Labbrapport Labb 1 D0012E

Författare
Adam Gatmon | adagat-1@student.ltu.se
David Malmblad | maldav-1@student.ltu.se
Albin Kullberg | albkul-0@student.ltu.se

LULEÅ TEKNISKA UNIVERSITET

28 november 2023

Innehåll

1 Inledning

I denna laboration kommer vi jämföra två sammansatta algoritmer som i grunden bygger på Merge Sort, men även implementerar algoritmerna Insertion sort och Binary Sort.

Labbens syfte är att ge insikt i de olika fördelarna och nackdelarna hos olika algoritmer beroende på kontexten av den data som ska sorteras. Genom att tillämpa Insertion Sort och Binary Sort på sublistor och samtidigt använda Merge Sort för att effektivt kombinera dem, strävar vi efter att analysera och förstå hur varje algoritm presterar under olika scenarier.

2 Teori

I denna laboration bygger vi två sammansatta sorteringsalgoritmer. Vi börjar med en ursprunglig lista som innehåller n element. Denna lista blir sedan uppdelad i mindre sublistor, och varje sublista är konstruerad för att rymma k element. Det innebär att det totala antalet sublistor blir n/k. Det är värt att notera att om n inte är jämnt delbart med k, kommer vissa sublistor att vara mindre än k medan andra kan vara lika stora.

Varje av dessa sublistor genomgår en av två olika sorteringsalgoritmer: Binary Sort och Insertion Sort. Binary Sort utnyttjar binärsökning för att effektivt placera varje element på rätt plats i den sorterade delen av sublistan. Å andra sidan använder Insertion Sort en enkel metod där varje element jämförs och placeras på rätt plats i den sorterade delen av sublistan. Dessa två sorteringsalgoritmer väljs för att dra nytta av deras effektivitet vid hantering av mindre datamängder. En djupare förklaring av samtliga algoritmer följer nedan.

Efter att varje sublista har sorterats individuellt kommer de sorterade sublistorna att kombineras igen. Denna sammanslagning av sublistor utförs med hjälp av Merge Sort. Merge Sort är en effektiv sorteringsalgoritm som delar upp listan i mindre delar, sorterar varje del separat och kombinerar dem sedan i ordning. Den används här för att effektivt hantera större mängder data och bevara ordningen vid sammanslagningen.

Sammanfattningsvis möjliggör denna kombination av sorteringsalgoritmer en effektiv och snabb sortering genom att optimera prestandan för både små och stora datamängder. Varje steg av processen är avsett att dra nytta av de specifika fördelarna hos de valda sorteringsalgoritmerna.

2.1 Merge Sort

Merge Sort (Mergesort) är en effektiv sorteringsalgoritm som börjar genom att dela upp den ursprungliga listan i mindre delar, så kallade sublistar. Varje sublista konstrueras för att innehålla k element, förutom den sista sublistan som tar in de återstående elementen om antalet element inte är jämnt delbart med k.

Efter att sublistarna har skapats och sorterats genom merge- och insertionsort, går mergesort vidare genom att slå samman dessa sublistor. Processen för sammanslagningen börjar genom att jämföra de första elementen i varje sublista. Det mindre av de två jämförda elementen placeras i den resulterande sorterade listan. Denna process upprepas tills alla element har inkluderats i den slutliga sorteringen.

Vid varje steg jämförs och kombineras elementen på ett sätt som säkerställer att den slutliga listan är i stigande ordning. Merge Sort fortsätter sedan att tillämpa denna jämförelse- och sammanslagningsprocess rekursivt på de bildade sublistorna tills hela listan är sorterad. Den här uppdelningen och sammanslagningen möjliggör en stabil och effektiv sorteringsmetod, vilket gör mergesort särskilt lämpad för hantering av större datamängder.

Tidskomplexiteten för mergesort blir väldigt låg, även i värsta fallet, då varje nivå av sammanslagningar kräver endast en operation för per element, och antalet nivåer blir $\log_2(n)$, därmed totalt bara $O(n \cdot \log n)$.

2.2 Insertion Sort

Insertion Sort (Insertionsort) tar emot en lista med heltal och börjar från det andra elementet (index 1). För varje element jämförs det med de tidigare elementen. Om det är mindre, flyttas det till vänster tills rätt plats hittas. Denna process upprepas tills hela listan är sorterad, och varje element är på sin rätta plats. Insertion Sort är särskilt bra för små dataset och när en del av listan redan är sorterad.

Om man implementerar algoritmen på sättet beskrivet ovan kan rätt plats hittas samtidigt som alla element flyttas efter till rätt plats, och allt detta på endast n operationer. Tidskomplexiteten för Insertionsort i värsta fallet blir därför $O(n)$ för placeringen av ett element, och då detta sker för varje element blir den totala tidskomplexiteten $O(n \cdot n) = O(n^2)$. Om implementationen separerar *förflyttningen av element* från *sökandet av index att lägga in på* krävs två stycken separata genomkörningar av tidskomplexitet $O(n)$, vilket gör den totala tidskomplexiteten till $O((n + n) \cdot n) = O(2 \cdot n^2)$.

2.3 Binary Sort

Binary *Search* är en effektiv algoritm för att hitta ett specifikt värde i en sorterad lista. Algoritmen jämför det sökta värdet med mitten av listan och eliminerar hälften av möjliga positioner beroende på resultatet av jämförelsen. Detta steg upprepas rekursivt i den återstående halvan av listan tills det sökta värdet hittas eller listan är tom. Binary Search har en snabb tidskomplexitet på O(log n), vilket gör den mycket effektiv för stora dataset.

Binary Sort (B-sort) ämnar att utnyttja den höga prestandan av Binary Search i Insertionsort, genom att hitta det närmaste värdet till det värdet som ska läggas in på rätt plats i listan med en Binary Sort-liknande algoritm, och därefter direkt lägga in värdet genom att förskjuta alla större värden ett steg åt höger samtidigt.

Tidskomplexiteten för B-sort blir liknande till insertionsort, men med en huvudsaklig skillnad: sökoperationen minskas drastiskt från $O(n)$ till $O(\log n)$. Om den separerade implementationen av insertionsort använts går då tidskomplexiteten ner från $O((n+n)\cdot n)$ till $O((n+\log n)\cdot n)$. B-sort blir dock teoretiskt sett långsammare i värsta fall än den kombinerade versionen av insertionsort ($O(n^2 + n \cdot \log n)$ mot $O(n^2)$).

3 Metod

För att utföra och utvärdera prestanda hos olika sorteringsalgoritmer har vi implementerat dem i programmeringsspråket Python. Vår metodik innefattar att analysera hur varierande dataset påverkar algoritmernas prestanda. Dessa dataset genereras med olika funktioner som är implementerade i Python.

De tre huvudsakliga typerna av dataset som kommer användas är:

- Unsorted (Helt osorterade element): Där elementen inte har någon specifik ordning när de matas in.
- Mostly Unsorted (Huvudsakligen osorterade element): Där en övervägande majoritet av elementen är i oordnad sekvens.
- Almost Sorted (Nästan sorterade element): Där en del av elementen är i rätt ordning, samt att det är mer sannolikt att stöta på mindre tal ju senare i listan man kommer.

Användningen av olika typer av dataset möjliggör en bedömning av hur förutsättningarna påverkar prestandan för varje algoritm. Efter att testerna har genomförts kan prestanda jämföras genom att analysera och jämföra tids- och minnesanvändning för varje algoritm. Resultaten av dessa jämförelser kommer sedan presenteras i resultatavsnittet. Nämnvärt är att

För att testa prestandan av de två algoritmerna kördes båda genom ett antal tester, där exekveringstiden mättes med hjälp av python-biblioteket timeit. För det allmänna resultatet genomfördes tester med följande varierande parametrar:

- $n = \{2^6, 2^7, \ldots, 2^{15}\}$
- $k = \{1, 10, n/10, n/2, n\}$

Utöver detta genomfördes olika tester i syfte att hitta en metod för att finna ett optimalt k för ett visst givet n. Vad som dock upptäcktes var att variansen mellan tester, även när ett genomsnitt av ett flertal (20) tester användes, var för stor för att säkert kunna bestämma ett specifikt värde. Resultatet står beskriven med relevant diskussion i del 4.2.

Denna metodik ger oss insikt i hur varje algoritm hanterar olika förhållanden och möjliggör en djupare förståelse av deras styrkor och svagheter under olika omständigheter.

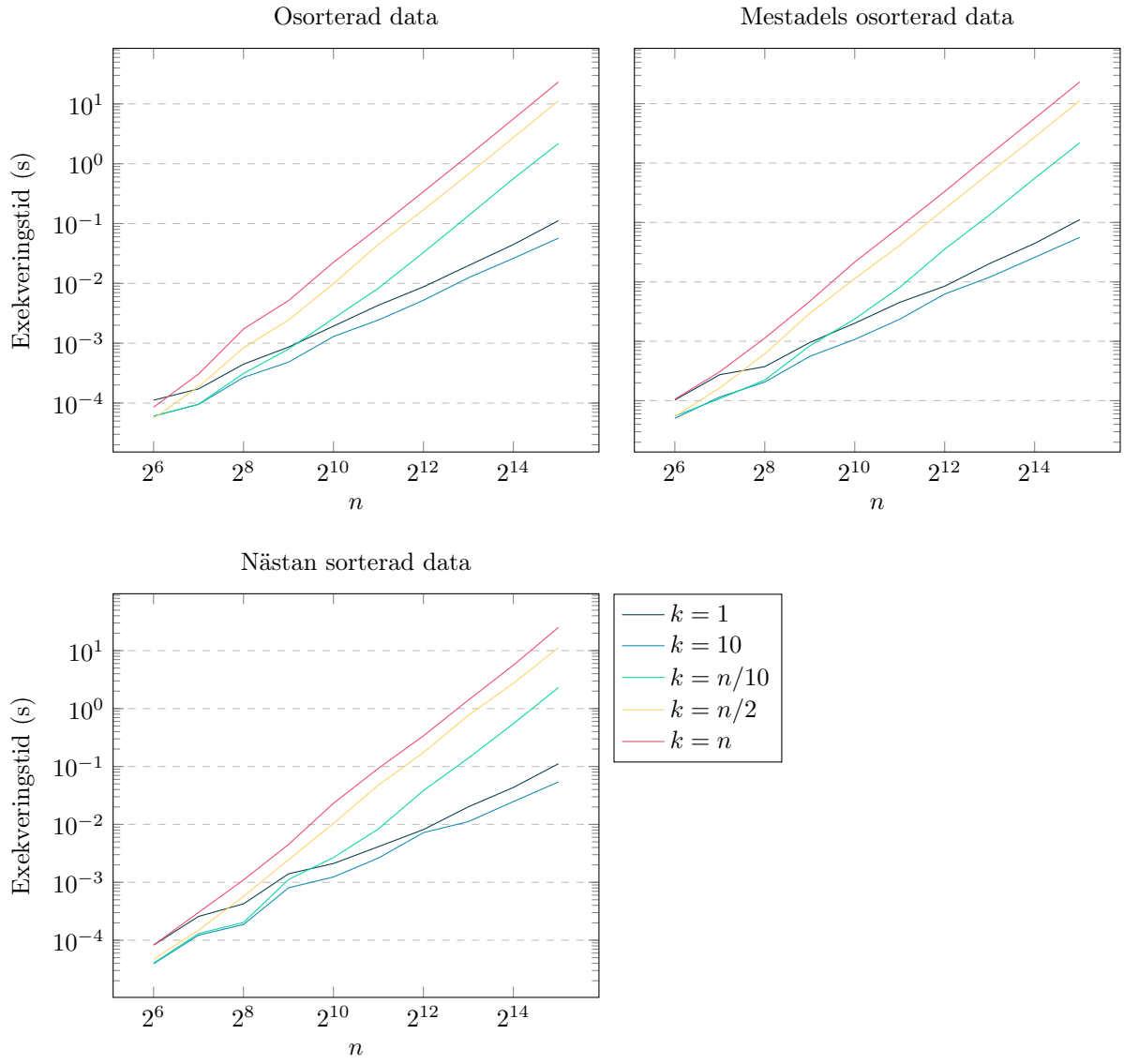
4 Resultat

Flera olika tester genomfördes enligt metoden beskriven i del 3.

4.1 Jämförelse av algoritmer

Resultatet av testerna som genomfördes kan ses i Figur 1 samt 2 (Notera att alla grafer i denna del är logaritmiska grafer). Följande tre observationer kan göras av resultatet:

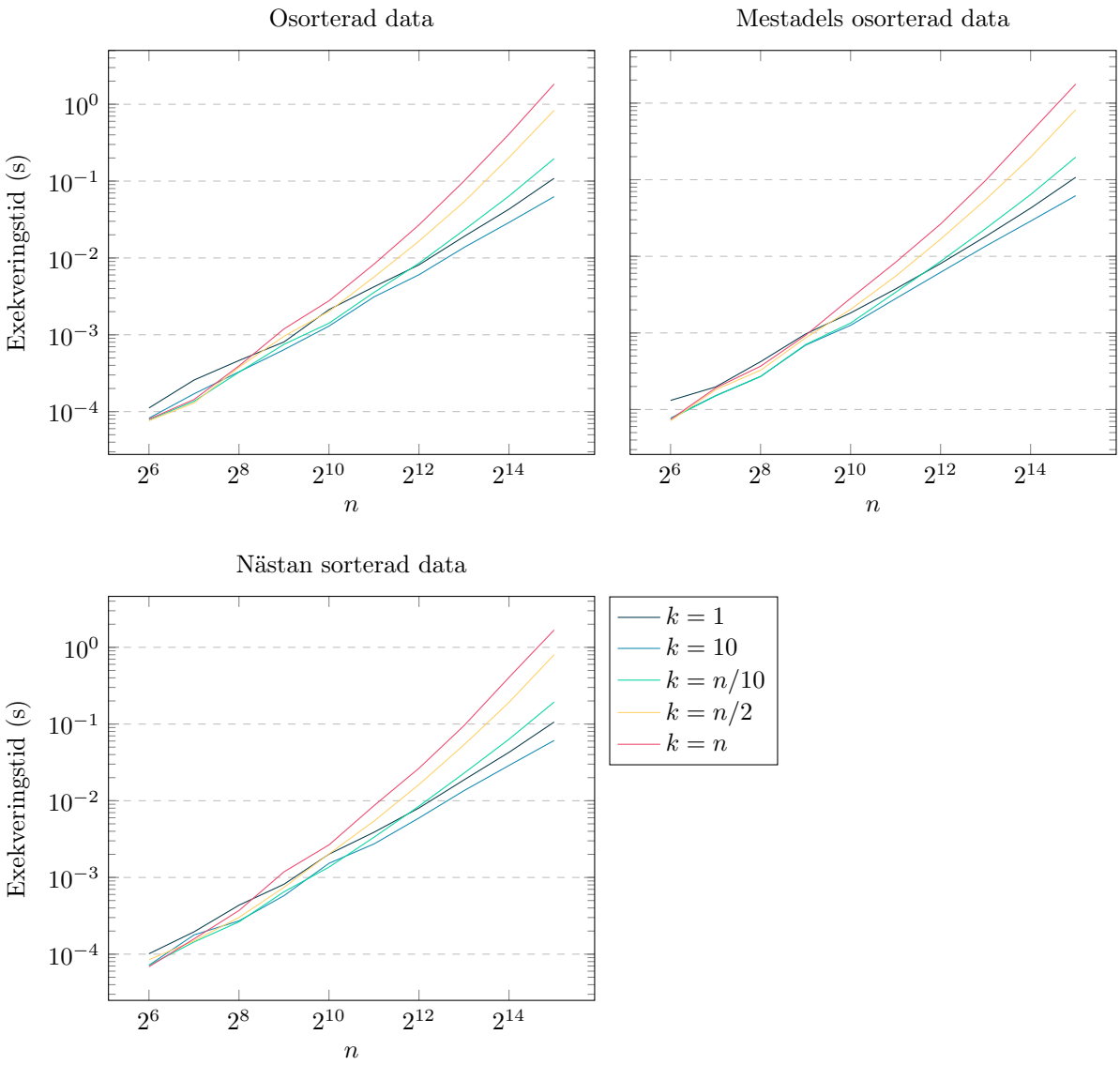
1. Olika värden av k gav stor påverkan på resultatet, framför allt vid stora n.
2. Bästa fallen för algoritmerna, med $10 < k < n/10$, gav ungefär lika bra resultat.
3. Sämsta fallen för algoritmerna, med $k = n$, gav flera gånger bättre resultat med Binary-Mergesort än Insertion-Mergesort vid stora n.



Figur 1: Insertion-Mergesort

4.2 Optimering av k

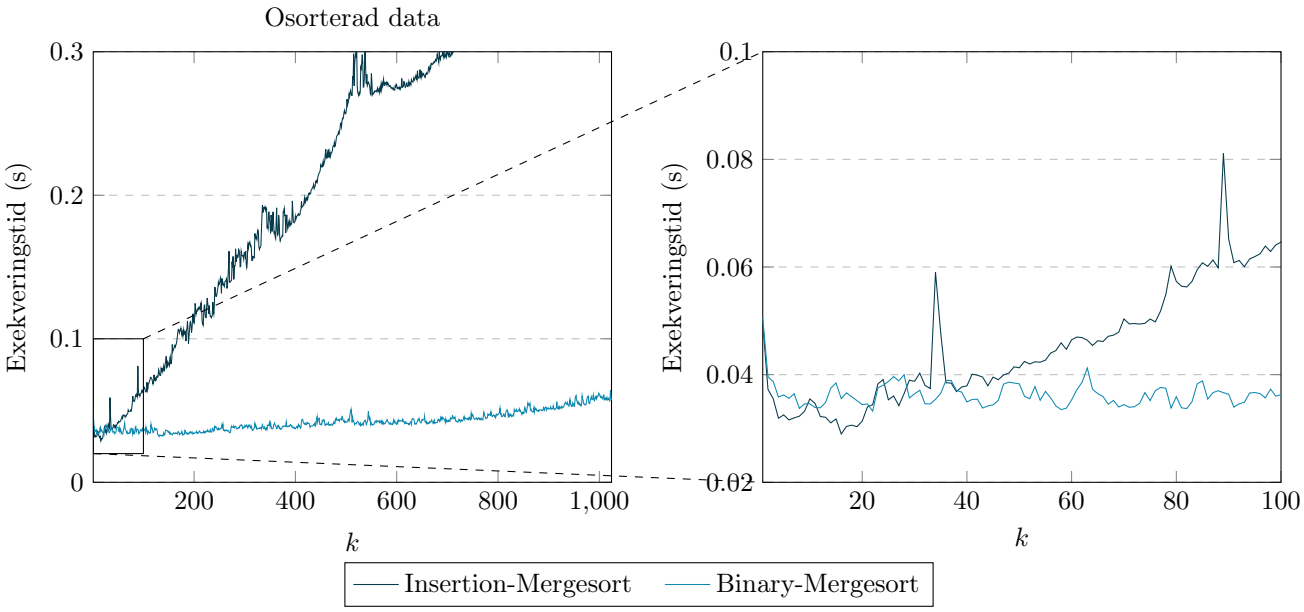
Som nämnt i del 3 kunde ingen ordentlig metod bestämmas för att kunna effektivt optimera k för ett givet n. För att demonstrera detta skapades en graf över som bestämmer exekveringstiden i sekunder till varje värde av k ($1 \leq k \leq n$) på ett specifikt (helt osorterat) dataset där $n = 2^{10} = 1024$. Resultatet kan ses i Figur 3. Denna graf ger även viktigt kontext till del 4.1, då man kan se kopplingen mellan k och exekveringstid, och därmed



Figur 2: Binary-Mergesort

även att de lägre värdena av k ger ett mycket bättre resultat, framför allt för Insertion-Mergesort.

Eftersom exekveringstiden varierade så kraftigt mellan tester kunde inget konkret bästa k tas fram, men i allmänhet låg det i intervallet $20 \leq k \leq 60$.



Figur 3: Relationen exekveringstid-k, för varje k av ett dataset med 1024 värden

5 Diskussion

5.1 Felkällor

I vår valda sorteringsmetod finns det några potentiella felkällor och överväganden som bör beaktas. För det första, om den ursprungliga listan inte är jämnt delbar med det önskade antalet element per sublista (k), kommer vi att ställas inför ojämn fördelning av data i sublistorna. Detta kan leda till att vissa sublistor är betydligt mindre än k medan andra är lika stora. En sådan ojämn fördelning kan påverka prestandan hos de sorteringsalgoritmer som används för varje sublista och kan potentiellt resultera i oönskade resultat.

5.2 Slutsatser

Vid analys av de två algoritmerna blir resultatet väldigt rimligt. I allmänhet blir relationen mellan k och n en fördelning av sorteringsarbetet mellan den underliggande Mergesort och den tillämpade insertionsort/B-sort (Binary-search Insertionsort).

Detta blir tydligt när man tittar på de sämsta fallen för de båda algoritmerna (se Figur 1 och 2). När $k = n$ genomförs hela sorteringen av insertionsort eller B-sort, då hela listan blir en enda "sublista". Här ser man alltså den stora skillnaden mellan Insertionsort och B-sort. I teorin ska B-sort i värsta fall ha en större tidskomplexitet än insertionsort, men eftersom det i Python verkar ta mycket kortare tid att flytta många objekt i en lista samtidigt än att flytta lika många objekt en och en verkar B-sort i praktiken vara ungefär 10 gånger snabbare än insertionsort.

Vidare kan man se, framförallt i den inzoomade versionen av grafen i Figur 3, att vid låga k-värden blir skillnaden mellan insertionsort och B-sort minimal. Detta beror sannolikt på att "fördelen" som diskuterades i föregående stycke blir försumbar när det handlar om att flytta väldigt små listor mot att flytta ett litet antal föremål i en lista.

Man kan även se (som tidigare upptäcktes i del 4.1) att en absolut mergesort ($k = 1$) är långsammare än ett lågt k-värde. Detta kan sannolikt läggas på det faktum att mergesort är väldigt bra på att just lägga ihop två sorterade listor i taget, men med ett lågt k måste även tid läggas på att dela upp datan i väldigt många sublistor, vilket alltså blir ineffektivt arbete som dessutom tar upp mer minne på rekursionsstacken. När detta arbete istället fördelas om till en av Insertionsort-varianterna, som är bra på att sortera små mängder data, fås de bästa delarna av båda sorteringsalgoritmer.