højre knude, og den venstre knude peger på de uforandrede knuder.

På det tidspunkt hvor endnu en sekvens er tom, stoppes løkken igen, og de to resterende "levende" sekvenser bliver flyttet til niveauet under rodknuden. Derved vil der kun foretages een sammenligning når rodknuden skal opdateres. Når een af de to sidste sekvenser bliver tom, stoppes løkken igen og den sidste levende sekvens bliver flyttet direkte over i arbejdsområdet.

## 8.5 Flettesortering med færre flytninger

Med henblik på en optimering af denne algoritme. Er det interresant at se, i hvilke funktioner den bruger størstedelen af sine kræfter. Følgende tabel viser sammenhængen mellem de forskellige funktioner og den værdi, der er knyttet til dem i testværktøjet:

| | |
|---|---|
| mergesort_with_fewer_moves | 0 |
| d_way_merge | 1 |
| d_way_merge_blocks | 2 |
| d_way_sort | 3 |
| make_encoding_zone | 4 |
| make_number | 5 |
| read_number | 6 |
| Initialize_encode_area | 7 |

I bilag B.4 er vist forholdet i henholdsvis flytninger og sammenligner mellem de forskellige funktioner i algoritmen. Det ses at langt det største antal flytninger —næsten 60%, udføres i `d_way_merge_blocks`, mens `read_number` ikke uventet, vores tidligere overvejelser taget i betragtning, står for langt den overvejende del af sammenligningerne. Dette er endnu en eksponering af nævnte misforhold.

Den tredje graf i B.4 præsenterer den mest spektakulære test. På trods af usikkerheden på tidstagningen — og med det forbehold i baghovedet, så viser den, at der bliver brugt lige så meget tid i `read_number` som der bliver brugt `d_way_merge_blocks`. Vi har set af analysen, at antallet af kald til `read_number` kan reduceres med en meget stor faktor, en faktor der burde gøre tiden i `read_number` ubetydelig i forhold til resten af programmet. Sat i forhold til den tredje graf i B.4 vil det betyde en gevaldig forbedring i effektivitet for algortimen.

# 9 Konklusion

Lad os her samle op på de resultater vi har set på i det forgående. I [4] redegjorde Katajainen et al. for, at inplace sorteringen med færre flytninger kun var af teoretisk interesse. Vores test har dog vist at med en ordentlig implementation, kan algoritmen måske få en praktisk interesse. Imidlertid må vi dog konstatere, at den ikke kan konkurrere med de to 4-vejs flettealgoritmer. Som det fremgik af vores test, og som det også fremgik af [3] er den 4-vejssortering, der benytter sig af et array til fletningen, praktisk overlegen i forhold til den, der benytter sig af træfletning. Igen må vi fremhæve, at træsorteringen kan optimeres. Desuden viste analysen, at for meget store datamængder, kan det teoretisk set være rentabelt med et træ af større dybde. Herved ville denne algoritmes fortrin fremfor sorteringen ved hjælp af array blive større. Man skal dog i implementationen være opmærksom på de problemstillinger, der tidligere er blevet fremhævet, da spiltiden forbundet med opdateringen af træet vil vokse med træets størrelse.

Vores forslag til en inplace flettesorteringsalgoritme til tilføjelse til CPH STL må dig være 4-vejs flettesortering med array.

## Litteraturliste

[1] Cormen, Leiserson, Rivest, and Stein. *Introduction to algorithms*. The MIT Press, 2 edition, 2001.

[2] Viliam Geffert, Jyrki Katajainen, and Tomi Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science 237*, pages 159–181, 2000.

[3] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing 3*, pages 27–40, 1996.

[4] Jyrki Katajainen and Tomi A. Pasanen. In-place sorting with fewer moves. *Technical Report 98/22, Department of Computing, University of Copenhagen*, 1998.

[5] Donald E. Knuth. *The art of computer programming vol. 3*. Addison-wesley, 2. udgave edition, 1998.

[6] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. 1997.

[7] J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theoretical Comnputer Science 165*, pages 311–323, 1996.

[8] David R. Musser. Introspective sorting and selection algorithms. 1997.

[9] John D. Valois. Introspective sorting and selection revisited. 1999.

[10] Ingo Wegener. Bottom-up-heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small). 1993.

# A Introsort test

## A.1 Subdivision limit

Figur 8: Subdivision limit test på Athlon-XP. Data for denne test findes i `subdiv_limit`.

Figur 9: Subdivision limit test på Athlon. Data for denne test findes i `subdiv_limit-athlon`.

Figur 10: Subdivision limit test på Pentium 3. Data for denne test findes i `subdiv_limit-p3`.

Figur 11: Subdivision limit test på Pentium 4. Data for denne test findes i `subdiv_limit-p4`.

## A.2 Subdivision limit - 16 vs. 22

Figur 12: Data for denne test findes i `subdiv_test2`.

## A.3 Quicksort test

Figur 13: Data for denne test findes i `quicksort_test`.

## A.4 Introsort på tilfældigt genereret data

Figur 14: Introsort tidsmåling. Data for denne test findes i `introsort_test`.

Figur 15: Introsort sammenligning. Data for denne test findes i `introsort_cm_test`.

Figur 16: Introsort elementflytning. Data for denne test findes i `introsort_cm_test`.

## A.5 Introsort på Median-of-3 killers

Figur 17: Introsort tidsmåling for Median-of-3 killers. Data for denne test findes i `m3k_test`.

Figur 18: Introsort sammenligning for Median-of-3 killers. Data for denne test findes i `m3k_ca_test`.

Figur 19: Introsort elementflytning for Median-of-3 killers. Data for denne test findes i `m3k_ca_test`.

# B Inplace test

## B.1 Flytninger

Figur 20: Test af antallet af flytninger kørt fra 50.000 til 2.000.000.

Figur 21: Test af antallet af flytninger kørt fra 5.000 til 200.000 elementer.

## B.2 Sammenligninger

Figur 22: Test af antallet af sammenligninger fra 50.000 til 1.000.000 elementer.

## B.3 Tidsforbrug

Figur 23: Tidsforbug for de forskellige algoritmer

## B.4 Færre flytninger

Figur 24: Antallet af flytninger pr. funktion i `mergesort_with_fewer_moves`.

Figur 25: Antallet af sammenligninger pr. funktion i `mergesort_with_fewer_moves`.

Figur 26: Udviklingen i antallet af flytninger pr. funktion i forhold til antal elementer.

# C Kildekode

## C.1 Fælles filer

### C.1.1 defines.h

```
#ifndef _defines_h
#define _defines_h

using namespace std;
#define diff_type typename iterator_traits<random_access_iterator>::difference_type
#define value_type typename iterator_traits<random_access_iterator>::value_type

#ifdef COUNT
extern unsigned long comparisons;
extern unsigned long assignments;
#define add_comparisons(x) (comparisons += x)
#define add_assignments(x) (assignments += x)
#else
#define add_comparisons(x) ;
#define add_assignments(x) ;
#endif

#ifdef STOPPER_COUNT
#define add_stopper(x) (stopper_calls += x)
#else
#define add_stopper(x) ;
#endif

#define FUNS 8

#define MERGESORT 0
#define D_WAY_MERGE 1
#define D_WAY_MERGE_BLOCKS 2
#define D_WAY_SORT 3
#define MAKE_ENC 4
#define MAKE_NUM 5
#define READ_NUM 6
#define INIT_ENC 7

#ifdef LOCAL_TIME
#include <sys/time.h>
#include "../sort/testing.h"
extern double function[];
extern struct timeval start_time[];

void start_count(int x) {
  gettimeofday(&start_time[x], 0);
}

void stop_count(int x)  {
  struct timeval stop;
  gettimeofday(&stop, 0);
  double spend = difftime(&start_time[x], &stop);
  function[x] += spend;
}

#else

#ifdef LOCAL_COUNT
#include <time.h>
```

```
extern unsigned long comparisons;
extern unsigned long assignments;
extern unsigned long function_m[];
extern unsigned long function_c[];
extern unsigned long start_move[];
extern unsigned long start_compare[];

void start_count(int x) {
  start_move[x] = assignments;
  start_compare[x] = comparisons;
}

void stop_count(int x)  {
  function_m[x] += assignments - start_move[x];
  function_c[x] += comparisons - start_compare[x];
}

#else
void start_count(int x) {}
void stop_count(int x)  {}
#endif
#endif
#endif //_defines_h
```

### C.1.2 moves.h

```
#ifndef _moves_h
#define _moves_h

#include <iterator>
#include "../common/defines.h"

namespace cphstl {

  /**********************************************************************
   Swap two blocks using a hole for fewer moves. It is stable in l2 */
  template <class random_access_iterator>
    void swap_blocks (random_access_iterator l1,
                      random_access_iterator l2,
                      diff_type  size) {

    value_type buf;
    random_access_iterator last = l1 + size -1;

    add_assignments(1);
    buf = *l1;

    while (l1 != last) {
      add_assignments(2);
      *(l1++) = *l2;
      *(l2++) = *l1;
    }

    add_assignments(2);
    *l1 = *l2;
    *l2 = buf;
  }

  /**********************************************************************
   Do a stable swap of two blocks */
  template <class random_access_iterator>
    void stable_swap_blocks (random_access_iterator l1,
```

```
                                random_access_iterator l2,
                                diff_type size) {
    while (size--){
      add_assignments(3);
      iter_swap(l1++, l2++);
    }
  }
}

#endif //_moves_h
```

### C.1.3 logarithms.h

```
#ifndef _logarithms_h
#define _logarithms_h

inline
int log_2 (int x) {
  int result = 0;
  x >>= 1;
  while (x > 0) {
    x >>= 1;
    result++;
  }
  return result;
}

inline
int log_2_ceil (int x) {
  int result = 0;
  bool up = false;
  if (x % 2 == 1) up = true;
  x >>= 1;
  while (x > 0) {
    if (x % 2 == 1) up = true;
    x >>= 1;
    result++;
  }
  if (up)
    return result + 1;
  else
    return result;
}

double log4 (double x) {
  double i = 1, j = 0;
  while (i < x) {
    i *= 4;
    j += 1;
  }
  return (j);
}

#endif //_logarithms_h
```

### C.1.4 testing.h

```
#ifndef _testing_h
#define _testing_h

#include <vector>
#include <iostream>
```

```
#include <iterator>

using namespace std;

double difftime (struct timeval * start , struct timeval * stop){
  double result = stop->tv_sec - start->tv_sec;
  result += (((double)(stop->tv_usec - start->tv_usec)) / 1000000);
  return result;
}

template <class random_access_iterator>
bool issorted (random_access_iterator first , random_access_iterator last ) {
  while ( first != last - 1) {
    if (* first > *( first +1)) return false;
    first ++;
  }
  return true;
}

template <class random_access_iterator>
void make_random (random_access_iterator first,random_access_iterator last ) {
  while ( first != last ){
    *( first ++) = rand() % 100000;
  }
}

void fill_random (vector<int>& v, int nr) {
  for (int i = 0; i < nr; i++) {
    v.push_back(rand() % 100000);
  }
}

template <class forward_access_iterator>
void print ( forward_access_iterator first , forward_access_iterator last ) {
  while ( first != last ){
    cout << *first << " ";
    first ++;
  }
  cout << endl;
}

void median_of_3_killer_sequence (vector<int>& v, int size) {
  int k = size / 2;
  for (int i = 1; i <= k; i++) {
    if (i % 2 == 1)
      v.push_back(i);
    else
      v.push_back(k + i - 1);
  }
  for (int i = 2; i <= 2*k; i += 2)
    v.push_back(i);
}

void sorted_sequence (vector<int>& v, int size) {
  for (int i = 1; i <= size; i++)
    v.push_back(i);
}

void organpipe_sequence (vector<int>& v, int size) {
  int k = size / 2;
  for (int i = 1; i <= k; i++)
    v.push_back(i);
```

```
  for (int i = k; i >= 1; i--)
    v.push_back(i);
}

#endif
```

## C.2 Introspective sorting

### C.2.1 sorting.h

```
#ifndef _cphstl_sorting_h
#define _cphstl_sorting_h

#include <vector>
#include <iterator>

using namespace std;

// For testing purposes this can be made a variable.
#ifdef SUBDIV_TEST
extern int _cphstl_subdiv_limit;
#else
#define _cphstl_subdiv_limit 22
#endif

#define diff_type typename iterator_traits<random_access_iterator>::difference_type
#define value_type typename iterator_traits<random_access_iterator>::value_type

#ifdef COUNT
extern unsigned long comparisons;
extern unsigned long assignments;
#define add_comparisons(x) (comparisons += x)
#define add_assignments(x) (assignments += x)
#else
#define add_comparisons(x) ;
#define add_assignments(x) ;
#endif

#ifdef STOPPER_COUNT
extern unsigned long stopper_calls;
#define add_stopper(x) (stopper_calls += x)
#else
#define add_stopper(x) ;
#endif

namespace cphstl {
  // Partitions the collection around the value in pivot.
  // Returns the position of the cut which is the first position in
  // the second part of the collection (ie. the 'last' iterator for the
  // first part and the 'first' iterator for the second part)
  template <class random_access_iterator>
  random_access_iterator partition (random_access_iterator first,
                                    random_access_iterator last,
                                    value_type pivot) {
    last--;
    while (true) {
      while (*first < pivot) {
        first++;
        add_comparisons(1);
      }
      add_comparisons(1);
      while (*last > pivot) {
```

```
        last--;
        add_comparisons(1);
      }
      add_comparisons(1);
      if (!( first < last ))
        return first;
      iter_swap ( first , last );
      add_assignments(3);
      first ++;
      last--;
    }
  }

  // Sorts a collection using insertionsort .
  // This is used for the final sorting in quicksort and introsort .
  template <class random_access_iterator>
  void insertion_sort (random_access_iterator first ,
                       random_access_iterator last ) {
    if ( first < last ) {
      random_access_iterator x, y;
      value_type tmp;
      x = first + 1;
      while (x != last) {
        tmp = *x;
        add_assignments(1);
        y = x - 1;
        while (y >= first && tmp < *y) {
          add_comparisons(1);
          *(y + 1) = *y;
          add_assignments(1);
          y--;
        }
        add_comparisons(1);
        *(y + 1) = tmp;
        add_assignments(1);
        x++;
      }
    }
  }

  // Finds the median of the three elements.
  // Returns this medians value.
  template <class value>
  value median_of_3 (value a, value b, value c) {
    if (a < b) {
      if (b < c){
        add_comparisons(2);
        return b; // a < b < c
      }
      else {
        add_comparisons(3);
        if (a < c) // a < c < b
          return c;
        else
          return a; // c < a < b
      }
    }
    else {
      if (a < c) { // b < a < c
        add_comparisons(2);
        return a;
      }
```

```
    else {
      add_comparisons(3);
      if (b > c)
        return b; // c < b < a
      else
        return c; // b < c < a
    }
  }
}

// Sorts the collection using median-of-three quicksort.
// This version uses insertion sort every time a collection of
// size < SUBDIV_LIMIT is encountered. This should give better
// cache performance.
template <class random_access_iterator>
void qsort_m3_many (random_access_iterator first,
                    random_access_iterator last) {
  while (last - first > _cphstl_subdiv_limit){
    random_access_iterator cut =
      cphstl::partition(first, last,
                        median_of_3 (*first, *(last - 1),
                                     *(first + (last - first) / 2)));
    qsort_m3_many (cut, last);
    last = cut;
  }
  insertion_sort (first, last);
}

// Auxiliary function for qsort_m3_one.
template <class random_access_iterator>
void qsort_m3_one_loop (random_access_iterator first,
                        random_access_iterator last) {
  while (last - first > _cphstl_subdiv_limit){
    random_access_iterator cut =
      cphstl::partition(first, last,
                        median_of_3 (*first, *(last - 1),
                                     *(first + (last - first) / 2)));
    qsort_m3_one_loop (cut, last);
    last = cut;
  }
}

// Sorts the collection using median-of-three quicksort.
// This version uses insertion sort only in one final pass.
template <class random_access_iterator>
void qsort_m3_one (random_access_iterator first,
                   random_access_iterator last) {
  if (first < last) {
    qsort_m3_one_loop (first, last);
    insertion_sort (first, last);
  }
}

// Reheap function specially for heapsort. Makes sure the max heap property is
// held in the collection.
template <class random_access_iterator>
void reheap (random_access_iterator first,
             diff_type parent,
             diff_type length) {
  diff_type secondchild = 2 * parent + 2;
  while (secondchild < length) {
    add_comparisons(1);
```

```
      if (*( first + parent) > *(first + secondchild - 1)){
        add_comparisons(1);
        if (*( first + parent) > *(first + secondchild)){
          // MAX is parent. Everything is fine!
          break;
        }
      }
      else {
        add_comparisons(1);
        if (*( first + secondchild - 1) > *(first + secondchild)){
          // Max is secondchild - 1.
          secondchild--;
        }
      }

      iter_swap ( first + parent, first + secondchild);
      add_assignments(3);

      parent = secondchild;
      secondchild = 2 * parent + 2;
    }
    // Check if there is one child left .
    if (secondchild == length){
      add_comparisons(1);
      if (*( first + parent) < *(first + secondchild - 1)){
        iter_swap ( first + parent, first + secondchild - 1);
        add_assignments(3);
      }
    }
}

template <class random_access_iterator>
void heapsort (random_access_iterator first , random_access_iterator last ) {
    diff_type length = last - first ;
    // Create the MAX-Heap.
    for ( diff_type i = (length - 2) / 2; i >= 0; i--) {
      reheap ( first , i , length);
    }

    // Do the sorting magic.
    for ( diff_type i = length - 1; i > 0; i--) {
      iter_swap ( first , first + i);
      add_assignments(3);
      reheap ( first , 0, i);
    }
}

// Log2 function for diff_types ( integers ).
template <class value>
inline
value log_2 (value x) {
    value result = 0;
    x >>= 1;
    while (x > 0) {
      x >>= 1;
      result++;
    }
    return result;
}

// This function makes sure the MAX-Heap property is held in the collection.
// This was inspired from the article 'BOTTOM-UP-HEAPSORT, a new variant...'
```

```
// by Ingo Wegener.
template <class random_access_iterator>
void bottom_up_reheap (random_access_iterator first,
                       diff_type parent, diff_type length) {
  diff_type specialleaf = parent;
  value_type tmp;

  // Find the special leaf.
  while ((specialleaf << 1) + 1 < length - 1) {
    add_comparisons(1);
    if (*(first + (specialleaf << 1) + 1) > *(first + (specialleaf << 1) + 2))
      specialleaf = (specialleaf << 1) + 1;
    else
      specialleaf = (specialleaf << 1) + 2;
  }
  if ((specialleaf << 1) + 1 == length - 1)
    specialleaf = length - 1;

  // Do the bottom-up-search.
  while (*(first + parent) > *(first + specialleaf)){
    add_comparisons(1);
    specialleaf = (specialleaf - 1) >> 1;
  }
  add_comparisons(1);

  // The variable 'specialleaf' should now point to the place where the
  // value in 'parent' will be placed.
  tmp = *(first + parent);
  add_assignments(1);
  diff_type k = log_2 ((specialleaf + 1)/(parent + 1));
  while (k > 0) {
    *(first + (((specialleaf + 1) >> k) - 1)) =
      *(first + (((specialleaf + 1) >> (k - 1)) - 1));
    add_assignments(1);
    k--;
  }
  *(first + specialleaf) = tmp;
  add_assignments(1);
}

// Bottom-up-heapsort.
// This was inspired from the article 'BOTTOM-UP-HEAPSORT, a new variant...'
// by Ingo Wegener.
template <class random_access_iterator>
void bottom_up_heapsort (random_access_iterator first,
                         random_access_iterator last) {
  diff_type length = last - first;
  // Create the MAX-Heap.
  for (diff_type i = (length - 2) / 2; i >= 0; i--) {
    bottom_up_reheap (first, i, length);
  }

  // Do the sorting magic.
  for (diff_type i = length - 1; i > 0; i--) {
    iter_swap (first, first + i);
    add_assignments(3);
    bottom_up_reheap (first, 0, i);
  }
}

// This function makes sure the MAX-Heap property is held in the collection.
// This was inspired in part by the current STL implementation and in part by
```

```
// some code from Jyrki's homepage.
template <class random_access_iterator>
void bottom_up_reheap2 (random_access_iterator first, diff_type parent,
                        diff_type length) {
  diff_type child = (parent << 1) + 1;
  value_type tmp = *(first + parent);

  // Run down to the special leaf, swapping elements as we go.
  diff_type hole = parent;
  while (child < length - 1) {
    add_comparisons(1);
    if (*(first + child) < *(first + child + 1))
      child++;
    *(first + hole) = *(first + child);
    add_assignments(1);
    hole = child;
    child = (hole << 1) + 1;
  }
  if (child == length - 1) {
    *(first + hole) = *(first + child);
    add_assignments(1);
  }
  else
    child = hole;

  // Run back up swapping elements until we hit the right place for
  // the element at 'parent'.
  while (child > parent) {
    add_comparisons(1);
    if (tmp > *(first + (((child + 1) >> 1) - 1))){
      *(first + child) = *(first + (((child + 1) >> 1) - 1));
      add_assignments(1);
      child = ((child + 1) >> 1) - 1;
    }
    else
      break;
  }
  *(first + child) = tmp;
  add_assignments(1);
}

// Another bottom-up-heapsort.
// This was inspired in part by the current STL implementation and in part by
// some code from Jyrki's homepage.
template <class random_access_iterator>
void bottom_up_heapsort2 (random_access_iterator first,
                          random_access_iterator last) {
  diff_type length = last - first;
  // Create the MAX-Heap.
  for (diff_type i = (length - 2) / 2; i >= 0; i--) {
    bottom_up_reheap2 (first, i, length);
  }

  // Do the sorting magic.
  for (diff_type i = length - 1; i > 0; i--) {
    iter_swap (first, first + i);
    add_assignments(3);
    bottom_up_reheap2 (first, 0, i);
  }
}
```

```
// Auxiliary function for introsort_one.
template <class random_access_iterator>
void introsort_one_loop (random_access_iterator first,
                         random_access_iterator last, int depthlimit) {
  while (last - first > _cphstl_subdiv_limit){
    if (depthlimit == 0){
      add_stopper(1);
      bottom_up_heapsort2(first, last);
      return;
    }
    depthlimit--;
    random_access_iterator cut =
      cphstl::partition(first, last,
                median_of_3 (*first, *(last - 1),
                             *(first + (last - first) / 2)));
    introsort_one_loop (cut, last, depthlimit);
    last = cut;
  }
}

// Introsort - as described by David Musser.
// Inspired by the current STL implementation.
template <class random_access_iterator>
void introsort_one (random_access_iterator first,
                    random_access_iterator last) {
  if (first < last) {
    introsort_one_loop (first, last, log_2((last - first) << 1));
    insertion_sort (first, last);
  }
}

// Inspired by the current STL implementation.
// This one uses many calls of insertion_sort instead of just one.
template <class random_access_iterator>
void introsort_many_loop (random_access_iterator first,
                          random_access_iterator last, int depthlimit) {
  while (last - first > _cphstl_subdiv_limit){
    if (depthlimit == 0){
      add_stopper(1);
      bottom_up_heapsort2(first, last);
      return;
    }
    depthlimit--;
    random_access_iterator cut =
      cphstl::partition(first, last,
                median_of_3 (*first, *(last - 1),
                             *(first + (last - first) / 2)));
    introsort_many_loop (cut, last, depthlimit);
    last = cut;
  }
  insertion_sort (first, last);
}

// Introsort - as described by David Musser.
// Inspired by the current STL implementation.
template <class random_access_iterator>
void introsort_many (random_access_iterator first,
                     random_access_iterator last) {
  if (first < last) {
    introsort_many_loop (first, last, log_2((last - first) << 1));
  }
}
```

```
#define _B 4
#define _COEF 2
#define _DIV 2

  // Introsort revisited. Implementation of the algorithm described by John D.
  // Valois.
  // This implementation uses fine grained introspection and remedial
  // randomization. It still used bottom-up-heapsort as a stopper algorithm.
  // This code was inspired by (taken from) the article by John D. Valois.
  // This version of introsort_rev uses one final pass of insertionsort.
  template <class random_access_iterator>
  void introsort_rev_one_loop (random_access_iterator first,
                               random_access_iterator last) {
    int counter = _B + _COEF + 1;
    diff_type threshold = (last - first) / _DIV;
    diff_type limit = (last - first);

    while (last - first > _cphstl_subdiv_limit) {
      if (--counter == 0) {
        if (last - first >= threshold) {
          if (limit /= 2) {
            iter_swap (first, first + rand() % (last - first));
            iter_swap (last - 1, first + 1 + rand() % (last - first - 1));
            iter_swap (first + (last - first) / 2,
                       first + 1 + rand() % (last - first - 2));
            add_assignments(9);
            counter = _B + _COEF;
            threshold = (last - first) / _DIV;
          }
          else {
            add_stopper(1);
            bottom_up_heapsort2 (first, last);
            break;
          }
        }
        else {
          counter = _COEF;
          threshold /= _DIV;
        }
      }

      random_access_iterator cut =
        cphstl::partition (first, last,
                  median_of_3 (*first,
                               *(first + (last - first) / 2),
                               *(last - 1)));
      if (cut - first >= last - cut) {
        introsort_rev_one_loop (cut, last);
        last = cut;
      }
      else {
        introsort_rev_one_loop (first, cut);
        first = cut;
      }
    }
  }

  template <class random_access_iterator>
  void introsort_rev_one (random_access_iterator first,
                          random_access_iterator last) {
    if (first < last) {
```

```
      introsort_rev_one_loop ( first , last );
      insertion_sort ( first , last );
    }
  }

  // This version of introsort_rev uses many calls to insertionsort .
  // It also uses fine grained introspection and remedial randomization.
  template <class random_access_iterator>
  void introsort_rev_many (random_access_iterator first ,
                           random_access_iterator last ) {
    int counter = _B + _COEF + 1;
    diff_type threshold = (last - first ) / _DIV;
    diff_type limit = (last - first );

    while (last - first > _cphstl_subdiv_limit ) {
      if (--counter == 0) {
        if ( last - first >= threshold) {
          if ( limit /= 2) {
            iter_swap ( first , first + rand() % (last - first ));
            iter_swap ( last - 1, first + 1 + rand() % (last - first - 1));
            iter_swap ( first + (last - first ) / 2,
                        first + 1 + rand() % (last - first - 2));
            add_assignments(9);
            counter = _B + _COEF;
            threshold = (last - first ) / _DIV;
          }
          else {
            add_stopper(1);
            bottom_up_heapsort2 (first , last );
            break;
          }
        }
        else {
          counter = _COEF;
          threshold /= _DIV;
        }
      }

      random_access_iterator cut =
        cphstl :: partition ( first , last ,
                    median_of_3 (* first ,
                                          *( first + (last - first ) / 2),
                                          *( last - 1)));
      if (cut - first >= last - cut) {
        introsort_rev_many (cut , last );
        last = cut;
      }
      else {
        introsort_rev_many ( first , cut);
        first = cut;
      }
    }
    insertion_sort ( first , last );
  }

#define _B_NR 4
#define _COEF_NR 2
#define _DIV_NR 2

  // This version of introsort_rev uses many calls to insertionsort .
  // This function does not use remedial randomization.
  template <class random_access_iterator>
```

```
  void introsort_rev_many_norandom (random_access_iterator first,
                                    random_access_iterator last ) {
    int counter = _B_NR + _COEF_NR + 1;
    diff_type threshold = (last - first ) / _DIV_NR;

    while (last - first > _cphstl_subdiv_limit ) {
      if (--counter == 0) {
        if ( last - first >= threshold) {
          add_stopper(1);
          bottom_up_heapsort2 (first , last );
          break;
        }
        else {
          counter = _COEF_NR;
          threshold /= _DIV_NR;
        }
      }

      random_access_iterator cut =
        cphstl :: partition ( first , last ,
                              median_of_3 (* first ,
                                           *( first + (last - first ) / 2),
                                           *(last - 1)));
      if (cut - first >= last - cut) {
        introsort_rev_many_norandom (cut, last);
        last = cut;
      }
      else {
        introsort_rev_many_norandom (first, cut);
        first = cut;
      }
    }
    insertion_sort ( first , last );
  }
};

#endif // _cphstl_sorting_h
```

## C.3 Practical in-place mergesort

### C.3.1 common.h

```
#ifndef _common_h
#define _common_h
#include <vector>
#include <iterator>
#include <iostream>
#include "../common/defines.h"
#include "../common/logarithms.h"
#include "../sort/sorting.h"

namespace cphstl{
  /*
  template <class random_access_iterator>
    void insertion_sort (random_access_iterator first ,
                         random_access_iterator one_past){

    random_access_iterator i , j;
    value_type tmp;
    for (j = first + 1; j < one_past; j++) {
      add_assignments(1);
      tmp = *j;
```

```
    i = j - 1;
    while (i >= first && (*i > tmp)) {
      add_comparisons(1);
      add_assignments(1);
      *(i+1) = *i;
      --i;
    }
    add_comparisons(1);
    add_assignments(1);
    *(i+1) = tmp;
  }
}
*/
/*********************************************************************/
template <class random_access_iterator>
  void reverse2(random_access_iterator first , random_access_iterator last){

   diff_type t;

  for(t = 0; t < ((last + 1 - first) >> 1); ++t){
    add_assignments(3);
    iter_swap( first + t, last - t );
  }
}

/*********************************************************************/
template <class random_access_iterator>
  void move_right(random_access_iterator first ,
                  random_access_iterator last ,
                   diff_type k){
  if(k > 0){
    reverse2( first , last );
    reverse2(last + 1, last + k);
    reverse2( first , last + k);
  }
}

/*********************************************************************
The function swaps two bloks of elements. It uses a hole to avoid
unnecessary moves.
*/
template <class random_access_iterator>
  void change_places(random_access_iterator first ,
                     random_access_iterator last ,
                     random_access_iterator buf){

  value_type tmp;

  add_assignments(1);
  tmp = *buf;

  while(first < last){
    add_assignments(2);
    *buf++ = *first;
    * first ++ = *buf;
  }
  add_assignments(2);
  *buf = * first ;
  * first = tmp;
}

/*********************************************************************
```

```
This function merge two subarrays together and puts the sorted elements
in a buffer. The buffer is placed between the two sorting areas and
its size is equal to the smallest of the sorting areas.  */
template <class random_access_iterator>
  void straight_merge(random_access_iterator xl,
                      random_access_iterator xr,
                      random_access_iterator yl,
                      random_access_iterator yr,
                      random_access_iterator buf){

  value_type temp;
  add_assignments(1);

  //The hole is created by storing the first element in the buffer in
  //tempoary variable. By that we get rid of swap-operations.
  temp = *buf;

  //The elements are merged until one of the arrays only contain one
  //element that need to be merged.
  while (xl < xr && yl < yr) {
    add_comparisons(1);
    add_assignments(2);
    if (*xl <= *yl) {
      *(buf++) = *xl;
      *(xl++) = *buf;
    }
    else {
      *(buf++) = *yl;
      *(yl++) = *buf;
    }
  }

  //The merge is made until one of the sorrting blocks is merged.
  while (xl <= xr && yl <= yr) {
    add_comparisons(1);
    add_assignments(2);
    if (*xl <= *yl) {
      *(buf++) = *xl;
      *xl = ((xl < xr) ? *buf : temp);
      ++xl;
    }
    else {
      *(buf++) = *yl;
      *yl = ((yl < yr) ? *buf : temp);
      ++yl;
    }
  }

  //If the buf equal to xl or yl the merging is over because we know
  //that these last elements are larger than the al the merged elements.
  if (buf == xl || buf == yl)
    return;
  add_assignments(1);

  // The hole is created again, and because we know that these elements are
  // larger than the merged elements they are put at the end.
  temp = *buf;
  if (yl > yr) {
    while (xl < xr) {
      add_assignments(2);
      *(buf++) = *xl;
      *(xl++) = *buf;
```

```
      }
      add_assignments(2);
      *buf = *xl;
      *xl = temp;
    }
    else {
      while (yl < yr) {
        add_assignments(2);
        *(buf++) = *yl;
        *(yl++) = *buf;
      }
      add_assignments(2);
      *buf = *yl;
      *yl = temp;
    }
  }

  /*********************************************************************
   The procedure search the highest index k, yl <= k <= yr, by
   binarysearch for which the relation *( left + k) <= z is true. */

  template <class random_access_iterator>
    random_access_iterator high_binary_search(random_access_iterator yl,
                                              random_access_iterator yr,
                                              value_type z){

    random_access_iterator mid, low = yl, high = yr;
    if (yr < yl)
      return (yl - 1);
    while ((high - low) >= 1) {
      mid = low + ((high - low + 1) >> 1);
      add_comparisons(1);
      if (*mid <= z)
        low = mid;
      else
        high = mid - 1;
    }
    add_comparisons(1);
    if (*low <= z)
      return (low);
    else
      return (low - 1);
  }

  /*********************************************************************

*/
  template <class random_access_iterator>
    void forward_block_merge(random_access_iterator xl,
                             random_access_iterator xr,
                             random_access_iterator yl,
                             random_access_iterator yr){

    random_access_iterator h;
    while (xr - xl >= 0) {
      h = high_binary_search(xr + 1, yr, *xl);
      move_right(xl, xr, h-xr);
      xl = h + (xl-xr) + 1;
      xr = h;
    }
  }
```

```
/*****************************************************************
 The procedure merges subarrays *(xl..xr) and *(yl..yr) into the
 subarray *buf by binary merge. The elements in the result area are
 swapped with the merged elements. */
template <class random_access_iterator>
  void binary_merge(random_access_iterator xl,
                    random_access_iterator xr,
                    random_access_iterator yl,
                    random_access_iterator yr,
                    random_access_iterator buf){

  diff_type m, n;
  value_type temp;
  random_access_iterator i;
  m = xr - xl + 1;
  n = yr - yl + 1;
  add_assignments(1);

  // The hole is created by storing the first element *buf of
  // result area. By that we get rid of swap-operations.
  temp = *buf;
  while (m > 0 && n > 0) {
    if (m <= n) {

      //Finds a element in y to compare with first element in x. The
      //probability says that each element in x coresponds to n/m
      //elements in y.
      i = yl + n / m - 1;
      add_comparisons(1);

      //If the element in x is greater than *i the elements in y smaller
      //than *i is moved to the buffer.
      if (*xl >= *i) {
        while (yl < i) {
          add_assignments(2);
          *buf++ = *yl;
          *yl++ = *buf;
        }
        add_assignments(2);
        *buf++ = *yl;
        *yl = ((yl < yr) ? *buf : temp);
        ++yl;
        n = yr - yl + 1;
      }
      else {

        //The place fore *xl in the yl + n / m-1 elemnts is found
        //and the elements in y smaller than *xl is moved to the buffer.
        if ((yl - 1) != (i = high_binary_search(yl, i - 1, *xl))) {
          while (yl < i) {
            add_assignments(2);
            *buf++ = *yl;
            *yl++ = *buf;
          }
          add_assignments(2);
          *buf++ = *yl;
          *yl = ((yl < yr) ? *buf : temp);
          ++yl;
          n = yr - yl + 1;
        }
        add_assignments(2);
```

```
        //Know *xl is moved to the buffer.
        *buf++ = *xl;
        *xl = ((xl < xr) ? *buf : temp);
        ++xl;
        --m;
      }
    }
    else {
      //If x is larger than y the opposite procedure is made.
      i = xl + m / n - 1;
      add_comparisons(1);
      if (*yl >= *i) {
        while (xl < i) {
          add_assignments(2);
          *buf++ = *xl;
          *xl++ = *buf;
        }
        add_assignments(2);
        *buf++ = *xl;
        *xl = ((xl < xr) ? *buf : temp);
        ++xl;
        m = xr - xl + 1;
      }
      else {
        if ((xl - 1) != (i = high_binary_search(xl, i - 1, *yl))) {
          while (xl < i) {
            add_assignments(2);
            *buf++ = *xl;
            *xl++ = *buf;
          }
          add_assignments(2);
          *buf++ = *xl;
          *xl = ((xl < xr) ? *buf : temp);
          ++xl;
          m = xr - xl + 1;
        }
        add_assignments(2);
        *buf++ = *yl;
        *yl = ((yl < yr) ? *buf : temp);
        ++yl;
        --n;
      }
    }
  }
  if (buf == xl || buf == yl)
    return;
  add_assignments(1);
  // The hole is created again
  temp = *buf;

  //There is only unmerged elements in one of the blocks, and
  //they are now moved to the buffer.
  if (n <= 0) {
    while (xl < xr) {
      add_assignments(2);
      *buf++ = *xl;
      *xl++ = *buf;
    }
    add_assignments(2);
    *buf = *xl;
    *xl = temp;
  }
```

```
    else {
      while (yl < yr) {
        add_assignments(2);
        *buf++ = *yl;
        *yl++ = *buf;
      }
      add_assignments(2);
      *buf = *yl;
      *yl = temp;
    }
}

/*********************************************************************
 Non-recursive merge sort of the subarray *(left .. right ) by using
 locations *(buf .. left1 -1) or *(right+1..buf) as the extra space. If
 to_right is 0 then the buffer-area is in *(buf .. left -1), else if
 to_right is 1 then it is in *(buf .. buf+right-left ). The assumption is
 that |*( buffer-area)| = |*( left .. right )|. */

template <class random_access_iterator>
  void two_way_msort(random_access_iterator left,
                     random_access_iterator right ,
                     random_access_iterator buf,
                      diff_type  to_right ){

   diff_type runs, nelements, size ;
   random_access_iterator cursor , left1 , right1 , left2 , right2 ;

   nelements = right - left + 1;

   //The size of a sequence is 1 at the start of the four-way-merge.
   size = 1;

   //The sorting is done untill there is only one sequence left with the
   //length of the array.
   while (size < nelements) {

     //The number of sequences is set.
     runs = nelements / size;

     //If there is a remainder this must also be merged.
     if (nelements % size)
       ++runs;
     cursor = left ;

     //The 2-way-merge is made.
     for (int i = 1; i <= (runs >> 1); ++i) {
       left1 = cursor;
       left2 = left1 + size ;
       right1 = left2 - 1;
       right2 = (((right1 + size) > right ) ? right : right1 + size );
       straight_merge ( left1 , right1 , left2 , right2 , buf);
       buf += (right2 - left1 + 1);
       cursor += (right2 - left1 + 1);
     }

     //If there is a remainder after the merge then the last sequences
     //must be taking care of this way.
     if (runs % 2 == 1) {
       change_places(cursor , right , buf);
     }
     buf = left ;
```

```
      if ( to_right ) {
        left += nelements;
        right += nelements;
        to_right = 0;
      } else {
        left -= nelements;
        right -= nelements;
        to_right = 1;
      }
      size = size << 1;
    }
  }

template <class random_access_iterator>
  void simple_tkp(random_access_iterator left,
                  random_access_iterator right){

  random_access_iterator cut, start, buf, end;
  diff_type nelements, half, diff;
  int nrounds_is_odd;

  cut = right;
  nelements = cut - left + 1;
  half = (nelements >> 1);
  if (nelements == 1)
    return;

  //If there is a remaider then the rounds is odd.
  nrounds_is_odd = (int) ceil(log2((double) half)) % 2;

  //If nrounds_is_odd is 1 the right zone is merged, else
  //the left zone.
  if (nrounds_is_odd) {
    buf = cut - half + 1;
    end = buf - 1;
    start = buf - half;
  }
  else {
    end = cut;
    start = end - half + 1;
    buf = start - half;
  }
  two_way_msort(start, end, buf, nrounds_is_odd);
  cut = cut - half;
  nelements = cut - left + 1;
  half = (nelements >> 1);

  // Handling of the first n - O(sqrt(n)) elements
  diff = right - left + 1;
  while (half >= (int) sqrt((double) diff)) {
    nrounds_is_odd = (int) ceil(log2((double) half)) % 2;
    if (nrounds_is_odd) {
      buf = left;
      start = buf + half;
      end = start + half - 1;
      nrounds_is_odd = 0;
    }
    else {
      start = left;
      end = start + half - 1;
      buf = end + 1;
      nrounds_is_odd = 1;
```

```
    }
    two_way_msort(start, end, buf, nrounds_is_odd);
    binary_merge(left, left + half - 1, cut + 1, right, cut - half + 1);
    cut = cut - half;
    nelements = cut - left + 1;
    half = (nelements >> 1);
  }

  // Handling of the first O(sqrt(n)) elements
  insertion_sort (left, left + nelements);
  forward_block_merge(left, left + nelements - 1,
                      left + nelements, right);
}

/*****************************************************************
      The procedure searches the smallest elements of two blocks
      [left .. cut] and [cut+1..right]. */

template <class random_access_iterator>
  void find_smallest_elements(random_access_iterator left,
                              random_access_iterator cut,
                              random_access_iterator right,
                              random_access_iterator *last1,
                              random_access_iterator *last2){

  diff_type bufsize, leftblocksize;

  bufsize = 0;
  leftblocksize = cut - left + 1;
  add_assignments(2);
  *last1 = left - 1;
  *last2 = cut;
  while (*last2 < right && bufsize < leftblocksize) {
    add_comparisons(1);
    if (*(*last1 + 1) <= *(*last2 + 1)) {
      add_assignments(1);
      *last1 = *last1 + 1;
      --leftblocksize;
    }
    else {
      add_assignments(1);
      *last2 = *last2 + 1;
    }
    ++bufsize;
  }
}

/*****************************************************************
 Sort the area [left .. right] with minimum space. First the second half
 is sorted by using first half as the auxiliary space. Then the first
 quarter is sorted by using the second quarter as the auxiliary space,
 and merged with the sorted second half, into locations starting from
 the second quarter. Now the first quarter is handled similarly,
 leaving 1/8 of the array unsorted, and so on. The process stops after
 log N phases, leaving the first element unsorted. It is finally
 inserted to the correct position. */

template <class random_access_iterator>

  void tkp(random_access_iterator left, random_access_iterator right){

  diff_type nelements, nrounds_is_odd, half;
```

```
random_access_iterator buf, end, start , cut;
cut = right;
nelements = cut - left + 1;
half = (nelements >> 1);
if (nelements == 1)
  return;

//If there is a remaider then the rounds is odd.
nrounds_is_odd = (int) ceil(log2((double) half)) % 2;

//If nrounds_is_odd is 1 the right zone is merged, else
//the left zone.
if (nrounds_is_odd) {
  buf = cut - half + 1;
  end = buf - 1;
  start = buf - half;
}
else {
  end = cut;
  start = end - half + 1;
  buf = start - half;
}
two_way_msort(start, end, buf, nrounds_is_odd);

//The pointers is moved and the merge is continued on the
//buffer area
cut = cut - half;
nelements = cut - left + 1;
half = (nelements >> 1);

//The 2-way-merge is continued until the part to sort + the buffer is smaller
//than n / log2 n elements. The area just sorted in every loop is merged
//with the previous sorted area.
while (nelements >= (right - left + 1) /
       log2((double)(right - left + 1))) {

  nrounds_is_odd = (int) ceil(log2((double) half)) % 2;
  if (nrounds_is_odd) {
    buf = left;
    start = buf + half;
    end = start + half - 1;
    nrounds_is_odd = 0;
  }
  else {
    start = left;
    end = start + half - 1;
    buf = end + 1;
    nrounds_is_odd = 1;
  }
  two_way_msort(start, end, buf, nrounds_is_odd);
  binary_merge(left, left + half - 1, cut + 1,
               right, cut - half +1);
  cut = cut - half;
  nelements = cut - left + 1;
  half = (nelements >> 1);
}
{
  random_access_iterator last1, last2;

  // Handling of the first O(n / log n) elements
  simple_tkp(left, cut);
```

```
      // Searching of the smallest O(n / log n) elements of two
      // ordererd block, the first is left .. cut and the other is
      // cut+1 .. right
      find_smallest_elements( left , cut, right, &last1, &last2 );

       // The smallest elements are in left .. last1 and cut+1..last2
       // If the smallest elements are in front already, the job is
       // done.
       if ( last1 != cut)
        if ( last2 == right)
          move_right( left , cut, right - cut);
        else {
          straight_merge( last1 + 1, cut, left + 1, left , left );
          binary_merge( left , left + (cut - last1 - 1), last2 + 1,
                        right , ( last2 - cut) + last1 + 1);
          simple_tkp( left , left + ((last1 - left ) + 1 + (last2 - cut )));
        }
    }
  }
}
#endif //_common_h
```

### C.3.2 inplace_merge_sort_a.h

```
#ifndef _inplace_merge_sort_a_h
#define _inplace_merge_sort_a_h

#include <vector>
#include <iterator>
#include "../common/defines.h"
#include "../common/logarithms.h"
#include <iostream>
#include "common.h"

//using namespace std;
namespace cphstl {
  template <class random_access_iterator>
    void i_four_way_merge(random_access_iterator buf, int nqueues,
                          random_access_iterator left1 ,
                          random_access_iterator right1 ,
                          random_access_iterator left2 ,
                          random_access_iterator right2 ,
                          random_access_iterator left3 ,
                          random_access_iterator right3 ,
                          random_access_iterator left4 ,
                          random_access_iterator right4 ) {

    start_count (0);

    value_type temp;

    // The hole is created by storing the first element of the result area.
    // This is done to get rid of one move operation when two elements are
    // being switched.
    temp = *buf;
    add_assignments(1);

    //This loop is continued until one of the sequences are empty.
    while (nqueues == 4) {
      add_comparisons(3);
      add_assignments(2);
```

```
//The smallest element is found, and it is swaped to the work area
if (*left1 <= *left2) {
  if (*left3 <= *left4) {
    if (*left1 <= *left3) {
      *(buf++) = *left1;
      *(left1++) = *buf;
      if (left1 > right1) {
        --nqueues;
        left1 = left4;
        right1 = right4;
      }
    }
    else {
      *(buf++) = *left3;
      *(left3++) = *buf;
      if (left3 > right3) {
        --nqueues;
        left3 = left4;
        right3 = right4;
      }
    }
  }
  else {
    if (*left1 <= *left4) {
      *(buf++) = *left1;
      *(left1++) = *buf;
      if (left1 > right1) {
        --nqueues;
        left1 = left4;
        right1 = right4;
      }
    }
    else {
      *(buf++) = *left4;
      *(left4++) = *buf;
      if (left4 > right4)
        --nqueues;
    }
  }
}
else {
  if (*left3 <= *left4) {
    if (*left2 <= *left3) {
      *(buf++) = *left2;
      *(left2++) = *buf;
      if (left2 > right2) {
        --nqueues;
        left2 = left4;
        right2 = right4;
      }
    }
    else {
      *(buf++) = *left3;
      *(left3++) = *buf;
      if (left3 > right3) {
        --nqueues;
        left3 = left4;
        right3 = right4;
      }
    }
  }
```

```
      else {
        if (*left2 <= *left4) {
          *(buf++) = *left2;
          *(left2++) = *buf;
          if (left2 > right2) {
            --nqueues;
            left2 = left4;
            right2 = right4;
          }
        }
        else {
          *(buf++) = *left4;
          *(left4++) = *buf;
          if (left4 > right4)
            --nqueues;
        }
      }
    }
  }

  //This loop is continued until one of the three remaining sequnces
  //are empty.
  while (nqueues == 3) {
    add_comparisons(2);
    add_assignments(2);

    //The smallest element is found
    if (*left1 <= *left2) {
      if (*left1 <= *left3) {
        *(buf++) = *left1;
        *(left1++) = *buf;
        if (left1 > right1) {
          --nqueues;
          left1 = left3;
          right1 = right3;
        }
      }
      else {
        *(buf++) = *left3;
        *(left3++) = *buf;
        if (left3 > right3)
          --nqueues;
      }
    }
    else {
      if (*left2 <= *left3) {
        *(buf++) = *left2;
        *(left2++) = *buf;
        if (left2 > right2) {
          --nqueues;
          left2 = left3;
          right2 = right3;
        }
      }
      else {
        *(buf++) = *left3;
        *(left3++) = *buf;
        if (left3 > right3)
          --nqueues;
      }
    }
  }
```

```
    //Now there is only two sequences left and only one comparison
    //is needed to find the smallest element.
    while (left1 < right1 && left2 < right2) {
      add_comparisons(1);
      add_assignments(2);
      if (*left1 <= *left2) {
        *(buf++) = *left1;
        *(left1++) = *buf;
      }
      else {
        *(buf++) = *left2;
        *(left2++) = *buf;
      }
    }
    while (left1 <= right1 && left2 <= right2) {
      add_comparisons(1);
      add_assignments(2);
      if (*left1 <= *left2) {
        *(buf++) = *left1;
        *left1 = ((left1 < right1) ? *buf : temp);
        ++left1;
      } else {
        *(buf++) = *left2;
        *left2 = ((left2 < right2) ? *buf : temp);
        ++left2;
      }
    }
    if (buf == left1 || buf == left2) {
      stop_count(0);
      return;
    }
    add_assignments(1);

    // The hole is created again
    temp = *buf;
    if (left2 > right2) {
      while (left1 < right1) {
        add_assignments(2);
        *(buf++) = *left1;
        *(left1++) = *buf;
      }
      add_assignments(2);
      *buf = *left1;
      *left1 = temp;
    }
    else {
      while (left2 < right2) {
        add_assignments(2);
        *(buf++) = *left2;
        *(left2++) = *buf;
      }
      add_assignments(2);
      *buf = *left2;
      *left2 = temp;
    }
    stop_count(0);
}

/*****************************************************************
 Non-recursive 4-way-merge sort of the subarray A[left..right] by using
 locations A[buf.. left1-1] or A[right+1..buf] as the extra space. If
```

```
  to_right is 0 then the buffer-area is in A[buf.. left -1], else if
  to_right is 1 then it is in A[buf..buf+right-left]. The assumption is
  that |A[buffer-area]| = |A[left .. right ]|. */

template <class random_access_iterator>

  void i_four_way_msort(random_access_iterator left,
                        random_access_iterator right,
                        random_access_iterator buf, int to_right){

  diff_type i, runs, size, nelements;
  random_access_iterator left1, right1, left2, right2, left3, right3,
    left4, right4, cursor;

  nelements = right - left + 1;

  //The size of a sequence is 1 at the start of the four-way-merge.
  size = 1;

  //The sorting is done when there is only one sequence left with the
  //length of the array.
  while (size < nelements) {

    //The number of sequences is set.
    runs = nelements / size;

    //If there is a remainder this must also be merged.
    if (nelements % size)
      ++runs;
    cursor = left;

    //The 4-way-merge is made.
    for (i = 1; i <= (runs >> 2); ++i) {
      left1 = cursor;
      right1 = cursor + size - 1;
      left2 = left1 + size;
      right2 = right1 + size;
      left3 = left2 + size;
      right3 = right2 + size;
      left4 = left3 + size;
      right4 = (((right3 + size) > right) ? right : right3 + size);
      i_four_way_merge(buf, 4, left1, right1, left2, right2,
                       left3, right3, left4, right4);
      buf += (right4 - left1 + 1);
      cursor += (right4 - left1 + 1);
    }
    runs = runs % 4;

    //If there is a remainder after the merge then the last sequences
    //must be taking care of this way.
    switch (runs) {

      //If there is 3 sequences these are merged with 4-way-merge.
    case 3:
      left1 = cursor;
      right1 = cursor + size - 1;
      left2 = left1 + size;
      right2 = right1 + size;
      left3 = left2 + size;
      right3 = right;
      left4 = right3 + 1;
      right4 = right3;
```

```
      i_four_way_merge(buf, 3, left1, right1, left2, right2,
                       left3, right3, left4, right4);
      break;

      //If there is 2 sequences these are merged with stright_merge
      //which is faster.
    case 2:
      left1 = cursor;
      left2 = left1 + size;
      right1 = left2 - 1;
      right2 = right;
      straight_merge(left1, right1, left2, right2, buf);

      break;

      //If there is one sequence left this is swapped in the buffer.
    case 1:
      change_places(cursor, right, buf);
      break;
    case 0:
      ;
    }

    //The pointers to the sorting area and the buffer is swiched and
    //the sequence size is made 4 times bigger.
    buf = left;
    if (to_right) {
      left += nelements;
      right += nelements;
      to_right = 0;
    }
    else {
      left -= nelements;
      right -= nelements;
      to_right = 1;
    }
    size = size << 2;
  }
}

/*****************************************************************/

template <class random_access_iterator>
  void inplace_merge_sort (random_access_iterator left,
                           random_access_iterator right){

  right--;

  diff_type nelements, half, nrounds_is_odd;

  random_access_iterator start, end, buf, cut;
  cut = right;
  nelements = cut - left + 1;
  half = (nelements >> 1);
  if (nelements == 1)
    return;

  //If there is a remaider then the rounds is odd.
  nrounds_is_odd = (int) ceil(log4((double) half)) % 2;

  //If nrounds_is_odd is 1 the right zone is merged, else
  //the left zone.
```

```
if (nrounds_is_odd) {
  buf = cut - half + 1;
  end = buf - 1;
  start = buf - half;
}
else {
  end = cut;
  start = end - half + 1;
  buf = start - half;
}
i_four_way_msort(start, end, buf, nrounds_is_odd);

//The pointers is moved and the merge is continued on the
//buffer area
cut = cut - half;
nelements = cut - left + 1;
half = (nelements >> 1);

//The 4-way-merge is continued until the part to sort + the buffer is smaller
//than n / log2 n elements. The area just sorted in every loop is merged
//with the previous sorted area.
while (nelements >= (right - left + 1) /
       log2((double) (right - left + 1))) {

  nrounds_is_odd = (int) ceil(log4((double) half)) % 2;
  if (nrounds_is_odd) {
    buf = left;
    start = buf + half;
    end = start + half - 1;
    nrounds_is_odd = 0;
  }
  else {
    start = left;
    end = start + half - 1;
    buf = end + 1;
    nrounds_is_odd = 1;
  }
  i_four_way_msort(start, end, buf, nrounds_is_odd);
  binary_merge(left, left + half - 1, cut + 1, right,
               cut - half + 1);

  cut = cut - half;
  nelements = cut - left + 1;
  half = (nelements >> 1);
}
{
  random_access_iterator last1, last2;
  // Handling of the first O(n / log n) elements.
  tkp(left, cut);

  // Searching of the smallest O(n / log n) elements of two
  // ordered block, the first is left .. cut and the other is
  // cut+1 .. right
  find_smallest_elements(left, cut, right, &last1, &last2);


  // The smallest elements are in left .. last1 and cut+1..last2
  //If the smallest elements are in front already, the job is
  //done.
  if (last1 != cut)
    if (last2 == right){
      move_right(left, cut, right - cut);
```

```
        }
        else {
          straight_merge(last1 + 1, cut, left + 1, left, left);
          binary_merge(left, left + (cut - last1 - 1), last2 + 1,
                       right, (last2 - cut) + last1 + 1);
          tkp(left, left + ((last1 - left) + 1 + (last2 - cut)));
        }
    }
  }
}
#endif //_inplace_merge_sort_a_h
```

### C.3.3 inplace_merge_sort.h

```
#ifndef _inplace_merge_sort_h
#define _inplace_merge_sort_h
#include "../common/defines.h"
#include <vector>
#include <iterator>
#include "../common/logarithms.h"
#include <iostream>
#include "common.h"

namespace cphstl {

  template <class random_access_iterator>
    void four_way_merge(random_access_iterator left3,
                        random_access_iterator right3,
                        random_access_iterator left4,
                        random_access_iterator right4,
                        random_access_iterator left5,
                        random_access_iterator right5,
                        random_access_iterator left6,
                        random_access_iterator right6,
                        random_access_iterator buf) {


    start_count(0);
    value_type tmp;

    // The hole is created by storing the first element of the result area.
    // This is done to get rid of one move operation when two elements are
    // being switched.

    add_assignments(1);
    tmp = *buf;

    // The selection tree which is used to pick out the smallest element.
    random_access_iterator index;
    int sel_tree[7];
    int minimum_leaf;

    // Initialzation of the selection tree.
    sel_tree[6] = (left6 <= right6 ? 6 : -1);
    sel_tree[5] = (left5 <= right5 ? 5 : -1);
    sel_tree[4] = (left4 <= right4 ? 4 : -1);
    sel_tree[3] = (left3 <= right3 ? 3 : -1);

    // sel_tree[2] is a pointer to then smallest element of
    // either the first element in left5-right5 or left6-right6.
    if(sel_tree[6] == -1){
      if(sel_tree[5] == -1)
```

```
      sel_tree [2] = 0;
    else
      sel_tree [2] = 5;
  }
  else {
    if( sel_tree [5] == -1)
      sel_tree [2] = 6;
    else {
      add_comparisons(1);
      sel_tree [2] = (* left5 <= *left6 ? 5 : 6);
    }
  }

  // sel_tree [1] is a pointer to then smallest element of
  // either the first element in left3 -right3 or left4 -right4.
  if( sel_tree [4] == -1) {
    if( sel_tree [3] == -1)
      sel_tree [1] = 0;
    else
      sel_tree [1] = 3;
  }
  else {
    if( sel_tree [3] == -1)
      sel_tree [1] = 4;
    else {
      add_comparisons(1);
      sel_tree [1] = (* left3 <= *left4 ? 3 : 4);
    }
  }

  // If both sel_tree [1] and se_tree [2] is 0, there is nothing to merge.
  // and the function returns.
  if( sel_tree [1] == 0 && sel_tree[2] == 0) {
    stop_count (0);
    return;
  }

  // 4-way-merging is made here.
  // As long as sel_tree [1] or se_tree [2] points to something not 0, there is
  // more to be merged.
  while(sel_tree [1] != 0 || sel_tree [2] != 0) {

    // sel_tree [0] is initialized .
    if ( sel_tree [1] == 0)
      sel_tree [0] = 2;
    else if( sel_tree [2] == 0)
      sel_tree [0] = 1;
    else {
      add_comparisons(1);
      sel_tree [0] =(((( sel_tree [1] == 3) ? * left3 : * left4 )
                      <= ((sel_tree[2] == 5) ? * left5 : * left6 )) ? 1 : 2);
    }

    // Minimum_leaf points to the smallest element i the tree.
    minimum_leaf = sel_tree[ sel_tree [0]];

    // The smallest element is moved over i the bufferarea.
    *buf++ = *(index = ((minimum_leaf == 3) ? left3 :
                        ((minimum_leaf == 4) ? left4 :
                         ((minimum_leaf == 5) ? left5 : left6 ))));

    // The bufferelement is moved to the sorting area and there
```

```
// is made space for the next element.
*index = *buf;
add_assignments(2);

// The pointer to the smallest element that was just moved, is moved one step
// in the subarray. The smallest element is then fund, but if there is no
// elements left the new pointer becomes 0.
switch(minimum_leaf){
case 3:
  if(index == right3){
    sel_tree[3] = -1;
    sel_tree[1] = (( sel_tree[4] == -1) ? 0 : 4);
  }
  else {
    left3 = index +1 ;
    if( sel_tree[4] != -1) {
      sel_tree[1] = ((* left3 <= *left4 ) ? 3 : 4 );
      add_comparisons(1);
    }
  }
  break;
case 4:
  if(index == right4){
    sel_tree[4] = -1;
    sel_tree[1] = (( sel_tree[3] == -1) ? 0 : 3);
  }
  else {
    left4 = index + 1;
    if( sel_tree[3] != -1) {
      sel_tree[1] = ((* left3 <= *left4 ) ? 3 : 4 );
      add_comparisons(1);
    }
  }
  break;
case 5:
  if(index == right5){
    sel_tree[5] = -1;
    sel_tree[2] = (( sel_tree[6] == -1) ? 0 : 6);
  }
  else {
    left5 = index + 1;
    if( sel_tree[6] != -1) {
      sel_tree[2] = ((* left5 <= *left6 ) ? 5 : 6 );
      add_comparisons(1);
    }
  }
  break;
case 6:
  if(index == right6){
    sel_tree[6] = -1;
    sel_tree[2] = (( sel_tree[5] == -1) ? 0 : 5);
  }
  else {
    left6 = index + 1;
    if( sel_tree[5] != -1) {
      sel_tree[2] = ((* left5 <= *left6 ) ? 5 : 6 );
      add_comparisons(1);
    }
  }
 }
}
add_assignments(1);
```

```
    *index = tmp;
    stop_count(0);
  }

  /*****************************************************************
 Non-recursive 4-way-merge sort of the subarray A[left..right] by using
 locations A[buf.. left1-1] or A[right+1..buf] as the extra space. If
 to_right is 0 then the buffer-area is in A[buf.. left-1], else if
 to_right is 1 then it is in A[buf..buf+right-left]. The assumption is
 that |A[buffer-area]| = |A[left .. right ]|. */

  template <class random_access_iterator>

    void four_way_msort(random_access_iterator left,
                        random_access_iterator right,
                        random_access_iterator buf, int to_right){

    diff_type i, runs, size, nelements;
    random_access_iterator left1, right1, left2, right2, left3, right3,
      left4, right4, cursor;

    nelements = right - left + 1;

    //The size of a sequence is 1 at the start of the four-way-merge.
    size = 1;

    //The sorting is done when there is only one sequence left with the
    //length of the array.
    while (size < nelements) {

      //The number of sequences is set.
      runs = nelements / size;

      //If there is a remainder this must also be merged.
      if (nelements % size)
        ++runs;
      cursor = left;

      //The 4-way-merge is made.
      for (i = 1; i <= (runs >> 2); ++i) {
        left1 = cursor;
        right1 = cursor + size - 1;
        left2 = left1 + size;
        right2 = right1 + size;
        left3 = left2 + size;
        right3 = right2 + size;
        left4 = left3 + size;
        right4 = (((right3 + size) > right) ? right : right3 + size);
        four_way_merge(left1, right1, left2, right2,
                       left3, right3, left4, right4, buf);
        buf += (right4 - left1 + 1);
        cursor += (right4 - left1 + 1);
      }
      runs = runs % 4;

      //If there is a remainder after the merge then the last sequences
      //must be taking care of this way.
      switch (runs) {

        //If there is 3 sequences these are merged with 4-way-merge.
      case 3:
        left1 = cursor;
```

```
      right1 = cursor + size - 1;
      left2 = left1 + size;
      right2 = right1 + size;
      left3 = left2 + size;
      right3 = right;
      left4 = right3 + 1;
      right4 = right3;
      four_way_merge(left1, right1, left2, right2,
                     left3, right3, left4, right4, buf);

      //If there is 2 sequences these are merged with stright_merge
      //which is faster.
      break;
    case 2:
      left1 = cursor;
      left2 = left1 + size;
      right1 = left2 - 1;
      right2 = right;
      straight_merge(left1, right1, left2, right2, buf);

      //If there is one sequence left this is swapped in the buffer.
      break;
    case 1:
      change_places(cursor, right, buf);
      break;
    case 0:
      ;
    }

    //The pointers to the sorting area and the buffer is swiched and
    //the sequence size is made 4 times bigger.
    buf = left;
    if (to_right) {
      left += nelements;
      right += nelements;
      to_right = 0;
    }
    else {
      left -= nelements;
      right -= nelements;
      to_right = 1;
    }
    size = size << 2;
  }
}

/*****************************************************************/
template <class random_access_iterator>
  void inplace_merge_sort_t (random_access_iterator left,
                             random_access_iterator right){

  right--;

  diff_type nelements, half, nrounds_is_odd;

  random_access_iterator start, end, buf, cut;
  cut = right;
  nelements = cut - left + 1;
  half = (nelements >> 1);
  if (nelements == 1)
    return;
```

```
//If there is a remaider then the rounds is odd.
nrounds_is_odd = (int) ceil(log4((double) half)) % 2;

//If nrounds_is_odd is 1 the right zone is merged, else
//the left zone.
if (nrounds_is_odd) {
  buf = cut - half + 1;
  end = buf - 1;
  start = buf - half;
}
else {
  end = cut;
  start = end - half + 1;
  buf = start - half;
}
four_way_msort(start, end, buf, nrounds_is_odd);

//The pointers is moved and the merge is continued on the
//buffer area
cut = cut - half;
nelements = cut - left + 1;
half = (nelements >> 1);

//The 4-way-merge is continued until the part to sort + the buffer is smaller
//than n / log2 n elements. The area just sorted in every loop is merged
//with the previous sorted area.
while (nelements >= (right - left + 1) /
       log2((double) (right - left + 1))) {

  nrounds_is_odd = (int) ceil(log4((double) half)) % 2;
  if (nrounds_is_odd) {
    buf = left;
    start = buf + half;
    end = start + half - 1;
    nrounds_is_odd = 0;
  }
  else {
    start = left;
    end = start + half - 1;
    buf = end + 1;
    nrounds_is_odd = 1;
  }
  four_way_msort(start, end, buf, nrounds_is_odd);

  binary_merge(left, left + half - 1, cut + 1, right,
               cut - half + 1);

  cut = cut - half;
  nelements = cut - left + 1;
  half = (nelements >> 1);
}
{
  random_access_iterator last1, last2;
  //Handling of the first n / log n elements.
  tkp(left, cut);

  // Searching of the smallest n / log n elements of two
  // ordered block, the first is left .. cut and the other is
  // cut+1 .. right
  find_smallest_elements(left, cut, right, &last1, &last2);
```

```
      // The smallest elements are in left .. last1 and cut+1..last2
      // If the smallest elements are in front already, the job is
      // done.
      if (last1 != cut)
        if (last2 == right){
          move_right(left, cut, right - cut);
        }
        else {
          straight_merge(last1 + 1, cut, left + 1, left, left);
          binary_merge(left, left + (cut - last1 - 1), last2 + 1,
                       right, (last2 - cut) + last1 + 1);
          tkp(left, left + ((last1 - left) + 1 + (last2 - cut)));
        }
    }
  }
}
#endif //_inplace_merge_sort_h
```

## C.4 In-place sorting with fewer moves

### C.4.1 fewer_moves.h

```
#ifndef _fewer_moves_h
#define _fewer_moves_h

#include <vector>
#include <iterator>
#include <algorithm>
#include <math.h>
#include "../common/logarithms.h"
#include "../common/moves.h"
#include "encoding.h"
#include "../inplace/inplace_merge_sort_a.h"

namespace cphstl {

  /*****************************************************************
   Ceil integer division with remainder. Remainder is b when a % b = 0.
   a and b must both be positive */
  template <class difference_type>
    difference_type hel_div_op_r (difference_type a, difference_type b,
                                  difference_type *rest) {

    difference_type count = 0;
    while (a > 0) {
      a = a - b;
      count++;
    }
    *rest = a + b;
    return count;
  }

  /*****************************************************************
   Ceil integer division with.
   a and b must both be positive */
  template <class difference_type>
  difference_type hel_div_op(difference_type a, difference_type b){
    difference_type count = 0;
    while(a > 0){
      a = a - b;
      count++;
    }
```

```
  return count;
}

/*********************************************************************
 Finds the next power of 2 after n or n if n is a power of 2. */
template <class difference_type>
difference_type find_next_2p ( difference_type n) {
  difference_type i = 31;
  while (!(n & (1 << i)))
    i--;
  for ( difference_type j = i - 1; j >= 0; j--)
    if (n & (1 << j))
      return 1 << (i + 1);

  return n;
}

/*********************************************************************
 Finds 2 in the power of x. x must be positive */
template <class difference_type>
difference_type pow_2 (difference_type x) {
  difference_type res = 1;
  while (x--)
    res <<= 1;
  return res;
}

/*********************************************************************
 Merges d sequences each of length seq_length. d is a power of 2.
 The first sequence starts at left, and they are merged into the area
 starting at buffer. If last is true there's only d_last sequences.
 and the last one is only last_seq_length. Does not divide the sequences
 into blocks */
template <class random_access_iterator>
  void d_way_merge (random_access_iterator left,
                    random_access_iterator buffer,
                    diff_type seq_length,
                    diff_type d,
                    diff_type d_last,
                    bool last,
                    diff_type last_seq_length ) {

  start_count(D_WAY_MERGE);

  random_access_iterator block1, block2;
  int size;

  // The size of the selection tree must be a power of 2 to be balanced.
  if ( last ) {
    size = find_next_2p(d_last );
    if(size == 1){
      size = 2;
    }
  }
  else
    size = d;

  // Array for the selection tree
  diff_type sel_tree [2 * size ];

  // All offsets are initially 0
  for (int i = size; i < 2 * size; i++)
```

```
    sel_tree [i] = 0;

  // If we only have d_last sequenses, the unused leafs are set to -1 to
  // maintain sel_tree as a complete binary tree. -1 indicates an empty
  // sequence
  if(last) {
    for (int i = 2 * size - (size - d_last); i < 2 * size; i++)
      sel_tree [i] = -1;

  }

  // All nodes on the level just above the leaves must point to the child,
  // that points to the smallest element
  int niv2 = size >> 1;
  for (int i = niv2; i < size; i++) {
    int i2 = 2 * i;

    if ( sel_tree [i2] == -1){
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = -1;
      else
        sel_tree [i] = i2 + 1;
    }
    else{
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = i2;
      else {
        block1 = left + (i2 - size) * seq_length;
        block2 = left + ((i2 + 1) - size) * seq_length;
        add_comparisons(1);
        sel_tree [i] = (*block1 < *block2) ? i2 : i2 + 1;
      }
    }
  }

  // Iterate over the rest of the tree, and let all nodes point to the same
  // leaf as is pointed to by the child which points to the leaf pointing on
  // the smallest element.
  for (int i = niv2-1; i > 0; i--) {
    int i2 = 2*i;

    if ( sel_tree [i2] == -1){
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = -1;
      else
        sel_tree [i] = sel_tree [i2+1];
    }
    else{
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = sel_tree [i2];
      else {
        block1 = left + (sel_tree [i2] - size) * seq_length;
        block2 = left + (sel_tree [i2 + 1] - size) * seq_length;
        add_comparisons(1);
        sel_tree [i] = (*block1 < *block2) ? sel_tree [i2] : sel_tree [i2 + 1];
      }
    }
  }

  // The selection tree is now initialize , and the merging can begin!

  value_type temp;
```

```
random_access_iterator next;

// sel_tree [1] holds the leaf that points to the smallest element.
diff_type min_leaf = sel_tree [1];
diff_type offset   = sel_tree [min_leaf];

random_access_iterator buf;
buf = buffer;

// next is the next element to be merged into the buffer area.
next = left + (min_leaf - size) * seq_length + offset;

// A hole is created
add_assignments(1);
temp = *buf;

// Keep extracting elements as long as the selection tree is not empty
while(1) {
  // Increment the offset in the leaf of the active sequence.
  sel_tree [min_leaf] = sel_tree [min_leaf] + 1;

  // If we're looking at the last sequnce in the last group, the end of
  // the sequence is at last_seq_length and seq_length otherwise.
  diff_type the_end;
  if ( last && ((min_leaf - size) + 1) == d_last) {
    the_end = last_seq_length;
  }
  else
    the_end = seq_length;

  // If the offset has reached the end of the sequence, the sequence is dead.
  if (the_end - 1 == offset){
    sel_tree [min_leaf] = -1;
  }

  // Update the tree with the next value in the sequnce (or -1 if the
  // sequence is empty)
  int node = min_leaf >> 1;
  int i2 = node << 1;

  if ( sel_tree [i2] == -1) {
    if ( sel_tree [i2 + 1] == -1)
      sel_tree [node] = -1;
    else {
      sel_tree [node] = i2 + 1;
    }
  }
  else {
    if ( sel_tree [i2 + 1] == -1){
      sel_tree [node] = i2;
    }
    else {
      block1 = left + (((i2 - size) * seq_length) + sel_tree [i2]);
      block2 = left + ((((i2+1) - size) * seq_length) + sel_tree [i2 + 1]);
      add_comparisons(1);
      sel_tree [node] = (*block1 < *block2) ? i2 : i2 + 1;
    }
  }

  // Update the values up to the root of the tree.
  node = node >> 1;
  while (node > 0) {
```

```
      i2 = node << 1;

      if ( sel_tree [i2] == -1){
        if ( sel_tree [i2 + 1] == -1)
          sel_tree [node] = -1;
        else {
          sel_tree [node] = sel_tree [i2+1];
        }
      }
      else {
        if ( sel_tree [i2 + 1] == -1) {
          sel_tree [node] = sel_tree [i2 ];
        }
        else {
          block1 = left + ((( sel_tree [i2] - size ) * seq_length)
                           + sel_tree [ sel_tree [i2 ]]);
          block2 = left + ((( sel_tree [i2 + 1] - size ) * seq_length)
                           + sel_tree [ sel_tree [i2  + 1]]);
          add_comparisons(1);
          sel_tree [node] = (*block1 < *block2) ?
            sel_tree [i2 ] :  sel_tree [i2  + 1];
        }
      }
      node = node >> 1;
    }

    // As long as there's  still  elements in the tree we swap next into the
    // buffer area and finds the next element to be swapped.
    if ( sel_tree [1] != -1) {
      min_leaf = sel_tree [1];
      offset   = sel_tree [min_leaf ];
      add_assignments(2);
      *(buf++) = *(next);
      *next = *buf;
      next = left + (min_leaf - size ) * seq_length + offset ;
    }
    else
      break;
  }

  // Fill the hole!
  add_assignments(2);
  *buf = *next;
  *next = temp;

  stop_count(D_WAY_MERGE);
}

/*****************************************************************
 Merges d sequences each of length seq_length . d is a power of 2.
 The first sequence starts at left , and they are merged into the area
 starting at buffer . If last is true there's only d_last sequences.
 and the last one is only last_seq_length . Divides the sequences
 into blocks of block_size using enc_area for encoding. */
template <class random_access_iterator>
  void d_way_merge_blocks (random_access_iterator left,
                           random_access_iterator buffer ,
                           diff_type seq_length , int d,
                           int d_last , bool last,
                           diff_type last_seq_length ,
                           diff_type block_size ,
                           random_access_iterator enc_area) {
```

```
start_count(D_WAY_MERGE_BLOCKS);

random_access_iterator block1, block2;
int size;

// Initialize the encoding zone with pointers to the next block.
diff_type bit = log_2_ceil(seq_length);

stop_count(D_WAY_MERGE_BLOCKS);
Initialize_encode_area(enc_area, block_size, bit, d);
start_count(D_WAY_MERGE_BLOCKS);

if (last && last_seq_length <= block_size){
  stop_count(D_WAY_MERGE_BLOCKS);
  make_number(enc_area + (2 * bit * (d_last - 1)), bit, 0);
  start_count(D_WAY_MERGE_BLOCKS);
}

// The size of the selection tree must be a power of 2 to be balanced
if (last) {
  size = find_next_2p(d_last);
  if(size == 1){
    size = 2;
  }
}
else
  size = d;

// Array for the selection tree
diff_type sel_tree[2 * size];

// All offsets are initially 0
for (int i = size; i < 2 * size; i++)
  sel_tree[i] = 0;

// If we only have d_last sequenses, the unused leafs are set to -1 to
// maintain sel_tree as a complete binary tree. -1 indicates an empty
// sequence
if (last) {
  for (int i = 2 * size - (size - d_last); i < 2 * size; i++)
    sel_tree[i] = -1;
}

// All nodes on the level just above the leaves must point to the child,
// that points to the smallest element
int niv2 = size >> 1;
for (int i = niv2; i < size; i++) {
  int i2 = 2 * i;

  if (sel_tree[i2] == -1) {
    if (sel_tree[i2 + 1] == -1)
      sel_tree[i] = -1;
    else
      sel_tree[i] = i2 + 1;
  }
  else {
    if (sel_tree[i2+1] == -1)
      sel_tree[i] = i2;
    else {
      block1 = left + (i2 - size) * seq_length;
      block2 = left + ((i2 + 1) - size) * seq_length;
```

```
        add_comparisons(1);
        sel_tree [i] = (*block1 < *block2) ? i2 : i2 + 1;
      }
    }
  }

  // Iterate over the rest of the tree, and let all nodes point to the same
  // leaf as is pointed to by the child which points to the leaf pointing on
  // the smallest element.
  for (int i = niv2 - 1; i > 0; i--) {
    int i2 = 2 * i;

    if ( sel_tree [i2] == -1) {
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = -1;
      else
        sel_tree [i] = sel_tree [i2 + 1];
    }
    else {
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [i] = sel_tree [i2];
      else {
        block1 = left + ( sel_tree [i2] - size ) * seq_length;
        block2 = left + ( sel_tree [i2 + 1] - size ) * seq_length;
        add_comparisons(1);
        sel_tree [i] = (*block1 < *block2) ? sel_tree [i2 ] : sel_tree [i2 + 1];
      }
    }
  }

  // The selection tree is now initialize , and the merging can begin!

  value_type temp;
  random_access_iterator next;

  // sel_tree [1] holds the leaf that points to the smallest element.
  diff_type min_leaf = sel_tree [1];
  diff_type offset   = sel_tree [min_leaf];

  // next is the next element to be merged into the buffer area.
  random_access_iterator buf;
  buf = buffer;
  next = left + (min_leaf - size ) * seq_length + offset ;

  // A hole is created
  add_assignments(1);
  temp = *buf;

  // block_incex holds the index of the next activate block in the sequence
  diff_type block_index;

  // last_last_block_length is the length of the last block in the last
  // sequence. Needed when we are sorting the last part of the vector, which
  // is not a power of 2.
  diff_type last_last_block_length ;
  if ( last_seq_length % block_size == 0)
    last_last_block_length = block_size;
  else
    last_last_block_length = last_seq_length % block_size;

  // last_block_length is the length of the last block in the other sequences
  diff_type last_block_length ;
```

```
if (seq_length % block_size == 0)
  last_block_length = block_size;
else
  last_block_length = seq_length % block_size;

// Keep extracting elements as long as the selection tree is not empty
while (1) {
  // Get the index to the next active block. Is 0 when no blocks are left.
  stop_count(D_WAY_MERGE_BLOCKS);
  block_index = read_number (enc_area + (2 * bit * (min_leaf - size)), bit);
  start_count(D_WAY_MERGE_BLOCKS);

  // Increment the offset in the leaf of the active sequence.
  sel_tree[min_leaf] = sel_tree[min_leaf] + 1;

  // If we're at the end of the last block the sequence is dead.
  if (block_index == 0) {
    if ( last && ((min_leaf - size) + 1) == d_last
        && last_last_block_length == offset + 1)
      sel_tree[min_leaf] = -1;
    else {
      if (!( last && ((min_leaf - size) + 1) == d_last)
          && last_block_length == offset + 1)
        sel_tree[min_leaf] = -1;
    }
  }

  // Update the tree with the next value in the sequnce (or -1 if the
  // sequence is empty)
  int node = min_leaf >> 1;
  int i2 = node << 1;

  if ( sel_tree[i2] == -1) {
    if ( sel_tree[i2 + 1] == -1)
      sel_tree[node] = -1;
    else {
      sel_tree[node] = i2 + 1;
    }
  }
  else {
    if ( sel_tree[i2 + 1] == -1) {
      sel_tree[node] = i2;
    }
    else {
      block1 = left + (((i2 - size) * seq_length) + sel_tree[i2]);
      block2 = left + ((((i2+1) - size) * seq_length) + sel_tree[i2 + 1]);
      add_comparisons(1);
      sel_tree[node] = (*block1 < *block2) ? i2 : i2 + 1;
    }
  }

  // Update the values up to the root of the tree.
  node = node >> 1;
  while (node > 0) {
    i2 = node << 1;

    if ( sel_tree[i2] == -1){
      if ( sel_tree[i2 + 1] == -1)
        sel_tree[node] = -1;
      else {
        sel_tree[node] = sel_tree[i2 + 1];
      }
```

```
    }
    else {
      if ( sel_tree [i2 + 1] == -1) {
        sel_tree [node] = sel_tree [i2 ];
      }
      else {
        block1 = left + ((( sel_tree [i2] - size ) * seq_length)
                        + sel_tree [ sel_tree [i2 ]]);
        block2 = left + ((( sel_tree [i2 + 1] - size ) * seq_length)
                        + sel_tree [ sel_tree [i2  + 1]]);
        add_comparisons(1);
        sel_tree [node] = (*block1 < *block2) ?
          sel_tree [i2 ] :  sel_tree [i2  + 1];
      }
    }
    node = node >> 1;
  }

  // As long as there are elements in the tree ...
  if ( sel_tree [1] != -1){
    // swap next into the buffer area
    add_assignments(2);
    *(buf++) = *next;
    *next = *buf;

    // If the offset equals the block size , a new block must be swapped in,
    // and the offset set to 0.
    if ( offset + 1 == block_size) {

      // If its the last block the sequence is dead
      if (block_index == 0) {
        sel_tree [min_leaf] = -1;
      }
      else {
        sel_tree [min_leaf] = 0;
        diff_type seq_remain;
        if ( last && ((min_leaf - size) + 1) == d_last)
          seq_remain = last_seq_length - block_index;
        else
          seq_remain = seq_length - block_index;
        if (seq_remain <= block_size) {
          // The last block is put in the right place.
          swap_blocks(( left + (min_leaf - size ) * seq_length ),
                      ( left + (min_leaf - size ) * seq_length
                       + block_index), seq_remain);

          // The block index is set to 0 as there are no more blocks.
          stop_count(D_WAY_MERGE_BLOCKS);
          make_number(enc_area + (2 * bit * (min_leaf - size)), bit , 0);
          start_count (D_WAY_MERGE_BLOCKS);
        }
        else {
          swap_blocks(( left + (min_leaf - size ) * seq_length ),
                              ( left + (min_leaf - size ) * seq_length
                               + block_index), block_size );

          // The encoding zone is updated to point to the next block
          stop_count(D_WAY_MERGE_BLOCKS);
          make_number(enc_area + (2 * bit * (min_leaf - size)), bit ,
                      (block_index + block_size ));
          start_count (D_WAY_MERGE_BLOCKS);
        }
```

```
    // Update the tree with the next value in the sequnce (or -1 if the
    // sequence is empty)
    int node = min_leaf >> 1;
    int i2 = node << 1;

    if ( sel_tree [i2] == -1){
      if ( sel_tree [i2 + 1] == -1)
        sel_tree [node] = -1;
      else {
        sel_tree [node] = i2 + 1;
      }
    }
    else{
      if ( sel_tree [i2+1] == -1){
        sel_tree [node] = i2;
      }
      else {
        block1 = left + (((i2 - size ) * seq_length) + sel_tree [i2 ]);
        block2 = left + ((((i2+1) - size) * seq_length)
                         + sel_tree [i2 + 1]);
        add_comparisons(1);
        sel_tree [node] = (*block1 < *block2) ? i2 : i2 + 1;
      }
    }

    // Update the values up to the root of the tree.
    node = node >> 1;
    while (node > 0) {
      i2 = node << 1;

      if ( sel_tree [i2] == -1){
        if ( sel_tree [i2 + 1] == -1)
          sel_tree [node] = -1;
        else{
          sel_tree [node] = sel_tree [i2 + 1];
        }
      }
      else{
        if ( sel_tree [i2 + 1] == -1){
          sel_tree [node] = sel_tree [i2 ];
        }
        else {
          block1 = left + ((( sel_tree [i2] - size ) * seq_length)
                           + sel_tree [ sel_tree [i2 ]]);
          block2 = left + ((( sel_tree [i2 + 1] - size ) * seq_length)
                           + sel_tree [ sel_tree [i2 + 1]]);
          add_comparisons(1);
          sel_tree [node] = (*block1 < *block2) ?
            sel_tree [i2 ] :  sel_tree [i2 + 1];
        }
      }

      node = node >> 1;
    }
  }
}

// Swap next into the buffer and find the smallest element in the rest
min_leaf = sel_tree [1];
offset   = sel_tree [min_leaf ];
next = left + (min_leaf - size ) * seq_length + offset ;
```

```
      }
      else
        break;
    }

    // Fill the hole!
    add_assignments(2);
    *buf = *next;
    *next = temp;
    stop_count(D_WAY_MERGE_BLOCKS);
}

/*********************************************************************
 Sorts a sequence from left to right into buf, using d-way merge*/
template <class random_access_iterator>
  void d_way_sort (random_access_iterator left,
                   random_access_iterator right,
                   random_access_iterator buf,
                   int d,
                    diff_type  block_size,
                   random_access_iterator start) {

  start_count(D_WAY_SORT);
  // place shows if the elements are in the rigth place
  // when al elements in the area is sorted.
  bool place = 1;
   diff_type  seq_length = 1;
   diff_type  nelements;
  nelements = (right + 1) - left;

   diff_type  last_sequence_length;
  int d_last;

  // The number of groups is nelements / d.
   diff_type  groups = hel_div_op_r(nelements, d, &d_last);
  // d_last is the nr of sequences in the remainder and last_sequence_length
  // is the length of the last sequence in the remainder.
  d_last = hel_div_op_r(d_last, seq_length, &last_sequence_length);

  random_access_iterator sort_zone = left;
  random_access_iterator work_zone = buf;
  random_access_iterator temp;
   diff_type  group_offset;
  /*
   if (groups > 1) {
     for (int i = 0; i < groups - 1; i++){
       group_offset = i * d;
        insertion_sort (sort_zone + group_offset, sort_zone + group_offset + d);
     }
     group_offset = group_offset + d;
      insertion_sort (sort_zone + group_offset, sort_zone + group_offset + d_last);

     seq_length = seq_length * d;
     groups = hel_div_op_r(nelements, (seq_length * d), &d_last);
     d_last = hel_div_op_r(d_last, seq_length, &last_sequence_length);
   }
   else {
      insertion_sort (sort_zone, sort_zone + d_last);
     }*/
  while (groups > 1) {
    // If the sequences are longer than the block size we merge them using
    // blocks and encoding zone. Otherwise we merge them as normal sequences.
```

```
  if (seq_length > block_size) {
    for (diff_type i = 0; i < (groups - 1); i++) {
      // Merge all the whole groups
      group_offset = i * d * seq_length;
      stop_count(D_WAY_SORT);
      d_way_merge_blocks(sort_zone + group_offset, work_zone + group_offset,
                         seq_length, d, d_last, false, seq_length,
                         block_size, start);
      start_count(D_WAY_SORT);
    }
    group_offset = group_offset + (d * seq_length);

    // Merge the d_last sequences in the last group
    stop_count(D_WAY_SORT);
    d_way_merge_blocks(sort_zone + group_offset, work_zone + group_offset,
                       seq_length, d, d_last, true, last_sequence_length,
                       block_size, start);
    start_count(D_WAY_SORT);
  }
  else {
    for (diff_type i = 0; i < (groups - 1); i++) {
      group_offset = i * d * seq_length;
      stop_count(D_WAY_SORT);
      d_way_merge(sort_zone + group_offset, work_zone + group_offset,
                  seq_length, d, d_last, false, seq_length);
      start_count(D_WAY_SORT);
    }
    group_offset = group_offset + (d * seq_length);
    stop_count(D_WAY_SORT);
    d_way_merge(sort_zone + group_offset, work_zone + group_offset,
                seq_length, d, d_last, true, last_sequence_length);
    start_count(D_WAY_SORT);
  }

  // Switch sorting direction
  place = !place;

  temp = work_zone;
  work_zone = sort_zone;
  sort_zone = temp;

  // Update sequence length and number of groups.
  seq_length = seq_length * d;
  groups = hel_div_op_r(nelements, (seq_length * d), &d_last);
  d_last = hel_div_op_r(d_last, seq_length, &last_sequence_length);
}

// merge the last bit
if (seq_length > block_size) {
  stop_count(D_WAY_SORT);
  d_way_merge_blocks(sort_zone, work_zone, seq_length, d, d_last, true,
                     last_sequence_length, block_size, start);
  start_count(D_WAY_SORT);
}
else {
  stop_count(D_WAY_SORT);
  d_way_merge(sort_zone, work_zone, seq_length, d, d_last, true,
              last_sequence_length);
  start_count(D_WAY_SORT);
}

// Make sure the elements are located in their original place.
```

```
    if (place) {
      swap_blocks(sort_zone, work_zone, nelements);
    }
    stop_count(D_WAY_SORT);
    return;
}

template <class type>
  bool p(type x, type y){return x < y;}

/*********************************************************************
 The main function. Sorts a vector from left to right inplace.*/
template <class random_access_iterator>
  void mergesort_with_fewer_moves(random_access_iterator left,
                                  random_access_iterator right) {

  start_count(MERGESORT);
  // Step 1 - Split in encoding and mergesort zone
  diff_type encoding_zone, sorting_zone_length, n;
  n = (right + 1) - left;

  diff_type d_min = (int) ceil (log2((double)n) / log2(log2((double)n)));
  int d = find_next_2p(d_min);
  encoding_zone = 2 * d * log_2(n);

  if (encoding_zone >= n) {
    insertion_sort (left, right + 1);
    stop_count(MERGESORT);
    return;
  }

  // make_encoding_zone = 0 when the sequence is sorted
  stop_count(MERGESORT);
  if (!make_encoding_zone(left, right, encoding_zone)){
    return;
  }
  start_count(MERGESORT);

  int k = log_2(n);
  sorting_zone_length = pow_2(k);
  if (sorting_zone_length == n)
    sorting_zone_length = pow_2(k - 1);

  diff_type last_zone_size = (n - encoding_zone) - sorting_zone_length;
  diff_type block_size = (int) ceil (log2((double)n) * log2((double)n));

  random_access_iterator sort_zone_start = left + encoding_zone;

  // Step 2 - Sort
  // Step 2a - Sorting phase

  // Sort the last part of the range, the length is not a power of 2.
  stop_count(MERGESORT);
  d_way_sort (left + encoding_zone + sorting_zone_length, right,
              left + encoding_zone + sorting_zone_length - last_zone_size,
              d, block_size, left);
  start_count(MERGESORT);

  diff_type half = sorting_zone_length >> 1;
  diff_type split = half;

  random_access_iterator first, last;
```

```
    last = sort_zone_start + sorting_zone_length - 1;
    first = sort_zone_start + split;

    // Split the range in blocks with sizes of increaseing power of 2.
    while (split > 0) {
      stop_count(MERGESORT);
      d_way_sort( first , last , sort_zone_start , d, block_size , left );
      start_count(MERGESORT);
      split = split >> 1;
      last = first - 1;
      first = sort_zone_start + split;
    }

    // Step 2b - merging phase
    if ( last - sort_zone_start == 1){
      first = sort_zone_start + 1;
    }
    else{
      first = sort_zone_start;
    }

    split = 1;

    // Merge all the sorted blocks, starting with the smallest ones.
    while (split <= half) {
      inplace_merge (sort_zone_start , sort_zone_start + split ,
                     sort_zone_start + 2 * split );
      split = split << 1;
    }

    if ( last_zone_size != 0) {
      inplace_merge (sort_zone_start , sort_zone_start + sorting_zone_length,
                     right );
    }

    // Step 3 - Sort encoding zone
    inplace_merge_sort ( left , left + encoding_zone);

    // Step 4 - Merge sort zone and encoding zone
    inplace_merge ( left , left + encoding_zone, right );
    stop_count(MERGESORT);
  }
}

#endif //_fewer_moves_h
```

### C.4.2 encoding.h

```
#ifndef _encoding_h
#define _encoding_h

#include <iostream>
#include <vector>
#include <iterator>
#include "../common/moves.h"
#include "../common/defines.h"
#include "../inplace/inplace_merge_sort_a.h"

namespace cphstl {

  /*****************************************************************
```

```
  Alternative encoding zone construction. The idea is to avoid partitioning
  around a median. The zone is made from one pass through the vector.
  This funktion makes a encoding zone at the start of the array
  of size d * 2 * log2 n. The zone is made of elements from
  the array by joining the elements in tubles. The elements in
  each tuple must have different values.
  Returns 0 when the vector is sorted, 1 otherwise. */
template<class random_access_iterator>
  int make_encoding_zone (random_access_iterator left,
                          random_access_iterator right,
                          diff_type encoding_zone) {

  start_count(MAKE_ENC);

  diff_type next_number, current_number;
  current_number = 0;
  next_number = 1;
  while (encoding_zone) {

    // If there's not enough different numbers to create an encoding zone, we
    // have to sort in another way. The only thing to be sorted is the
    // encoding zone since the rest of the elements are all equal.
    if ((left + next_number) == right + 1) {
      // Sort the encoding zone
      inplace_merge_sort (left, left + current_number + 1);

      // Put all the equal numbers from outside the encoding zone into the
      // right place in the sorted encoding zone.
      value_type pivot;
      diff_type split;

      add_assignments(1);
      pivot = *(left + next_number - 1);
      split = current_number >> 1;
      random_access_iterator l1, r1;
      l1 = left;
      r1 = left + current_number;

      while (l1 != r1) {
        add_comparisons(1);
        if (*(l1 + split) > pivot)
          r1 = l1 + split;
        else {
          add_comparisons(1);
          if (*(l1 + split) < pivot)
            l1 = l1 + split;
          else
            break;
        }

        split = (r1 - l1) >> 1;
      }
      diff_type size = current_number - split;

      add_comparisons(1);
      while (*(left + split) == pivot && split <= current_number) {
        add_comparisons(1);
        split++;
        size--;
      }
      add_comparisons(1);
      size ++;
```

```
      add_comparisons(1);
      if (pivot < *(left + split)) {
        swap_blocks(right - size + 1, left + split, size);
      }
      stop_count(MAKE_ENC);
      return 0;
    }

    // If the element on the first place in the tuple is different from the
    // element we are looking at, the latter is put into the tuple.
    add_comparisons(1);
    if (*(left + current_number) != *(left + next_number)) {

      // If the elements doesn't lie next to each other the one pointed to by
      // next_number is swapped, otherwise we proceed to the next tuple.
      if (current_number + 1 != next_number) {
        add_assignments(3);
        iter_swap(left + current_number + 1, left + next_number);
        ++next_number;
      }
      else
        next_number = next_number + 2;

      current_number = current_number + 2;
      encoding_zone = encoding_zone - 2;
    }

    // If the numbers are equal next_number is incremented, and the next
    // number is checked.
    else
      ++next_number;
  }
  stop_count(MAKE_ENC);
  return 1;
}

/*********************************************************************
 This function represents an int in the incoding zone. The int
 is represented by a binary number made by tubles of elements.
 If the first element in the tuble is greater than the other the
 tuble represents a 1, else the opposite. */
template <class random_access_iterator>
  void make_number (random_access_iterator start, diff_type e_pointer_size,
                    diff_type number) {

  start_count(MAKE_NUM);

  // The value of the first bit
  diff_type bit_value = 1;

  // All bits are set to the right value.
  for(diff_type i = 0; i < e_pointer_size; i++){

    if(number & bit_value){
      add_comparisons(1);
      if(*start < *(start + 1)) {
        add_assignments(3);
        iter_swap(start, (start + 1));
      }
    }
    else{
      add_comparisons(1);
```

```
        if(*start > *(start + 1)) {
          add_assignments(3);
          iter_swap(start , ( start + 1));
        }
      }

      start = start + 2;
      bit_value = bit_value << 1;
    }
    stop_count(MAKE_NUM);
  }

  /*********************************************************************
  Reads a number in the incoding zone. */
  template<class random_access_iterator>
    diff_type read_number (random_access_iterator start,
                           diff_type e_pointer_size ) {

    start_count(READ_NUM);

    diff_type number = 0;
    diff_type bit_place = 1;
    while ( e_pointer_size ) {
      add_comparisons(1);
      if (*start > *(start + 1))
        number = number | bit_place;
      bit_place = bit_place << 1;
      e_pointer_size = e_pointer_size - 1;
      start = start + 2;
    }

    stop_count(READ_NUM);
    return number;
  }

  /*********************************************************************
    Initializes the encoding numbers in the encoding area with 'number'*/
  template<class random_access_iterator>
    void Initialize_encode_area (random_access_iterator left ,
                                 diff_type number, diff_type pointer_size ,
                                 diff_type d) {

    // Every sequence in the sort zone is encoded in the
    // encoding zone. This is done by representing the offset
    // from the start of the sort zone to the particular sequence.
    random_access_iterator start;
    start = left ;
    for( diff_type i = 0; i < d; i++){
      make_number(start, pointer_size, number);
      start = start + 2 * pointer_size ;
    }
  }
}

#endif //_encoding_h
```