Laborationsrapport

Sortering

Blekinge Tekniska Högskola

Institutionen för Datavetenskap

Datastrukturer och algoritmer

DV1625

16/05/2022

Av Oliver Bölin

olbo20@student.bth.se

Sammanfattning

Syftet med laborationen var att sortera indata med olika typer av algoritmer. De algoritmer som användes i laborationen var *Quicksort* och *Heapsort*, där tre olika varianter skulle implementeras. Programmet skrevs och testades i kodspråket Python. Genom att implementera algoritmerna rätt och göra mindre förändringar på dem så fungerade dem till dess syfte.

1. Inledning

1.1 Bakgrund

Laborationens syfte var att implementera sorteringsalgoritmerna Heapsort och Quicksort i Python och sedan mäta tiderna av dessa. För Quicksort skulle varianterna *första som pivot* och *medianen av tre* som pivot implementeras. För Heapsort skulle d-Heapsort användas.

1.2 Algoritmer

1.2.1 Quicksort

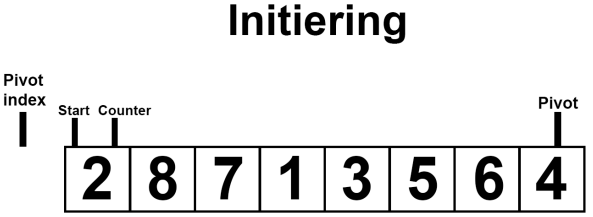
Quicksort är en välanvänd sorteringsalgoritm som är duktig på sortera många element. Den anses ofta använda lite internminne eftersom det går att använda den med *inplace.*
Inplace betyder att den gör sina byten och minnesallokeringar inuti sin egna lista. Den är mindre effektiv på små listor och det är känt att man ska ofta använda mer simplare sorteringsalgoritmer i sådana fall.

1.2.1-1 Hur Quicksort fungerar

Quicksort använder en välkänd sorteringsmetod som kallas för *divide-and-conquer* vilket betyder att man delar in sorteringen i tre olika steg. Dela, sortera delningen och kombinera. Quicksort väljer först ett pivotelement i delningen, sådant man kan sortera runtom. Detta kan väljas valfritt och först görs en så kallad grovsortering. Quicksort är även jämförelse baserad, vilket betyder att den gör jämförelser inuti algoritmen.

Den första delningen, även kallad partitioneringen kan börjas på detta vis. Varje gång counter pekar på ett element som är mindre eller lika med pivot elementet så kommer pivot index flyttas fram ett steg, men den kommer även byta på elementen som ligger på pivot index och counter.

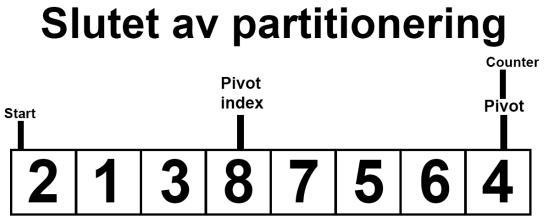
Bilaga 1. En array går in i Quicksort-partitionering



Efter grovsorteringen så är alla element till vänster mindre än pivot och alla till höger mer än pivot. Då gör man ett rekursivt kall på sig själv, för att sedan börja sortera till vänster och till höger

.

Bilaga 2. Slutet av den första Quicksort-partitioneringen



När counter kommer fram till pivot elementet så har alla element jämförts med pivot elementet. De har gjorts förflyttningar när element har hittats som är mindre. Den sista förflyttningen är att flytta pivot-elementet till där pivot indexet är och då återfinns en grov sorterad lista. Detta görs sedan om ett flertal gånger med rekursion för att sedan få en helt sorterad lista. De olika listorna kombineras även automatiskt med hjälp av rekursionen. [1]

1.2.1-2 Quicksort och körtid

Quicksort som en algoritm är relativt snabb och effektiv, speciellt när alla element är unika. Den har en förväntad genomsnittligt fall på

$$T(n) = O(n \log_2 n) \quad (1)$$

Om Quicksort får en dålig rekursion och partitionering, som oftast händer om partitioneringen hamnar på $n - 1$ element eller om datan är delvist sorterad. Då kan man få ett värsta fall på

$$T(n) = O(n^2) \quad (2)$$

I det bästa fallet kan Quicksort partitionera på $n / 2$ och hela tiden skapa två sub-listor, då kan man få ett bästa fall på

$$T(n) = O(n \log_2 n) \quad (3)$$

1.2.2 Heapsort

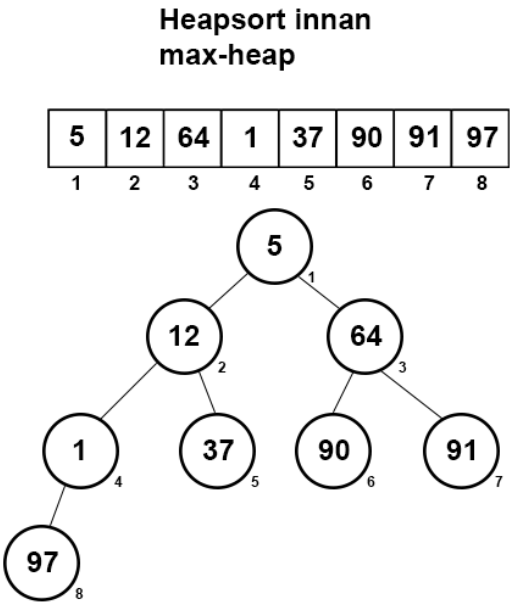
Heapsort är en sorteringsalgoritm som ses som 2 till 3 gånger långsammare än Quicksort i genomsnitt, och används inte lika ofta. Men den har ett bättre värsta fall än Quicksort. Heapsort, sådant som Quicksort är också en *inplace*-algoritm. Heapsort använder något som

heter förälder och barn. Där en förälder kan ha flera barn, när man pratar om *d-heap, 2-heap, 3-heap* så förklarar det hur många barn en förälder har, som man sedan kan visualisera i en träd. Se bilaga 3 för en 2-heap. [2]

1.2.2-1 Hur Heapsort fungerar

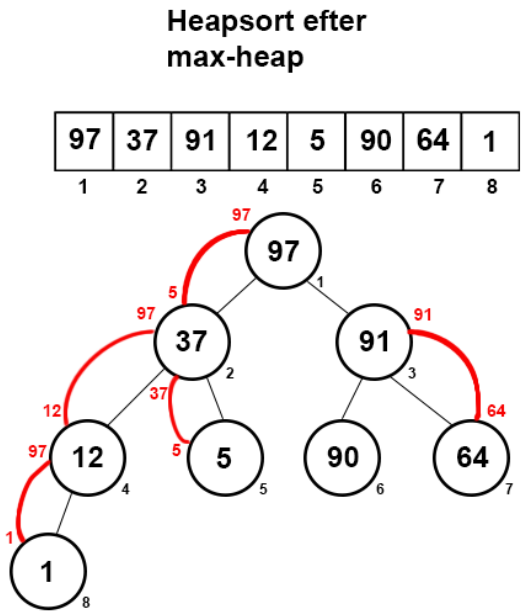
Heapsort sorterar listor genom två olika metoder. Först genom att bygga en heap från datan som finns. Det byggs en *max-heap* som är en grovsorterad lista men den har alltid en förälder som är mer än sig själv. Den andra metoden är att ta ut det största elementet i heapen, även kallat för *root,* och lägga tillbaka det i sorteringslistan längst bak.

Bilaga 3. En osorterad *2-heap* innan en *max-heap* har byggts



När vi bygger upp vår första *max-heap*, så kallas det att göra en *heapify.* Det heapify gör är att den kollar på elementet som ligger i mitten, som är index-nummer fyra i detta fall, och jämför det med sitt största barn. Om de är mindre än det, så byter dem plats. Detta fortsätter tills index ett. Medans används rekursion för att kolla på elementen ovanför, så ett element kan flyttas från längst ner till högst upp.

Bilaga 4. En grovsorterad *2-heap* efter en *max-heap* har byggts



När man har en max-heap så börjar man med det andra steget, att ta ut *root* elementet och lägger det längst bak i listan. Man använder en räknare som börjar längst bak och går till ett. För varje steg gör man en ny *heapify* utan det elementet som togs bort, så att man får en ny *root*. Detta fortsätter tills räknaren kommer till ett. [3]

1.2.2-2 Heapsort och körtid

Eftersom det tar tid n att bygga max-heapen och heapify tar $log_2 n$ i en 2-heap, så är medel, värsta och bästa fall alltid

$$T(n) = O(n \log_2 n) \quad (4)$$

Men i fallet där man använder en d-heap så kommer det att ta $log_d n$ för heapify. Därav kommer ekvationen bli

$$T(n) = O(n \log_d n) \quad (5)$$

1.3 Frågeställning

- Hur presterar algoritmerna med förhållande till varandra?
- Håller sig algoritmerna till teorin?
- Finns det skillnader på beteendet vid olika typer av indata?
- Hur förhåller sig resultaten av förväntad körtid för $T(100\ 000)$?

2. Metod

Testerna genomfördes på Nvidia GTX 970, Intel Core i7 6700K med 4.00GHz och 16.0GB RAM. När testerna genomfördes så togs det tre stycken tidsstämplar och medel av de fem användes. Som indata användes en slumpmässig lista, en sorterad lista och en sorterad bakvänd lista. För heapsort så kollades 2,3,4 och 5-heapsort. Alla programmen skrevs och testades i Python 3.8.10 med kodredigeraren *Visual Studio Code.*

2.1 Quicksort

Quicksort består av 2 funktioner, *Partition* och *Quicksort*. Funktionerna implementerades precis som pseudokoden nedan.

Bilaga 4. Pseudokod av funktionen *Quicksort*, grå marketing visar funktioner [4]

```
QUICKSORT(lst, low, high)

1    if low < high
2        partition_index = PARTITION(lst, low, high)
3        QUICKSORT(lst, low, partition_index - 1)
4        QUICKSORT(lst, partition_index + 1, high)
```

Bilaga 5. Pseudokod av funktionen *Partition*, grå marketing visar Funktioner och returnering [5]

```
PARTITION(lst, low, high)

1   pivot = lst[high]
2   pivot_index = low - 1
3   for j = pivot_index to high - 1
4       if lst[j] ≤ high - 1
5           pivotindex = pivotindex + 1
6           exchange lst[pivot_index with lst[j]
7   exchange lst[pivot_index + 1] with lst[high]
8   return pivot_index + 1
```

Den enda märkbara skillnaden med *första som pivot* och *median av tre som pivot* var att elementet valdes ut, och sedan lades det längst bak innan partitioneringen påbörjades.

2.2 Heapsort

Heapsort består av 3 funktioner, *Heapsort, Build_Max_Heap* och *Heapify* Funktionerna implementerades enligt pseudokoden nedan med lite förändringar för att få det till en d-Heapsort.

Bilaga 5. Pseudokod av funktionen *Heapify*, grå marketing visar funktioner [6]

```
HEAPIFY(lst, i)

1   left = LEFT(i)
2   right = RIGHT(i)
3   if left < lst.heap_size and lst[left] > lst[i]
4           largest = left
5   else largest = i
6   if right < lst.heap_size and lst[left] > lst[i]
7           largest = right
8   if largest ≠ i
9           exchange lst[i] with lst[largest]
10          HEAPIFY(lst, largest)
```

Funktionen *Heapify* ändrades så att den kunde ta in variabelen *heap_size* in i funktionen, istället för att kalla på den. Den ändrades även så att funktionerna *LEFT(i)* och *RIGHT(i)* istället beräknades direkt med en loop så att den kunde kolla på d-barn.

För att ta reda på följande barn så användes formlerna nedan i en loop.

$$barn_1 = d \cdot i + 1 \qquad (6)$$

$$barn_2 = d \cdot i + 2 \qquad (7)$$

$$barn_d = d \cdot i + d \qquad (8)$$

Bilaga 6. Pseudokod av funktionen *Build_Max_Heap*, grå marketing visar funktioner [*Heapsort*, grå **Bilaga 7.**

```
BUILD_MAX_HEAP(lst)

1    lst.heap_size = lst.lenght
2    for i = [lst.lenght / 2] downto 1
3        HEAPIFY(lst, i)
```

Pseudokod av funktionen *Heapsort*, grå marketing visar funktioner [8]

```
HEAPSORT(lst)

1    BUILD_MAX_HEAP(lst)
2    for i = lst.length downto 2
3        exchange lst[1] with lst[i]
4        lst.heap_size = lst.heap_size - 1
5        HEAPIFY(lst, i)
```

2.3 Teoretisk metod

För att beräkna C för heapsort använder vi

$$C = \frac{T(n)}{n \cdot log_2(n)} \quad (9)$$

Kalkulation för Quicksort blir liknande i fallen där partitionering är balanserad

$$C = \frac{T(n)}{n \cdot log_2(n)} \quad (10)$$

Möjligheten finns att få värsta fall där partitioneringen blir dålig och då blir beräkning av C som nedan

$$C = \frac{T(n)}{n^2} \quad (11)$$

3. Resultat

3.1 Faktiska tidsmätningar

Tabell 1. Resultat från Pivot first och Pivot median vid n = 50 000, n = 100 000, n = 1000 och n = 10 000. Resultatet är i sekunder.

Algoritm						
Quicksort						
Pivot First	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.14645346	0.22211782	0.00152388	0.02278812	0.0000001529111966	0.0000001714976916
Sorted	0.51001188	1.37589806	0.00568736	0.1340038	0.00000000568736	0.000000001340038
Sorted Reverse	0.32606804	0.55745816	0.00538342	0.08033998	0.00000000538342	0.0000000008033998
Pivot Median	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.09945896	0.20989996	0.00116272	0.01594816	0.0000001166711989	0.0000001200218634
Sorted	0.0881487	0.18940932	0.00103486	0.01556196	0.0000001038413004	0.0000001171154188
Sorted Reverse	0.09827488	0.19780442	0.001456	0.01492338	0.0000001460998912	0.0000001123096254

Tabell 2. Resultat från d-heapsort vid n = 50 000, n = 100 000, n = 1000 och n = 10 000. Resultatet är i sekunder.

Algoritm						
Heapsort						
2-heap	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.37141554	0.79743722	0.00409788	5.88E-02	0.0000004111949329	4.42E-07
Sorted	0.37552856	0.80498692	0.00464616	0.06347552	0.0000004662111749	0.0000004777008878
Sorted Reverse	0.33150746	0.7308222	0.00396866	0.0588481	0.0000003982285675	0.0000004428760822
3-heap	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.29187722	0.63587444	0.0046388	0.06123738	0.0000007377566921	0.0000007304413895
Sorted	0.28971924	0.61623908	0.00444926	0.05739682	0.0000007076121713	0.0000006846310694
Sorted Reverse	0.270661	0.7308222	0.00440284	0.05323816	0.000000700229515	0.0000006350264425
4-heap	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.26707242	0.6147539	0.00452882	0.05699758	0.0000009088737766	0.000000857899063
Sorted	0.29266092	0.58812272	0.0039814	0.06144882	0.0000007990138832	0.0000009248969009
Sorted Reverse	0.25209978	0.55463684	0.00322044	0.04830746	0.0000006462993595	0.0000007270997237
5-heap	T(n) (s)	T(n) (s)	T(n) (s)	T(n) (s)	C1	C2
n	50 000	100 000	1 000	10 000	1 000	10 000
Random	0.28166	0.57850126	0.00388606	0.05667572	0.000000905413125	0.0000009903657064
Sorted	0.28182386	0.57686862	0.00537322	0.05900926	0.000001251906536	0.000001031142568
Sorted Reverse	0.25612272	0.56845836	0.0032298	0.04839556	0.0000007525111067	0.0000008456761196

3.2 Teoretiska tidsmätningar

Tabell 3. Beräkning av förväntad körtid med **C2 & C1** för Quicksort.

Estimerad tid					
Pivot First	T(n) (s)	T(n) (s)	**Pivot First**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.1193444402	0.25398	C2 med Random	0.1338508654	0.2848515
C1 med Sorted	14.2184	56.8736	C2 med Sorted	3.350095	13.40038
C1 med Sorted Reverse	13.45855	53.8342	C2 med Sorted Reverse	2.0084995	8.033998
Pivot Median	T(n) (s)	T(n) (s)	**Pivot Median**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.09105977339	0.1937866667	C2 med Random	0.09367490683	0.199352
C1 med Sorted	0.08104626831	0.1724766667	C2 med Sorted	0.09140647906	0.1945245
C1 med Sorted Reverse	0.1140283388	0.2426666667	C2 med Sorted Reverse	0.08765564373	0.18654225

Tabell 4. Beräkning av förväntad körtid med **C2 & C1** m för Heapsort.

Estimerad tid					
Heapsort					
2-heap	T(n) (s)	T(n) (s)	**2-heap**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.3209302534	0.68298	C2 med Random	3.45E-01	7.35E-01
C1 med Sorted	0.3638694413	0.77436	C2 med Sorted	0.3728369556	0.793444
C1 med Sorted Reverse	0.3108102383	0.6614433333	C2 med Sorted Reverse	0.3456568209	0.73560125
3-heap	T(n) (s)	T(n) (s)	**3-heap**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.3632930343	0.7731333333	C2 med Random	0.3596907647	0.7193815294
C1 med Sorted	0.348448988	0.7415433333	C2 med Sorted	0.3371324194	0.6742648388
C1 med Sorted Reverse	0.3448135516	0.7338066667	C2 med Sorted Reverse	0.3127056462	0.6254112923
4-heap	T(n) (s)	T(n) (s)	**4-heap**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.3546798223	1.509606667	C2 med Random	0.3347873984	0.71246975
C1 med Sorted	0.3118079863	1.327133333	C2 med Sorted	0.3609327025	0.76811025
C1 med Sorted Reverse	0.252212516	1.07348	C2 med Sorted Reverse	0.2837441319	0.60384325
5-heap	T(n) (s)	T(n) (s)	**5-heap**	T(n) (s)	T(n) (s)
n	50 000	100 000	n	50 000	100 000
C1 med Random	0.3533293341	1.503858649	C2 med Random	0.3328968853	0.7084465
C1 med Sorted	0.4885452732	2.079371746	C2 med Sorted	0.3466034284	0.73761575
C1 med Sorted Reverse	0.2936606957	1.249893893	C2 med Sorted Reverse	0.284261606	0.6049445

4. Diskussion

4.1 Hur presterar algoritmerna med förhållande till varandra?

Det första vi kan se är att *Quicksort med medianen av tre* presterar ungefär 2-3 gånger bättre än Heapsort, vilket togs upp i inledningen. Anledningen till detta må vara att Quicksort partitioneringen kan användas in-place och effektivt när den fungerar som bäst. Men vi ser även att *Quicksort med pivot först* presterar sämre än *Quicksort med medianen av tre* vid sorterade eller nästan sorterade listor, vilket visar att Quicksort inte är kontinuerlig i sin prestation. [9]

4.2 Håller sig algoritmerna till teorin?

De flesta håller sig något nära teorin någorlunda. Många algoritmer håller sig inom en $\pm\ 20\%$ av teorin. Närmast är C1 med 4-Heapsort med omvänd sorterad lista i som estimerades till 100% av de faktiska mätvärdet för 50 000 med sin tidskomplexitet på, (*se ekvation 5*).

Längst ifrån är Sorted Reverse och Reverse för *Quicksort pivot först* med både C1 och C2, med sin tid komplexitet på (*se ekvation 2*) som estimerades till värsta fall på Quicksort så är teorin alldeles för förringande. Enligt teorin så ska våran körning med C1 på 100 000 ta cirka 54 sekunder fast en faktiskt körning tog 0.66 sekunder. Här så kommer, (*se ekvation 3*), att komma mycket närmre eller så är det fel på mätvärdena.

Det är förståeligt att teoretiska data hoppar med $\pm\ 20\%$, då minnet kan jobba annorlunda vid olika mätningar, även fast man har i metoden försökt att göra det så kontinuerligt som möjligt.

4.3 Finns det skillnader på beteendet vid olika typer av indata?

Om man enbart kollar på tabell 1 så ser man definitivt skillnader på beteende för olika typer av indata. Vi ser att Quicksort pivot median presterar 3-5 gånger bättre än *Quicksort pivot först* om data är delvis sorterad. Vi ser även att *2-Heapsort* presterar lite sämre än de övriga *d-Heapsort*.

4.3 Hur förhåller sig resultaten av förväntad körtid för $T(100\ 000)$?

Som det sades innan så är den förväntade körtiden för *Quicksort pivot först* med delvist eller helt sorterade listor väldigt höga, vilket kan bero på att Quicksort presterar sämre vid små listor och ännu sämre om de är sorterade. Den estimerades till cirka 54 sekunder. Här förfaller *4-Heapsort* och *5-Heapsort* också med en 2-3 gånger så hög förväntad körtid. Det finns möjligheter att datorn minneshantering gör det svårt att beräkna mindre sorterade listor. Om sorteringen för de sorterade listorna hade varit cirka 100 gånger snabbare så hade vi fått förväntad körtid som förhållit sig till resultatet.

5. Slutsats

Resultatet visar att Quicksort är snabbare än Heapsort, det visar även att vid de flesta fallen så håller sig teorin till verkligheten med en viss grad fel som kan vara beroende på hårdvara, mjukvara eller minneshantering. Typen av indata spelar även roll då Quicksort presterar annorlunda vid speciella fall sådant som med delvist eller helt sorterade listor. Typen av pivot spelar även roll då *medianen av tre* presterar bäst oavsett typ av indata. Teorin för värsta fallen i Quicksort höll sig inte till de faktiska mätvärdena vilket kan bero på algoritmens prestanda vid små listor.

6. Referenser

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 170, Introductions to algorithms
2. Steven S. Skiena (2008), s. 129, The Algorithm Design Manual
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 151, Introductions to algorithms
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 171, Introductions to algorithms
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 171, Introductions to algorithms
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 154, Introductions to algorithms
7. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 157, Introductions to algorithms
8. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), s. 160, Introductions to algorithms
9. Mark Allen Weiss, (2014), s. 311, Data Structures and Algorithm Analysis in C++