Sorteringsalgoritmer i samband med MAA11 Algoritmer och talteori

Simon Lindqvist

8 juni 2022

Innehåll

Abstrakt

Hösten 2021 togs en ny läroplan i bruk för gymnasierna i Finland. Till innehållet i studieavsnittet *MAA11 Talteori och algoritmer* hör programmering av enkla algoritmer, någonting som är nytt för gymnasiets matematikundervisning. Mera specifikt listas sorteringsalgoritmer som en del av innehållet, vilket leder till nya möjligheter och utmaningar.

I den här pro gradu-avhandlingen presenteras och jämförs olika sorteringsalgoritmer, med målet att få en bättre bild av hurudant innehåll som lämpar sig för studieavsnittet MAA11. Sorteringsalgoritmerna är en viktig del av datavetenskapen, och de har en central roll i lärandet av programmering då de innehåller många viktiga allmänna koncept och strukturer. Vid programmeringen av sorteringsalgoritmerna uppstår ändå många utmaningar, även då det gäller de enklaste algoritmerna. En del algoritmers korrekthet kan vara svåra att begripa, och en del algoritmer är svåra att koda.

Det finns inte så mycket forskning kring ämnet i Finland [1], och den största delen av forskningen om programmeringsundervisningen handlar om undervisningen på högskolor. Då läroplanen uppger väldigt lite information om ett väldigt brett ämne är det svårt för lärare att avgöra vad som borde behandlas. Det skulle därför behövas mera forskning, information och material om ämnet.

Kapitel 1

Inledning

Hösten 2021 trädde en ny läroplan i kraft för de som inledde sina studier på ett gymnasium. Ett av innehållen i den nya läroplanens studieavsnitt *MAA11 Algoritmer och talteori* (MAA11) är programmering av algoritmer [2]. Programmeringen är någonting som är helt nytt för den nya läroplanen vilket medför nya möjligheter och utmaningar. Mera specifikt är en del av det centrala innehållet i MAA11 ”programmering av enkla algoritmer, sorteringsalgoritmer eller en algoritm som anknyter till numerisk lösning av en ekvation” [2, s. 235], och ett av studieavsnittets mål är att ”lära sig att utföra enkla algoritmer genom programmering” [2, s. 234]. Det finns många olika *sorteringsalgoritmer*, och det kan uppstå förvirring om vilka algoritmer som bör behandlas under studieavsnittet och på vilket sätt. Dessutom kan det för lärare vara svårt att avgöra vad som anses med ”programmering av sorteringsalgoritmer”. Målet med den här avhandlingen är att presentera och jämföra olika vanliga sorteringsalgoritmer för att ge en bild av vilka möjligheter och utmaningar som kan förekomma i samband med programmering av dem.

Definition 1.1 (Sorteringsalgoritm)**.** En sorteringsalgoritm är en algoritm vars input är en följd av reella tal $[a_1, a_2, \ldots, a_n]$ och vars output är en permutation $[a'_1, a'_2, \ldots, a'_n]$ av den ursprungliga följden så att $a'_k \leq a'_{k+1}$, för varje $k \in \{1, 2, \ldots, n-1\}$ [3].

Vid programmering av algoritmer är det viktigt att beakta den producerade algoritmens effektivitet. För att kunna jämföra olika algoritmer

med varandra behöver man kunna kvantifiera deras tidseffektivitet. Kapitel 3 *Tidskomplexitet* behandlar det här temat, vilket gör att vi kan jämföra sorteringsalgoritmernas tidseffektivitet.

För att kunna programmera en sorteringsalgoritm är det också nödvändligt att känna till hur den osorterade/sorterade datan lagras i datorns minne, och hur man kan hämta och manipulera datan. Till detta används så kallade *datastrukturer*. Kapitel 4 Datastrukturer behandlar ett par viktiga datastrukturer som sedan används för att beskriva olika sorteringsalgoritmer.

I kapitel 5 Sorteringsalgoritmer presenteras ett antal av de vanligaste och enklaste sorteringsalgoritmerna. Då kan vi jämföra de olika algoritmerna med varandra, och bättre avgöra vilka sorteringsalgoritmer som vore lämpliga att behandlas under MAA11 samt på vilket sätt.

Kapitel 2

Sorteringsalgoritmernas relevans och deras utmaningar för undervisningen

Sorteringsalgoritmerna har en central roll i datavetenskapen. Ofta lärs de ut i en introduktionskurs i algoritmer, vilken är en av de första kurserna man avlägger i universitetsprogram för datavetenskap.

Här diskuteras sorteringsalgoritmernas roll och relevans både för datavetenskapen och för matematikundervisningen, samt vilka utmaningar som finns då det gäller att behandla sorteringsalgoritmer i MAA11.

2.1 Sorteringsalgoritmernas betydelse

Lärandet av sorteringsalgoritmer har flera syften, varav främst två kunde anses som relevanta från gymnasiematematikens synvinkel. Den första är att sorteringsalgoritmer har flera direkta tillämpningar. Många algoritmer använder en sorteringsalgoritm som en viktig procedur [3]. Det här faktumet kan understrykas i MAA11 genom att presentera problem vars lösning är mycket enklare att hitta då man först sorterar datan. Ett exempelproblem är att angivet en mängd med reella tal bestämma den största skillnaden mellan två element i mängden. Om man först sorterar mängden erhåller man nämligen lösningen på problemet genom att helt enkelt beräkna skillnaden

mellan mängdens sista och första element.

Det andra syftet är att de vanligaste sorteringsalgoritmerna använder sig av koncept och tekniker som annars är fundamentala för programmering [3]. Sådana koncept är bland annat slingor, rekursion, ”söndra och härska”, samt användningen av list- och trädstrukturer. Det här innebär att programmering och studerandet av sorteringsalgoritmer inte behöver betraktas som någonting speciellt. Istället handlar det om att utveckla väldigt allmänna färdigheter som krävs för det algoritmiska och det datalogiska tänkandet.

2.2 Datalogiskt tänkande och sorteringsalgoritmer

Till matematikundervisningens uppdrag hör att den studerande ska utveckla sina färdigheter i användningen av datorprogram vid problemlösning [2]. Dit hör också att ”bedöma hur nyttiga olika tekniska hjälpmedel är och vad som begränsar användningen av dem” [2, s. 227]. Användningen av programvara poängteras i målet för varje studieavsnitt i matematiken [2, s. 229-239].

Det upprepade understrykandet av användning av programvara indikerar tydligt att man vill att den studerande utvecklar sina färdigheter i *datalogiskt tänkande*. Datalogiskt tänkande handlar om att formulera och lösa problem, så att lösningarna är representerade i en sådan form som är enkel att implementera med hjälp av en dator [4]. Till det datalogiska tänkandet hör också att känna till vilka problem som en dator kan lösa, och vilka problem som en människa är bättre på att lösa [5, 6], vilket direkt stämmer överens matematikundervisningens uppdrag, enligt grunderna för läroplanen.

Kopplingen mellan lärande av sorteringsalgoritmer och datalogiskt tänkande leder således till att lärandet av algoritmerna främjar målen i läroplanen.

Färdigheter i programmering och datalogiskt tänkande har i övrigt en positiv effekt med avseende på framtida studier [7], och dessutom kan de vara till nytta i de övriga matematikstudierna i gymnasiet. Som ett konkret exempel kan vi lösa en studentexamensuppgift från studentprovet i lång matematik våren 2021 genom att programmera. Problemet löd:

”Talföljden (a_n) definieras rekursivt med formlerna $a_1 = 1$, $a_{2n} = a_n$ och $a_{2n+1} = 1 - a_n$, då $n = 1, 2, 3, \ldots$. Bestäm talföljdens sjunde element a_7 och det 2021:a elementet a_{2021}.”

Problemet kan lösas ”manuellt” genom att exempelvis starta med a_{2021} och bryta ner indexet tills man når basfallet $n = 1$, men en kortare lösning är att skriva om funktionen som programkod direkt på basis av dess definition i uppgiftsbeskrivningen. Om man lyckas med det är problemet mer eller mindre trivialt.

```
1 def a(n):
2     if n==1:
3         return 1
4     if n%2==0:
5         return a(n//2)
6     return 1 - a(n//2)
7
8 print('a(7) =', a(7))
9 print('a(2021) =', a(2021))
```

```
a(7) = 1
a(2021) = 0
```

2.3 Utmaningar med programmering av sorteringsalgoritmer

Programmering är en väldigt bred term som består av många olika steg [6]. Med hjälp av en väldigt grov förenkling kan vi ändå dela in programmeringen av ett problem i två olika tydliga steg. Det första steget är att analysera och modellera problemet, och sedan formulera en sådan lösning som går att tillämpa med hjälp av en dator. Det andra steget är själva kodningen. Då ”översätter” man idén till programkod så att datorn kan utföra de nödvändiga beräkningarna.

Då det gäller programmeringen av en sorteringsalgoritm kan vi tänka oss att processen består av följande två steg:

1. Beskriv (med hjälp av en modell) hur algoritmen ska fungera, och formulera sedan algoritmens olika steg i form av ord eller som pseudokod.

2. Skriv programkoden för algoritmen med hjälp av beskrivningen i Steg 1.

Då man programmerar olika sorteringsalgoritmer stöter man på olika utmaningar i de olika stegen. En del algoritmer baserar sig på en enkel och tydlig idé, men sedan märker man att kodningen av algoritmen är svår. Andra algoritmer kan kännas abstrakta och komplicerade, men då man presenterats de olika stegen som algortmen består av visar det sig vara enkelt att översätta idén till programkod. Som lärare är det bra att känna till och beakta de här utmaningarna då man planerar undervisningsinnehållet för MAA11, och väljer vilka typer av sorteringsalgoritmer som lönar sig att behandla i studieavsnittet. I Kapitel 5. Sorteringsalgoritmer kommer vi att mera specifikt inse vilka utmaningar man kan stöta på då det gäller de olika sorteringsalgoritmerna.

En annan utmaning är att det helt enkelt inte forskats så mycket kring det datalogiska tänkandet i Finland [1]. Dessutom behandlar den största delen av forskningen om undervisningen i programmering om undervisningen på universitet och andra högsskolor. Gymnasister är märkbart yngre och deras hjärnor är generellt inte lika utvecklade, vilket kan betyda att behoven och utmaningarna skiljer sig märkbart.

Kapitel 3

Tidskomplexitet

Lärandet av algoritmer skiljer sig från lärandet av matematik i allmänhet med avseende på att algoritmer är skapade för att utföras av datorer. Då man skapar och använder sig av algoritmer bör man därför vara medveten om datorernas begränsningar. Begränsningarna man måste beakta är vanligtvis *tid* och *rum* [8], det vill säga hur mycket tid och minne en algoritm kräver för att kunna utföras. Den viktigare aspekten är ofta tiden, och för att beskriva hur tidseffektiv en algoritm är pratar man om algoritmens *tidskomplexitet* eller kortare bara *komplexitet*. Det här kapitlet behandlar olika algoritmers komplexitet, vilket senare kommer att behövas för att kunna utvärdera och jämföra olika sorteringsalgoritmer.

Man beskriver ofta tidseffektiviteten av en algoritm med hjälp av en *tidskomplexitetsfunktion* $T(n)$. Funktionens syfte är att beskriva hur körtiden (*running time*) ökar i takt med storleken av algoritmens input (=indata). Tidskomplexitetsfunktionen anger inte den ”riktiga” tiden i exempelvis sekunder, eftersom detta vore omöjligt med tanke på att olika datorer utför samma beräkningar på olika tider. Istället definierar man funktionen så att den anger antalet *elementära operationer* som utförs. Då man antar att de elementära operationerna utförs på samma tid anger tidskomplexitetsfunktionen ett mått på hur tidseffektiv algoritmen är.

Exempel 3.1. Betrakta följande algoritm som ger en utskrift av alla jämna tal i mängden $\{1, 2, 3, \ldots, n\}$.

```
jämnaTal(n):
    for i = 1 to n:
        if i % 2 == 0:
            print(i)
```

De elementära operationerna som utförs är beräkningen av divisionsresten $i\%2$, jämförelsen $i\%2 == 0$, samt utskriften *print(i)*. De två första utförs en gång för varje $i \in \{1, \ldots, n\}$, det vill säga n gånger, och utskriften sker $\lfloor n/2 \rfloor$ gånger. Tidskomplexitetsfunktionen för *jämnaTal* är därmed

$$T(n) = 2n + \left\lfloor \frac{n}{2} \right\rfloor .$$

Det finns dock flera praktiska problem med analysen av algoritmen i föregående exempel. För mera komplicerade algoritmer är det mer eller mindre omöjligt att bestämma det exakta antalet elementära operationer som utförs. För det första kan antalet elementära operationer vara så stort att de i praktiken inte går att räkna, och för det andra kan det i teorin vara omöjligt ifall det exempelvis inte går att avgöra hur många iterationer utförs innan en slinga avbryts. Oftast är den exakta tidskomplexitetsfunktionen ändå inte ens särskilt intressant, utan istället är man intresserad av körtidens storleksordning som funktion av inputens storlek n. Exempelvis kan det räcka att säga att körtiden av funktionen *jämnaTal* i föregående exempel växer linjärt med inputen n, det vill säga $T(n) \sim n$.

Ett annat problem man vanligtvis stöter på är att körtiden inte endast är beroende av inputens storlek, utan den kan variera för olika inputar av samma storlek. Sökalgoritmer är ett sådant exempel. En sökalgoritm genomsöker en mängd efter ett angivet element, och då det eftersökta elementet hittas terminerar algoritmen. Om elementet hittats efter att ha granskat 15 element, så terminerar algoritmen på kortare tid, jämfört med om varje element i mängden genomsöks utan att hitta elementet. När man anger komplexiteten beaktar man därför alltid (om inget annat anges) det värsta möjliga utfallet.

För att kunna beskriva komplexiteten hos olika algoritmer och undgå de problem som diskuterats ovan har man inom datavetenskapen skapat en notation som blivit standardnotation då det gäller att beskriva tidskomplexiteten av olika algoritmer.

Definition 3.2 (O-notation)**.** Funktionen $g(n)$ är av klassen $O(f(n))$ om det finns tal $c \in \mathbb{R}$ och $n_0 \in \mathbb{N}$ så att

$$g(n) \leq cf(n),$$

då $n > n_0$.

Anmärkning 1. Istället för att skriva ”funktionen $g(n)$ är av klassen $O(f(n))$ så använder vi vardagligt beteckningen $g(n) = O(f(n))$.

Den här notationen medför ett par fördelar:

- Körtiden för varje operation som inte är beroende av inputens storlek är $O(1)$.
- Man slipper räkna alla skalära faktorer, och dessutom är komplexiteten skriven i O-notation endast beroende av den förekommande komplexitetsfunktionen som växer snabbast i oändligheten. Exempelvis är $O(4n^2 + n + 50\log_2(n)) = O(n^2)$.

Samtidigt som notationen förenklar beskrivningen av komplexitetsfunktioner, så gömmer den information som ibland kan vara viktig. Betrakta två algoritmer med komplexitetsfunktionerna $T_1(n) = 1000n$ och $T_2(n) = n^2$. Nu är $T_1(n) = O(n)$ och $T_2(n) = O(n^2)$ vilket indikerar att den förstnämnda algoritmen är mera tidseffektiv. Å andra sidan ser man att om inputen är tillräckligt liten ($n < 1000$) så är den andra algoritmen mera tidseffektiv. Vid tillämpningen av algoritmer bör man därmed ibland beakta mera noggranna komplexitetsfunktioner samt göra uppskattningar om hur inputen kommer att se ut.

Det man ytterligare bör notera vad gäller O-notationen är att det är överenskommet att man anger komplexitetsfunktionen som växer möjligast långsammast. På basis av definitionen är det alltså helt rätt att påstå att

en algoritm med komplexiteten $O(n)$ även har komplexiteten $O(2^n)$, vilket skulle ge en förvrängd bild av dess komplexitet.

Det är ofta naturligt att beskriva komplexiteten av en algoritm som funktion av antalet element n i dess input. Då man analyserar sorteringsalgoritmers komplxitet använder man sig alltså av antalet element som ska sorteras. Ett annat sätt att beskriva komplexiteten av en algoritm är genom att använda antalet *bitar* som krävs för att representera inputen [3]. I teorin ökar därmed komplxiteten även för enskilda räkneoperationer då talen blir större, men i praktiken kan man behandla dem som operationer med konstant körtid.

3.1 Vanliga komplexitetsfunktioner

Några av de vanligaste komplexitetsfunktionerna är

- konstant komplexitet $O(1)$,
- logaritmisk komplexitet $O(\log(n))$,
- polynomisk komplexitet $O(n^k)$, för något $k \in \mathbb{N}$,
- exponenitiell komplexitet $O(a^n)$, för något $a > 0$,
- samt någon produkt eller sammansatt funktion av dessa.

Konstant tid

Varje operation vars körtid har en övre gräns som inte är beroende av inputens storlek anses ha konstant tidskomplexitet. Benämningen ”konstant” är därför aningen vilseledande. Det kan ta längre att utföra operationen ”$433494437 \cdot 2971215073$” än operationen ”$2 \cdot 3$”, men om vi exempelvis antar att operanderna är begränsade till heltal som kan representeras av 64 bitar så finns det en fixt övre gräns för tiden det tar att multiplicera två tal.

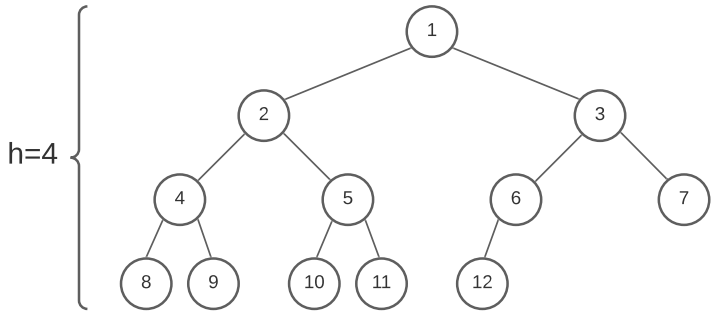
Logaritmisk tid

Logaritmisk komplexitet förekommer ofta i samband med operationer på trädstrukturer och procedurer som fortskrider genom att man delar en mängd i (ungefär) lika stora delar för varje iteration. Höjden h på ett komplett[1] binärt träd med n noder är $h = \lfloor \log_2(n) \rfloor + 1$. Ofta är det logaritmen med basen 2 som syns i komplexitetsfunktionen, men i O-notationen brukar man inte skriva ut basen eftersom basen endast bidrar med en konstant faktor.

$$O(\log_2(n)) = O\left(\frac{\log_k(n)}{\log_k(2)}\right) = O(\log_k(n))$$

En algoritm som kräver logaritmisk tid anses vara väldigt tidseffektiv även om logaritmfunktionen divergerar mot oändligheten. Det här är på grund av att funktionen växer väldigt långsamt. Exempelvis då man fördubblar inputens storlek växer funktionens värde endast med en konstant.

$$\log_2(2n) = \log_2(2) + \log_2(n) = 1 + \log_2(n)$$



Figur 3.1: Höjden i ett komplett binärt träd med n noderär ungefär $\log_2(n)$.

[1] I ett komplett träd är nivåskillnaden mellan två löv högst ett.

Exempel 3.3. Anta att $n \in \mathbb{N}$ och att komplexitetsfunktionen för proceduren *f(n)* är $T(n)$. Då är komplexiteten för följande algoritm $O(T(n)\log(n))$.

```
i = n
while i > 1:
    f(n)
    i = floor(i/2)
```

Polynomisk tid

Polynomisk tid uppstår bland annat vid användningen av slingor. Nästlade slingor bidrar med polynom av högre gradtal. Tidskomplexiteten av en slinga som itererar över en betydande del av inputen erhålls vanligtvis genom att multiplicera n med komplexiteten för operationerna innanför slingan. Det här beror helt enkelt på att antalet upprepningar för operationerna innanför slingan växer linjärt med inputens storlek n.

Exempel 3.4. Betrakta följande funktion som returnerar *false* om inputfältet (Kapitel 4.1) innehåller två identiska element, och annars *true*.

```
allaUnika(A):
    n = length(A)
    for i = 1 to n-1:
        for j = i+1 to n:
            if A[i] == A[j]:
                return false
    return true
```

Inputen för funktionen *AllaUnika* är en mängd av tal, så vi beskriver komplexitetsfunktionen som funktion av antalet element i mängden. Innehållet innanför den yttre slingan upprepas $n-1$ gånger, vilket ger upphov till komplexiteten $O(n \cdot T(n))$, där $T(n)$ är komplexiteten för vad som sker innanför den yttre slingan. Under största delen av iterationerna utför den inre slingan ett betydande antal iterationer i förhållande till n. Således är den inre slingans komplexitetsfunktion $T(n) = n$, och hela algoritmens komplexitet är $O(n^2)$.

Problem för vilka man känner till lösningar vars komplexitetsfunktion är högst ett polynom, det vill säga en polynomfunktion eller en funktion som växer långsammare, bildar mängden av *hanterliga problem* [8]. Problem som antingen saknar sådana lösningar eller vars lösningar man inte känner till bildar mängden av *ohanterliga problem*. Syftet med dessa begrepp är att dela in problem enligt huruvida de i praktiken är beräkningsbara.

Exponentiell tid

Exponentiell tid kan uppstå bland annat då en algoritm går genom alla olika möjligheter av utfall, eller åtminstone en betydande andel av de möjliga utfallen. Exempelvis kan det handla om att algoritmen går genom varje delmängd av en mängd.

Sats 3.5. *Antalet delmängder av en mängd med n element är 2^n.*

Körtiden för algoritmer med exponentiell tidskomplexitet växer väldigt snabbt i förhållande till inputen. Om komplexitetsfunktionen är $T(n) = 2^n$ fördubblas körtiden då inputens storlek växer med 1.

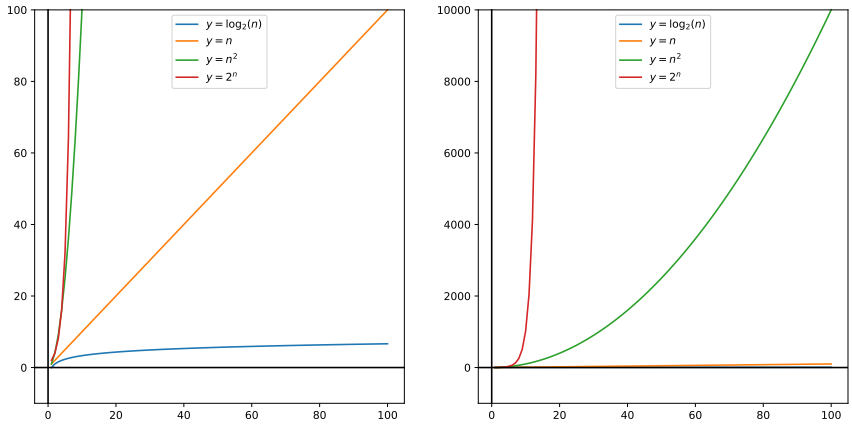
$$T(n+1) = 2^{n+1} = 2 \cdot 2^n = 2T(n)$$

Algoritmer med exponentiell tidskomplexitet kan i praktiken därmed endast användas för problem med små inputs.

Exempel 3.6. Ett kombinationslås med n platser har totalt 10^n olika kombinationer, då varje plats är ett heltal $0 \leq k \leq 9$. En algoritm som öppnar ett sådant lås genom att testa varje kombination har därmed tidskomplexiteten $O(10^n)$.

3.2 Tidskomplexiteten av rekursiva funktioner

Rekursion är ett väldigt användbart verktyg, och med hjälp av rekursion kan man hitta en kort och enkel lösning till många svåra problem. Det är ibland inte så enkelt att bestämma komplexiteten av en rekursivt definierad funktion, och en enkel lösning är ofta en indikator på en hög komplexitet.



Figur 3.2: Bilden visar hur olika några vanliga komplexitetsfunktioner växer i förhållande till varandra.

Exempel 3.7. *Fibonaccis talföljd* definieras rekursivt genom att sätta

$$F_n = \begin{cases} 1, & \text{om } n = 1, 2 \\ F_{n-2} + F_{n-1}, & \text{om } n > 2. \end{cases}$$

Man kan direkt på basis av definitionen med pseudokod beskriva en funktion *fib* som beräknar och returnerar det n:te Fibonaccitalet.

```
fib(n):
    if n == 1 or n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

Varje gång funktionen *fib(n)* kallas utförs först en villkorssats som tar konstant tid, varefter funktionen kallar på sig själv två gånger. Om funktionens tidskomplexitet är T_n så får man då ekvationen

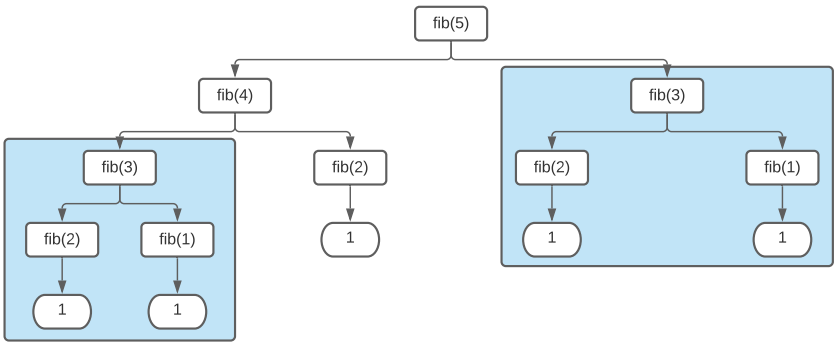
$$T_n = 1 + T_{n-1} + T_{n-2}.$$

Ur denna ekvation kan man visa att tidskomplexiteten är exponentiell $O(2^n)$. Man inser detta även genom att rita ett rekursionsträd (figur 3.3). Ur trädet märker man också att funktionen utför många onödiga repetitioner. I det

här fallet är det alltså mera tidseffektivt att definiera funktionen genom att iterera.

```
fib(n):
    if n == 1 or n == 2:
        return 1
    a = b = 1
    for i = 3 to n:
        a, b = b, a + b
    return b
```

Nu sker inga repetitioner, och man ser att tidskomplexiteten är $O(n)$ vilket är mycket bättre än $O(2^n)$.



Figur 3.3: Rekursionträdet då man kallar *fib(5)*. Ur trädet ser man att funktionen genomför onödiga repetitioner (jämför de blåa rutorna).

Övning 3.8. Betrakta följande algoritm som räknar antalet iterationer i två nästlade for-slingor.

```
räknare = 0
for i = 1 to n:
    for j = 1 to n:
        räknare = räknare + 1
print(räknare)
```

1. Undersök algortimens output för olika värden på $n \in \mathbb{N}$.
2. Bestäm algoritmens tidskomplexitet.

Kapitel 4

Datastrukturer

I det här kapitlet presenteras två olika datastrukturer som är nödvändiga för att kunna beskriva sorteringsalgoritmerna som presenteras i kapitel 5 Sorteringsalgoritmer. För detta behöver man bland annat veta vilka olika operationer man kan utföra på strukturerna, samt operationernas komplexitet för att kunna jämföra och utvärdera de olika sorteringsalgoritmerna.

Då det gäller datastrukturer finns det inte några enhetliga överenskommelser gällande terminologin och implementeringen av dem [8]. Samma datastruktur kan alltså ha olika namn i olika litteratur, och vara implementerade på olika sätt i olika programspråk.

4.1 Fält

Vi kommer här att definiera datastrukturen *fält*. Strukturen är en enkel och intuitiv list-struktur som dessutom påminner om Pythons inbyggda `list`-objekt.

Ett n-dimenesionellt fält (*array*) är en struktur vars element kan tänkas som punkter i ett n-dimensionellt kartesiskt koordinatsystem [8]. Koordinaten för varje element i fältet är en följd av heltalsindex $(i_1, i_2, \ldots, i_n)$. Här fokuserar vi på ett 1-dimensionellt fält. Ett sådant fält kan därmed jämföras med en vektor. Varje element i ett 1-dimensionellt fält nås genom ett unikt heltalsindex som börjar med 1. Således är indexet för fältets sista element n, då n är antalet element i fältet. I pseudokoden som används här kan man

avläsa värdet av elementet på plats k i fältet A genom kommandot $A[k]$. Om man vill hänvisa till ett *delfält* av fältet A, som består av de element $A[k]$, där $p \leq k \leq q$ så skriver vi $A[p : q]$. Med $A[p :]$ avses delfältet av A vars index är minst p, och med $A[: q]$ avses delfältet av A vars index är högst q.

Anmärkning 2. I många programspråk har det första elementet i en fält- eller liststruktur indexet 0.

Vanligtvis implementeras ett fält som en *statisk* struktur. Det innebär att efter att man skapat strukturen kan man inte ändra på dess storlek. Det här innebär att om man vill ta bort element eller lägga till element i strukturen måste den i princip skapas från början. I det här kapitlet behandlar vi det man kunde kalla ett *dynamiskt fält*. Det betyder alltså att själva strukturen av fältet kan ändras, bland annat dess storlek. I fortsättningen menas med ”fält” ett **1-dimensionellt dynamiskt fält**.

Värdena av elementen i ett fält lagras i datorns minne på adresser som är en funktion av elementens index. Det här dikterar vilka fältoperationer som är snabba, respektive långsamma. Exempelvis går det snabbt att lägga till och ta bort element i slutet av fältet, men om man gör det i början eller i mitten, så måste man flytta alla senare element.

Operationer på fält

Vi undersöker olika operationer som kan utföras på fält vars element alla är reella tal. Bland de enklaste algoritmerna som hör till fält är de som går ut på att traversera genom fältet en gång. Exempel på sådana operationer är

- *max(A)*: returnerar fältets största värde,
- *min(A)*: returnerar fältets minsta värde,
- *sum(A)*: returnerar summan av fältets element.

Vi kan implementera funktionen *max(A)* på följande sätt.

```
max(A):
    störst = A[1]
    for i = 2 to length(A):
        if A[i] > störst:
            störst = A[i]
    return störst
```

Summafunktionen kunde beskrivas så här.

```
sum(A):
    summa = 0
    for i = 1 to length(A):
        summa = summa + A[i]
    return summa
```

Precis som med de flesta matematiska problem, så kräver många programmeringsproblem att man delar in problemet i mindre enklare problem, löser dem, och sedan slår ihop lösningarna. Det här kommer att vara nyckeln till att lyckas beskriva annars komplicerade sorteringsalgoritmer som presenteras i följande kapitel. Exempelvis om man vill beskriva en funktion *medelvärde(A)* som returnerar medelvärdet av elementen i fältet A, så är det en bra idé att till först definiera funktionen *sum(A)* på samma sätt som ovan. På så sätt är lösningen mycket enkel.

```
medelvärde(A):
    return sum(A) / length(A)
```

Anmärkning 3. Förutom att lösningen ser enklare ut så är det möjligt att man behöver funkionen *sum(A)* för att lösa andra problem. Det är alltså en bra idé att skapa generella funktioner inte bara för enkelhetens skull, utan man sparar tid då man inte behöver definiera många väldigt specifika funktioner.

Övning 4.1. Beskriv en funktion *min(A)* som returnerar det minsta värdet i fältet A.

Övning 4.2. Beskriv en funktion *innehåller(x, A)* som returnerar *true* om fältet A innehåller elementet x, och annars *false*.

Övning 4.3. Beskriv en algoritm (funktion) vars input är ett fält, och vars output är fältets *typvärde*, dvs. det mest förekommande värdet bland fältets element. Bestäm också algoritmens komplexitet.

Pythons `list()`-objekt

Som tidigare nämnt så innehåller Python (version 3) en inbyggd datastruktur `list` som kan jämföras med vad vi definierade som ett 1-dimensionellt dynamiskt fält. Vi kan kalla dessa strukturer *listor*. Man kan skapa en lista genom hakparenteser och genom att separera elementen med kommatecken. Man kan även skapa en tom lista. Indexeringen börjar vid 0, och man kan hämta ett element i listan genom att sätta indexet innanför hakparenteser.

```
1 vinklar = ['alfa', 'beta', 'gamma']
2 tomLista = []
3 print(vinklar[2])
```

```
gamma
```

Det som är viktigt att komma ihåg då man skapar en lista med ”=”, så hänvisar variabeln inte till den exakta listan tillsammans med dess element, utan den anges en så kallad *pointer* till själva listan vars element kan manipuleras. Det här innebär att man får följande resultat som kan vara förvirrande.

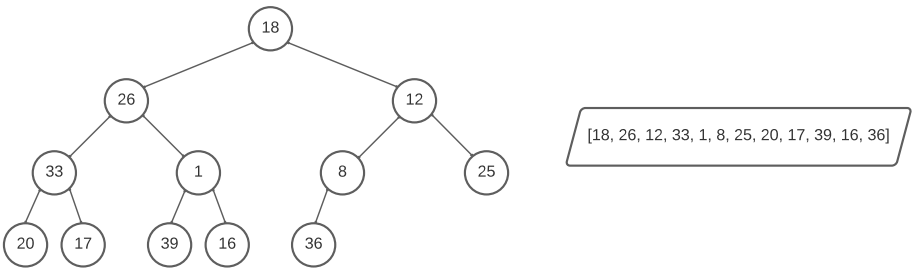
```
1 A = ['a', 'b', 'c']
2 B = A
3 A[0] = 42
4 print('B =', B)
```

```
B = [42, 'b', 'c']
```

Om man alltså önskar att `B` endast ska vara en kopia av `A` kan man istället sätta `B = A[:]`.

4.2 Binär hög

En *binär hög* (binary heap) är ett vänsterkomplett binärt träd. Ett komplett träd är ett träd där nivåskillnaden mellan två löv högst är ett. Ett vänsterkomplett träd innebär att löven ligger så långt till vänster som möjligt. Detta innebär att det är enkelt att implementera en binär hög med hjälp av ett fält (se Figur 4.1).



Figur 4.1: En binär hög och motsvarande fält.

Noden ovanför en viss nod kallas nodens *förälder* (*parent*), och noderna nedanför (om de finns) är nodens *barn* (*child*). Eftersom en nod högst har två barn i en binär hög, så kan vi benämna dem *vänsterbarn* och *högerbarn*. För att kunna utföra operationer på binära högar behöver vi med hjälp av indexet för en nod kunna hänvisa till nodens förälder och barn. Detta kan vi göra med följande funktioner.

```
förälder(i):
    return floor(i/2)

vänsterBarn(i):
    return 2i

högerBarn(i):
    return 2i + 1
```

Målet är att bygga en så kallad *max-hög* (*max-heap*). Det innebär att elementen blir större då man rör sig uppåt i högen. Med andra ord så är fältet

A en max-hög om $A[förälder(i)] \geq A[i]$, för varje $i \in \{2, 3, \ldots, n\}$, där n är antalet element i A. Att bygga en max-hög är nyckeln till högsorteringsalgoritmen som presenteras i nästa kapitel.

För att bygga en max-hög använder vi oss av en hjälpfunktion *högifiera(A,i)* (*heapify*) som startar med elementet *A[i]* och låter det ”rinna” neråt i trädet genom att varje iteration byta plats med det största elementet av nodens barn. Med andra ord så stiger större element uppåt i högen. Funktionen fortsätter sedan rekursivt med att flytta på element så länge som noden har något barn vars element är större än det i noden.

```
högifiera(A, i):
    n = length(A)
    vänster = vänsterBarn(i)
    höger = högerBarn(i)
    störst = i
    if vänster <= n and A[vänster] > A[störst]:
        störst = vänster
    if höger <= n and A[höger] > A[störst]:
        störst = höger
    if störst != i:
        A[i], A[störst] = A[störst], A[i]
        högifiera(A, störst)
```

Vi undersöker algoritmens komplexitet. Höjden i trädet är $h \approx \log_2(n)$. Funktionen innehåller endast beräkningar som kräver konstant tid, och antalet rekursiva kall är högst h. Således är funktionen av klassen $O(\log(n))$.

Nästa steg är att högifiera varje index i fältet börjandes nerifrån. Detta kommer att resultera i en max-hög [3].

```
byggMaxHög(A):
    n = length(A)
    for i = n downto 1:
        högifiera(A, i)
```

Eftersom funktionen *byggMaxHög(A)* högifierar n index, så är dess komplexitet $O(n \log(n))$.

Anmärkning 4. En stor del av noderna (åtminstone hälften) är så kallade löv[1], så en stor del av iterationerna är onödiga. En enkel förbättring av algoritmen vore att skippa högifieringen av noder som är löv eftersom de inte kan sjunka lägre i högen.

Övning 4.4. Förbättra funktionen *byggMaxHeap(A)* genom att inte kalla på *högifiera(A, i)* då noden med indexet i är ett löv..

Övning 4.5. Skriv om funktionerna *förälder(i)*, *vänsterbarn(i)* och *högerbarn(i)* så att de kan användas då man implementerar en binär hög med hjälp av en lista i Python. Tänk på att första indexet i en lista är 0.

[1]Ett löv är en nod som inte har några barn.

Kapitel 5

Sorteringsalgoritmer

I det här kapitlet presenteras ett antal av de vanligaste sorteringsalgoritmerna. Först presenteras de mest primitiva algoritmerna som sorterar ett fält av reella tal på tiden $O(n^2)$. Sedan behandlas några av de mera avancerade algoritmer som har en asymptotisk körtid på $O(n \log(n))$. Slutligen presenteras även ett par sorteringsalgoritmer som med hjälp av olika restriktioner uppnår linjär komplexitet, det vill säga $O(n)$.

Definition 5.1 (Sorteringsalgoritm)**.** En sorteringsalgoritm är en algoritm vars input är en följd av reella tal $[a_1, a_2, \ldots, a_n]$ och vars output är en permutation $[a'_1, a'_2, \ldots, a'_n]$ av den ursprungliga följden så att $a'_k \leq a'_{k+1}$, för varje $k \in \{1, 2, \ldots, n-1\}$ [3].

Vanligtvis har följden som ska sorteras formen av ett fält [3], men inputen kan också vara exempelvis ett binärt träd eller en länkad lista.

Låt oss inledningsvis undersöka sorteringsproblemet genom ett vardagligt och konkret exempel: att sortera en korthand. En metod för att sortera en hand med spelkort är att starta med de osorterade korten i den vänstra handen, och varje gång plocka ut det minsta kortet och placera det längst till höger i den högra handen. Till slut kommer alla kort att vara sorterade i den högra handen. Nu behöver man bara flytta korten till den vänstra handen igen, så att deras ordning bevaras.

Genom att tänka den ursprungliga korthanden som ett fält av tal som ska sorteras, så kan man skriva om idén på följande sätt.

```
sortera(A):
    n = length(A)
    B = []
    for i = 1 to n:
        minsta = 1
        for j = 2 to length(A):
            if A[j] < A[minsta]:
                minsta = j
        B[i] = A.pop(minsta)
    for i = 1 to n:
        A[i] = B[i]
```

Anmärkning 5. Metoden `pop(i)` returnerar och tar ur A bort elementet på plats i. Det är en metod som finns i flera programspråk, Python bland andra.

```
help(list.pop)
```

```
Help on method_descriptor:

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is
   out of range.
```

Då man analyserar algoritmen märker man att den innehåller flera nackdelar. Den första är att funktionen skapar ett tillfälligt fält B som lagrar de sorterade elementen, varefter de flyttas tillbaka till det ursprungliga fältet. För det här tillfälliga fältet måste man reservera lika mycket utrymme som fältet som ska sorteras tar upp. Om man istället byter plats på element i det ursprungliga fältet, så krävs endast ett konstant antal tillfälliga variabler. Man säger att en sådan sorteringsalgoritm sorterar på plats (*in-place*). I fortsättningen menas med *bytPlats(A, i, j)* att man byter plats på elementen på plats i och j i fältet A. En sådan funktion kunde implementeras på följande sätt.

A							B						
2	5	**0**	6	4	1	3							
2	5	6	4	**1**	3		0						
2	5	6	4	3			0	1					
5	6	4	**3**				0	1	2				
5	6	**4**					0	1	2	3			
5	6						0	1	2	3	4		
6							0	1	2	3	4	5	
							0	1	2	3	4	5	6
0	1	2	3	4	5	6	0	1	2	3	4	5	6

Tabell 5.1: Tabellen visualiserar hur funktionen *sortera* sorterar mängden $A = \{0, 1, 2, 3, 4, 5, 6\}$. Elementet markerat i fet stil är det minsta elementet i det osorterade fältet.

```
bytPlats(A, i, j):
    kopia = A[i]
    A[i] = A[j]
    A[j] = kopia
```

En annan nackdel är att implementeringen är väldigt långsam. Under varje iteration söker algoritmen genom hela det osorterade fältet A efter dess minsta element. Algoritmens idé är väldigt enkel, men den innehåller många onödiga jämförelser.

På basis av att algoritmen innehåller två nästlade traverseringar (två for-slingor inuti varandra), så är algoritmens körtid kvadratisk, det vill säga $O(n^2)$. De enklaste kända sorteringsalgoritmerna sorterar på kvadratisk tid. Genom att använda sig av mera abstrakta tekniker är det också möjligt att uppnå komplexiteten $O(n \log(n))$, vilket är mycket bättre än $O(n^2)$. Man kan även visa att det är omöjligt att sortera snabbare om algoritmen baserar sig på jämförelser av element [3]. I det här kapitlet presenteras några av de vanligaste sorteringsalgoritmerna vars komplexitet är $O(n^2)$ eller $O(n \log(n))$.

Om man kan göra olika antaganden om inputen är det även möjligt att sortera snabbare än $O(n \log(n))$. Exempel på ett sådant antagande är att man har information om elementens distribution. Ett par sorteringsalgoritmer som

med hjälp av olika restriktioner av inputen sorterar på linjär tid kommer också att presenteras.

5.1 Urvalssortering

Urvalssortering (*selection sort*) bygger i själva verket på samma idé som exempelalgoritmen som presenterades i början av kapitlet. Den enda skillnaden är att urvalssorteringen sorterar på plats. Det uppnås genom att byta plats på element i fältet, istället för att kopiera elementen till ett tillfälligt skapat fält. Pseudokoden för urvalssorteringsalgoritmen kunde formuleras så här.

```
urvalssortering(A):
    n = length(A)
    for i = 1 to n-1:
        minst = i
        for j = i + 1 to n:
            if A[j] < A[minst]:
                minst = j
            bytPlats(A, i, minst)
```

Man traverserar fältets $n - 1$ första element för att under varje iteration sätta in det minsta elementet i den osorterade delen på rätt plats. För att hitta det minsta elementet traverserar man den osorterade delen.

Sats 5.2. *Tidskomplexiteten för urvalssortering är $O(n^2)$, där n är antalet element i fältet som ska sorteras.*

Övning 5.3. Skapa en tabell som illustrerar hur urvalssorteringen opererar på inputen $[79, 52, 46, 12, 39, 39, 64]$, på samma sätt som Tabell 5.2.

Övning 5.4. Bestäm det exakta antalet jämförelser mellan element som utförs under urvalssorteringen.

Övning 5.5. Skriv om urvalssorteringen så att algoritmen traverserar efter det största osorterade elementet och placerar det i slutet av fältet, istället för att traversera efter det minsta elementet och placera det i början av fältet.

5	2	4	1	7	3	6	**0**
0	2	4	**1**	7	3	6	5
0	1	4	**2**	7	3	6	5
0	1	2	4	7	**3**	6	5
0	1	2	3	7	**4**	6	5
0	1	2	3	4	7	6	**5**
0	1	2	3	4	5	**6**	7
0	1	2	3	4	5	6	7

Tabell 5.2: Visualisering av hur urvalssortering opererar på inputen $[5, 2, 4, 1, 7, 3, 6, 0]$. Elementet med fet stil är det följande osorterade minsta elementet, det vill säga det element som är följande i tur att flyttas till sin rätta plats i fältet.

5.2 Insättningssortering

Insättningssortering (*insertion sort*) använder också nästlade slingor för att sortera ett fält, ett element i taget. Anta att A är det fält som ska sorteras. Anta att $1 \leq k < n$, att delfältet $A[: k]$ är sorterat, och att $A[k+1 :]$ ännu inte sorterats. Då kan man sätta in det följande osorterade elementet $A[k+1]$ på rätt plats i $A[: k+1]$ genom att flytta ner elementet ett index i taget så länge man stöter på element som är större. Ifall elementet råkar vara det minsta hittills sorterade så kommer det att hamna längst fram i fältet. Vi beskriver först en funktion *insättning(A,i)* som i fältet A sätter in elementet $A[i]$ på rätt plats i $A[: i+1]$, där vi antar att $A[: i]$ är ett sorterat fält.

```
insättning(A, i):
    while i >= 2 and A[i] < A[i-1]:
        bytPlats(A, i, i-1)
        i = i-1
```

För att sortera hela fältet krävs nu endast att man kallar på funktionen `insättning(A, i)` för varje index $i \in \{2, 3, \ldots, n\}$, där n är fältets längd.

```
insättningssortering(A):
    for i = 2 to length(A):
        insättning(A, i)
```

10	37	67	74	<u>85</u>	<u>56</u>	59
10	37	67	<u>74</u>	<u>56</u>	85	59
10	37	<u>67</u>	<u>56</u>	74	85	59
10	<u>37</u>	<u>56</u>	67	74	85	59
10	37	56	67	74	85	59

Tabell 5.3: Tabellen illustrerar hur operationen `insättning(A, 6)` jämför och byter plats på element i $A = [10, 37, 67, 74, 85, 56, 59]$. De understreckade elementen hänvisar till vilka element som är i tur att jämföras och eventuellt byta plats. Observera att de fem första elementen redan är sorterade.

5	**6**	2	1	3	4	7	0
5	6	**2**	1	3	4	7	0
2	5	6	**1**	3	4	7	0
1	2	5	6	**3**	4	7	0
1	2	3	5	6	**4**	7	0
1	2	3	5	6	4	**7**	0
1	2	3	4	5	6	7	**0**
0	1	2	3	4	5	6	7

Tabell 5.4: Visualisering av insättningssortering på fältet $[5, 6, 2, 1, 3, 4, 7, 0]$. Elementen till vänster om delaren är redan sorterade, och elementen till höger är ogranskade. Elementet med fet stil är det element som är i tur att sättas in på rätt plats i den sorterade delen.

Sats 5.6. *Tidskomplexiteten för insättningssortering är $O(n^2)$, där n är antalet element i fältet som ska sorteras.*

Bevis. Proceduren *insättning(A,i)* utförs $n - 1$ gånger. Eftersom den utförs på tiden $O(n)$, så är insättningssorteringens komplexitet $O(n \cdot (n-1)) = O(n^2 - n) = O(n^2)$. □

Effektivare insättningssortering

Även om tidskomplexiteten för insättningssortering är $O(n^2)$ så sorterar algoritmen små inputs effektivt [3]. Vi undersöker hur man skapa en algoritm

som mera effektivt sorterar ett fält av reella tal med hjälp av insättning. Ett sätt är att dela fältet i två lika stora delfält, sortera dessa och sedan sammanfoga delarna. Anta att funktionen *sammanfoga(A,i)* sammanfogar fältet A till ett sorterat fält, då delfälten $A[:i]$ och $A[i+1:]$ redan är sorterade. Den önskade algoritmen kan nu beskrivas på följande sätt.

2	5	6	9	1	3	4	7	8
1	2	3	4	5	6	7	8	9

Tabell 5.5: De sorterade delfälten $A[:4]$ och $A[5:]$ sammanfogas till ett enda sorterat fält.

```
effektivInsättningssortering(A):
    m = floor(length(A)/2)
    insättningssortering(A[:m])
    insättningssortering(A[m+1:])
    sammanfoga(A, m)
```

Låt oss till följande motivera varför den här algoritmen är mera effektiv än den ursprungliga insättningssorteringen. Anta att komplexitetsfunktionen för *insättningssortering(A)* är

$$T_1(n) = an^2 + bn + c,$$

där n är antalet element i A, och a, b, c är konstanter. Anta dessutom att komplexiteten för funktionen *sammanfoga(A,i)* är $T_2(n) = dn + e$, där d och e är konstanter. Vi kan göra detta antagandet eftersom funktionen *sammanfoga(A,i)* kan implementeras med endast en slinga som traverserar A. Då är hela algoritmens tidskomplexitet

$$\begin{aligned} T(n) &= 2T_1\left(\frac{1}{2}n\right) + T_2(n) \\ &= 2\left(a\left(\frac{1}{2}n\right)^2 + b\left(\frac{1}{2}n\right) + c\right) + dn + e \\ &= \frac{1}{2}an^2 + (b+d)n + 2c + e. \end{aligned}$$

Den relevanta termen i funktionens uttryck är $\frac{1}{2}an^2$. Resultatet är således att då antalet element är stort så är körtiden ungefär den halva av körtiden för den ursprungliga insättningssorteringen.

Anmärkning 6. Genom att dela in fältet i flera delfält förbättras körtiden ytterligare. Senare kommer vi att se att *samsortering* är en sorteringsalgoritm som använder en liknande metod för att uppnå tidskomplexiteten $O(n\log(n))$.

Övning 5.7. Skapa en tabell som illustrerar hur insättningssorteringen opererar på inputen $[12, 98, 75, 0, 44, 33, 74]$, på samma sätt som Tabell 5.4.

Övning 5.8. Skriv om funktionen *insättningssortering(A)* så att outputen är en avtagande följd istället för växande. [3]

Övning 5.9. Skriv om funktionen *insättningssortering(A)* som Python-kod, där inputen A är en lista. Observera att indexeringen för ett `list`-objekt börjar med 0.

Övning 5.10. Insättningssorteringen är speciellt långsam om många små element i början är i slutet av tabellen. Man kan alltså göra algoritmen mera effektiv om man genom att approximera först kan flytta varje element till sin ungefärliga plats. Bestäm insättningssorteringens komplxitet om man antar att varje element i inputfältet ligger högst k steg från sin rätta plats.

5.3 Bubbelsortering

Bubbelsortering (*bubble sort)* är en annan primitiv men långsam sorteringsalgoritm. Algoritmen är mycket enkel att implementera som kod, men det kan vara svårt att se varför algoritmen fungerar. Idén till algoritmen är att upprepade gånger byta plats på på varandra följande element i inputfältet, tills hela inputen är sorterad. Betrakta följande slinga som itererar över hela fältet, och under varje iteration byter plats på två efter varandra följande element om de inte är i rätt ordning förhållande till varandra.

```
for i = 2 to length(A):
    if A[i-1] > A[i]:
        bytPlats(A, i-1, i)
```

I stort sett flyttas små element närmare början av fältet, och stora element flyttas närmare slutet. Speciellt så har fältets största element hamnat sist i

fältet. Det här betyder att om man gör exakt samma procedur en gång till, så kommer fältets näststörsta element också hamna på sin rätta plats, det vill säga näst sist. Vidare om man utför proceduren n gånger så kommer hela fältet slutligen att vara sorterad. Dessutom kan man göra varje iteration snabbare genom att låta bli att utföra jämförelser av de element i slutet som redan garanterat är ordnade. På det här sättet får vi det som kallas för bubbelsortering.

```
bubbelsortering(A):
    n = length(A)
    k = n
    for i = 1 to n:
        for j = 2 to k:
            if A[j-1] > A[j]:
                bytPlats(A, j-1, j)
        k = k - 1
```

5	2	3	7	4	1	0	6
2	3	5	4	1	0	6	7
2	3	4	1	0	5	6	7
2	3	1	0	4	5	6	7
2	1	0	3	4	5	6	7
1	0	2	3	4	5	6	7
0	1	2	3	4	5	6	7

Tabell 5.6: Illustration av hur bubbelsortering opererar på inputen $[5, 2, 3, 7, 4, 1, 0, 6]$.

Sats 5.11. *Tidskomplexiteten för urvalssortering är $O(n^2)$, där n är antalet element i fältet som ska sorteras.*

Bevis. Anta att fältets längd är n. If-satsen utförs alltid på konstant tid. Därmed är tidskomplexiteten för den innersta for-slingan $O(n)$ och vidare är tidskomplexiteten för den yttersta for-slingan, dvs hela algoritmen, $O(n^2)$. □

Övning 5.12. Om varje element i inputen är högst k steg från sin rätta plats, så kommer fältet att vara sorterat efter k iterationer. Det här betyder att inga byten sker de sista $n-k$ iterationerna, men ändå fortsätter jämförelserna. Skriv om pseudokoden till funktionen *bubbelsortering*, så att itereringen avbryts då fältet är sorterat.

5.4 Samsortering

Sorteringsalgoritmerna som hittills presenterats är relativt enkla, men men priset man betalar är en långsam kvadratisk körtid. Till följande introducerar vi tre olika algoritmer som sorterar på en mycket förbättrad tid $O(n\log(n))$, men som kan anses som mera komplicerade och använder sig av abstrakta koncept.

Söndra och härska

Samsortering (*merge sort*) är en av flera sorteringsalgoritmer som bygger på tekniken ”söndra och härska” (*divide-and-conquer*). Det här är en problemlösningsteknik som går ut på att dela in ett problem i mindre delproblem, som i sin tur delas in i mindre delproblem med hjälp av rekursion. Tekniken består av tre olika steg. [3]

1. **Söndra.** Dela in problemet i mindre delproblem som är av samma typ.
2. **Härska.** Då problemet delats in i tillräckligt små delproblem (basfall) med hjälp av rekursion kan dessa små delproblem lösas direkt och konkret.
3. **Sammanfoga.** Då man löst de små delproblemen sammanfogar man lösningarna till en enda lösning som löser det ursprungliga problemet.

Samsortering använder den här tekniken så här.

1. **Söndra.** Om det osorterade fältet består av åtminstone två element; dela fältet i två lika stora delfält genom att dela det på mitten.

2. **Härska.** Då fältet har delats in i delfält så att varje delfält innehåller endast ett element så har man nått rekursionens basfall. Ett fält som innehåller endast ett element är ett sorterat fält!

3. **Sammanfoga.** Parvis sammanfogar man de splittrade delfälten till ett sorterat fält.

Dessa tre steg kunde med hjälp av pseudokod beskrivas på följande sätt.

```
samsortering(A, v, h):
    if v < h:
        m = floor( (v+h) / 2 )
        samsortering(A, v, m)
        samsortering(A, m+1, h)
        sammanfoga(A, v, m, h)
```

Inputen består av tre parametrar: A som är fältet som ska sorteras, samt v (vänster) och h (höger) som anger vänster- samt högerindexet för det intervall av fältet som ska sorteras, där v och h är inkluderade. Då man vill sortera hela fältet använder man kommandot `samsortering(A, 1, length(A))`. Om $v < h$, eller med andra ord om intervallet innehåller minst två element, så fortsätter funktionen att dela intervallet på mitten, sortera dessa delar av fältet och till slut sammanfoga de sorterade delfälten till ett enda sorterat fält.

Anmärkning 7. Ibland har man behov av att implementera en funktion med valbara parametrar. I Python är det möjligt att genom att i funktionens argument definiera ett initialvärde för en parameter. Det gör att parametern antar det värdet ifall inget värde anges då funktionen kallas. Det här betyder att man i Python kunde definiera funktionen *samsortering(A,v,h)* så att om v och h inte anges, så sorteras hela fältet A.

```
1  def samsortering(A, v=None, h=None):
2      if v==None:
3          v = 0
4      if h==None:
5          h = len(A)-1
6      if v < h:
7          m = (v+h) // 2
8          samsortering(A, v, m)
9          samsortering(A, m+1, h)
10         sammanfoga(A, v, m, h)
```

Idén bakom samsorteringen är alltså väldigt enkel. Det svåra är att implementera funktionen som sammanfogar två sorterade fält till ett enda sorterat fält på samma sätt som i Tabell 5.5.

```
sammanfoga(A, v, m, h):
    V = A[v:m]
    H = A[m+1:h]
    V[m-v+1] = inf
    H[h-m] = inf
    i = 1
    j = 1
    for k = v to h:
        if V[i] < H[j]:
            A[k] = V[i]
            i = i + 1
        else:
            A[k] = H[j]
            j = i + 1
```

Med initieringen av V och H på andra och tredje raden avses att de är kopior av delfält av A. Efter intieringen består V av $m-v$ element och H av $h-m-1$ element. Syftet med att sedan till V och H lägga till elementet *inf* som sista element, är att man kan jämföra vilket som helst element med dessa

och alltid erhålla att *inf* är större. Till följande vill man nämligen jämföra element parvis i V och H för att avgöra i vilken ordning de ska placeras tillbaka i A. Indexen i och j håller reda på vilka element som ska jämföras.

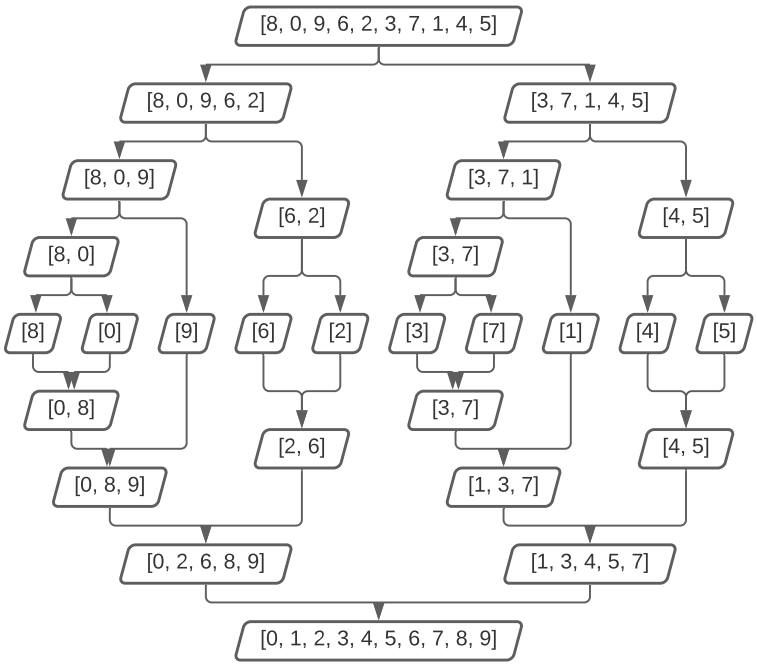
Sats 5.13. *Tidskomplexiteten för samsortering är $O(n\log(n))$, där n är antalet element i fältet som ska sorteras.*

V					**H**				**A**						
38	65	77	81	∞	40	42	78	∞	81	38	65	77	42	78	40
<u>38</u>	65	77	81	∞	<u>40</u>	42	78	∞	38	38	65	77	42	78	40
38	<u>65</u>	77	81	∞	<u>40</u>	42	78	∞	38	40	65	77	42	78	40
38	<u>65</u>	77	81	∞	40	<u>42</u>	78	∞	38	40	42	77	42	78	40
38	<u>65</u>	77	81	∞	40	42	<u>78</u>	∞	38	40	42	65	42	78	40
38	65	<u>77</u>	81	∞	40	42	<u>78</u>	∞	38	40	42	65	77	78	40
38	65	77	<u>81</u>	∞	40	42	<u>78</u>	∞	38	40	42	65	77	78	40
38	65	77	<u>81</u>	∞	40	42	78	<u>∞</u>	38	40	42	65	77	78	81

Tabell 5.7: Tabellen illusterar hur de sorterade fälten V och H sammanfogas så att A till slut är sorterad. De understreckade elementen indikerar vilka element i V och H som är i tur att jämföras, för att sedan kopieras till rätt plats i A.

5.5 Högsortering

I kapitel 4 Datastrukturer definierades den binära högen och ett antal operationer på binära högar. Operationen *skapaHög(A)* konstruerar en så kallad max-hög på linjär tid. Med hjälp av den samt operationen *högifiera(A,i)* är det enkelt att sortera ett fält. Om fältet A är en max-hög, så är fältets största element $A[1]$. Genom att sätta det elementet sist i fältet, och genom att heapifiera den kvarstående heapen vid indexet 1 kommer man åt följande största element. För att lyckas heapifiera fältet utan att flytta på de största elementen som är sist i tabellen behöver vi istället för längden av hela fältet A använda oss av längden av *högen A* som vi kan kalla *höglängd*. Vi lägger till detta till funktionen *högifiera*.



Figur 5.1: Grafen illustrerar hur det osorterade fältet $[8, 0, 9, 6, 2, 3, 7, 1, 4, 5]$ rekursivt spjälks upp i mindre fält och sedan sammanfogas till ett sorterat fält.

```
högifiera(A, i, höglängd):
    vänster = vänsterBarn(i)
    höger = högerBarn(i)
    störst = i
    if vänster <= höglängd and A[vänster] > A[störst]:
        störst = vänster
    if höger <= höglängd and A[höger] > A[störst]:
        störst = höger
    if störst != i:
        A[i], A[störst] = A[störst], A[i]
        högifiera(A, störst, höglängd)
```

Nu kan vi formulera det som kallas för *högsorteringsalgoritmen.*

```
högsortering(A):
```

```
skapaHög(A)
n = length(A)
höglängd = length(A)
for i = n downto 2:
    bytPlats(A, 1, i)
    höglängd = höglängd - 1
    högifiera(A, 1, höglängd)
```

Sats 5.14. *Tidskomplexiteten för högsortering är $O(n\log(n))$, där n är antalet element i fältet som ska sorteras.*

Bevis. Till först byggs max-högen på tiden $O(n)$, så det är for-slingan som är relevant. Eftersom den itererar över hela listan (förutom första elementet), så bidrar slingan med en faktor på n. Den mest tidskrävande operationen innanför slingan är `högifiera`, som utförs på logaritmisk tid $O(\log(n))$. Därmed är körtiden för hela sorteringsalgoritmen $O(n\log(n))$. □

5.6 Kvicksortering

Kvicksortering (quicksort) är en sorteringsalgoritm som i värsta fall har en komplexitet som är $O(n^2)$. I praktiken däremot, då elementen är slumpmässigt ordnade, uppnår algoritmen komplexiteten $O(n\log(n))$ [3]. Algoritmen använder den tidigare nämnda ”söndra och härska”-tekniken för att sortera ett delfält $A[p, r]$ på följande sätt.

1. **Söndra.** Ordna om elementen i $A[p, r]$ och beräkna ett index q, så att varje element i $A[p : q-1]$ är mindre än eller lika med varje element i $A[q+1 : r]$.

2. **Härska.** Fortsätt med hjälp av rekursion att dela in $A[p : q-1]$ och $A[q+1 : r]$ på samma sätt som i föregående steg.

3. **Sammanfoga.** Då delfälten $A[p : q-1]$ och $A[q+1 : r]$ endast innehåller ett element, så är hela fältet $A[p, r]$ sorterat.

Till följande antar vi att funktionen $delaUpp(A, p, r)$ utför uppdelningen av $A[p, r]$ enligt beskrivningen i söndra-steget, och att den dessutom retur-

nerar indexet q. Då kan vi beskriva kvicksorteringsalgoritmen på följande sätt.

```
kvicksortering(A, p, r):
    if p < r:
        q = delaUpp(A, p, r)
        kvicksortering(A, p, q-1)
        kvicksortering(A, q+1, r)
```

Hela fältet A kan då sorteras genom att välja parametrarna $p = 1$ och $r = length(A)$. Det komplicerade med algoritmen är att lyckas koda funktionen $delaUpp(A, p, r)$ så att den fungerar på önskat vis. Det finns flera olika sätt att göra det här på. Ett sätt är att först välja delfältets sista element $x = A[r]$ som en så kallad *pivå* (pivot). Sedan ordnas $A[p, r]$ om så att pivåelementet x placeras på rätt plats. Med andra ord flyttas alla element vars värde högst är x till vänster om pivån, och alla element större än x flyttas till höger om pivån. För att förverkliga det här traverserar man delfältet $A[p, r-1]$ och håller reda på två olika index i och $j \geq i$ som delar in $A[p, r-1]$ i tre olika delar med följande egenskaper. $A[p : i]$ innehåller endast element som högst har värdet x, $A[i+1 : j-1]$ innehåller endast element större än x, och $A[j, r-1]$ innehåller ogranskade element. I ett visst skede kunde delfältet $A[p, r]$ således se ut så här.

$$\ldots \overbrace{\overset{p}{3}\ 2\ 3\ \overset{i}{5}}^{\leq x}\ \overbrace{9\ 8\ 8\ 6}^{>x}\ \overbrace{\overset{j}{3}\ 5\ 2\ 7\ 8}^{\text{osorterade}}\ \overbrace{\overset{r}{5}}^{x}\ \ldots$$

Man traverserar $A[p, r-1]$ med j som variabel. Varje gång $A[j] \leq x$ så ska elementet $A[j]$ flyttas till posititionen $i+1$ eftersom $A[i+1] > x$. Vidare måste i öka med 1 för att upprätthålla kravet på delfältet $A[p : i]$.

Då man granskat alla element i $A[p, r-1]$, så bör pivåelementet x ännu flyttas till sin rätta position. Den positionen är $i+1$, eftersom $A[i+1]$ är det element längst till vänster som är större än x. Vi kan nu beskriva funktionen $delaUpp(A, p, r)$ mera komprimerat som pseudokod.

```
delaUpp(A, p, r):
    x = A[r]
```

```
    i = p - 1
    for j = p to r-1:
        if A[j] <= x:
            i = i + 1
            bytPlats(A, i, j)
    bytPlats(A, i+1, r)
    return i
```

Ett annat sätt att beskriva uppdelningsfunktionen är genom att använda den så kallade Hoare-uppdelningen, efter C.A.R. Hoare. Uppdelningen ger samma resultat men den beskrivs på ett annat sätt. Med Hoare-uppdelningen låter man indexet i starta från vänster och indexet j från höger. Sedan ökar man i med 1, och minskar j med 1, tills $A[i] \geq x$ och $A[j] \leq x$. Då byter man plats på dessa element, och sedan fortsätter man på samma sätt tills i och j möts. Stället där indexena möts är sedan det index som ska returneras. En sådan uppdelningsfunktion kan vi beskriva på följande sätt. [3]

```
delaUpp(A, p, r):
    x = A[r]
    i = p - 1
    j = r + 1
    while True:
        while A[i] < x:
            i = i + 1
        while A[j] > x:
            j = j + 1
        if i < j:
            bytPlats(A, i, j)
        else:
            return j
```

Även om idén bakom kvicksorteringsalgoritmen är relativt enkel kan vi konstatera att implementeringen är väldigt svår då den innehåller många detaljer om indexen man bör hålla reda på. Därmed är programmeringen av kvicksorteringen inte så lämplig för studieavsnittet MAA11.

Sats 5.15. *Tidskomplexiteten för kvicksortering är i värsta fall $O(n^2)$, och den förväntade komplexiteten är $O(n\log(n))$, där n är antalet element i fältet som ska sorteras.*

I värsta fall är fältet i själva verket redan sorterat. Det här innebär att pivåelementet alltid är det största elementet, och efter varje uppdelning av $A[p,r]$ kommer de nya uppdelade fälten att vara $A[p,r-1]$ och $[A[r]]$. I det fallet, eller i liknande fall, kommer komplexiteten att vara $O(n^2)$.

Om vi däremot antar att elementen är slumpmässigt ordnade, kommer uppdelningsfunktionen $delaUpp(A,p,r)$ i de flesta fall dela upp $A[p,r]$ i två delfält av betydande storlek. Detta resulterar i att antalet uppdelningar är $\sim \log(n)$, och således uppnår algoritmen komplexiteten $O(n\log(n))$.

Anmärkning 8. Man kan ytterligare göra uppdelningarna mera effektiva genom att välja pivåelementet på ett annat sätt. Ett bättre val är att välja tre element, exempelvis det första, mittersta och sista elementet i $A[p,r]$, och som pivåelement välja det element vars värde ligger mellan de två andra. På så sätt förhindrar man till en del att pivåelementet väljs till ett väldigt stort eller litet tal.

Övning 5.16. Rita en tabell som illustrerar hur kvicksorteringsalgoritmen opererar då inputfältet är $[8,0,9,6,2,3,7,1,4,5]$.

5.7 Facksortering

Sorteringsalgoritmerna som hittills presenterats är jämförelsebaserade algoritmer. Det innebär att sorteringen bygger på att parvis jämföra element. Vi konstaterade tidigare att den bästa möjliga komplexiteten för en jämförelsebaserad sorteringsalgoritm är $O(n\log(n))$, då elementen som ska sorteras är godtyckliga reella tal. Nu introducerar vi två sorteringsalgoritmer som med hjälp av antaganden om mängden som ska sorteras, uppnår tidskomplexiteten $O(n)$.

Den första är *facksortering* (*bucket sort*). Nu antar vi att värdena i fältet som ska sorteras är jämnt fördelade över intervallet $[0,1)$. Anta att fältet innehåller n element. Facksorteringsalgoritmen går ut på att dela in intervallet $[0,1)$ i n stycken lika stora delintervall som vi kan kalla ”fack”, placera varje

element i rätt fack, sortera varje fack separat, och sedan sammanfoga facken till ett sorterat fält. Eftersom elementen är jämt fördelade över intervallet $[0, 1)$ så kommer varje fack troligtvis innehålla ett litet antal element som snabbt kan ordnas exempelvis med insättningssortering.

Låt oss beskriva algoritmen mera entydigt som pseudokod. För det behöver vi använda en datastruktur vars element är de olika facken, där varje fack dessutom är en datastruktur i vilket man bör kunna lagra elementen. Det här kan vi förverkliga genom att använda ett 2-dimensionellt dynamiskt fält. Ett sådant fält kan tänkas som ett ”fält av fält”. I koden nedanför skapar och returnerar kommandot `B = emptyArray(2)` ett sådant fält. Med `sammanfoga(B)` menas att fälten (facken) som B består av sammanfogas till ett enda fält av tal så att ordningen bevaras.

```
facksortering(A):
    B = emptyArray(2)
    n = length(A)
    for i = 1 to n:
        x = floor(n * A[i])
        B[x].append(A[i])
    for i = 1 to n:
        insättningssortering(B[i])
    sammanfoga(B)
    for i = 1 to n:
        A[i] = B[i]
```

Idén som facksorteringen bygger på är väldigt enkel. Däremot stöter man snabbt på utmaningar vid själva kodningen av facksorteringsalgoritmen om man saknar erfarenhet av programmering. Vad som gör kodningen svår är behovet av fältet vars element också är fält, eller en annan teknik som möjliggör implementeringen. Det här kan vara ett utmanande koncept för många studerande.

Sats 5.17. *Tidskomplexiteten för facksortering är $O(n)$, där n är antalet element i fältet som ska sorteras.*

Bevis. Koden innehåller tre for-slingor och funktionen *sammanfoga* som bör beaktas för att bevisa påståendet. Den första slingan placerar varje element

i rätt fack. Placeringen av ett element kräver konstant körtid $O(1)$, så hela slingan bidrar med komplexiteten $O(n)$.

Den andra slingan som sorterar varje fack använder här insättningssortering. Vi känner till att detta är en långsam algoritm, men eftersom vi kan anta att varje fack innehåller ett litet antal element, så kommer körtiden för sorteringen av varje fack inte bero på storleken n. Således bidrar den andra slingan också med komplexiteten $O(n)$.

Kommandot som sammanfogar B kan implementeras på olika sätt. Antalet fält som sammanfogas är n, och varje fält innehåller ett litet antal element, så funktionen går att implementera så att dess komplexitet är $O(n)$.

Slutligen ser vi att den tredje och sista slingan som kopierar innehållet från B till A endast innehåller den elementära operationen `A[i] = B[i]`, och därmed är även körtiden för denna slinga $O(n)$.

Algoritmen består alltså av fyra successiva procedurer med komplexiteten $O(n)$. Således är facksorteringsalgoritmens komplexitet $O(n + n + n + n) = O(n)$, där n är antalet element som ska sorteras. □

Övning 5.18. Implementera i Python en funktion *sammanfoga(A)*, vars input A är en lista av listor av tal, och vars output är en sammanfogad lista av talen i listorna som A består av.

Exempel på hur funktionen opererar:

```
1 A = [[91, 55, 20, 92], [68], [15, 89]]
2 print(sammanfoga(A))
```

```
[91, 55, 20, 92, 68, 15, 89]
```

Övning 5.19. Skriv om facksorteringsalgoritmens pseudokod så att den fungerar för ett fält av tal som är jämnt distribuerade över intervallet $[a, b)$, istället för $[0, 1)$. Tips: Det enda som bör ändras i koden är beräkningen av facket i vilket elementet $A[i]$ ska placeras.

5.8 Uppräkningssortering

Till följande antar vi att mängden som ska sorteras är ett fält A av heltal i intervallet $[1, m]$, för något angivet $m \in \mathbb{Z}$. *Uppräkningssortering* (*counting*

sort) går ut på att med hjälp av det här antagandet placera varje element $A[i]$ på rätt plats genom att räkna antalet element i A som är mindre än $A[i]$. Om vi exempelvis märker att 14 element i inputfältet är mindre än $A[i]$, så ska elementet $A[i]$ vara på position 15 i det sorterade fältet. Detta skulle fungera om fältet endast består av unika element, så vid implementeringen måste vi beakta detta. Istället kommer vi att räkna antalet element som är mindre än *eller lika med* ett visst element.

För att kunna beskriva uppräkningssorteringsalgoritmen bör vi alltså hitta ett sätt att för varje heltal $k \in \{1, 2, \ldots, m\}$ som kan förekomma i inputen beräkna hur många element som är mindre än eller lika med k. Det här problemet är enklare att lösa om vi först beräknar antalet element som är lika med k, för varje $x \in \{1, 2, \ldots, m\}$. För att hålla reda på dessa antal kan vi skapa ett fält $C = [\underbrace{0, 0, \ldots, 0}_{m \text{ st}}]$ av m stycken nollor. Om vi sedan traverserar A, och för varje element $A[i]$ ökar värdet $C[A[i]]$ med 1, så kommer $C[k]$ slutligen att ange antalet gånger elementet k förekommer i A.

Vidare genom att sätta $C[i] = C[i] + C[i-1]$, då i löper från 2 till m i stigande ordning, så kommer $C[k]$ slutligen att ange antalet element i A som är mindre än eller lika med k.

Nu kvarstår endast att med hjälp av C som innehåller information om de nya positionerna placera rätt element på rätt position. Under denna procedur kan vi inte ändra på elementen i intputfältet A, så vi behöver ett till fält B där vi tillfälligt lagrar de sorterade elementen. Vi måste dessutom beakta att element kan förekomma flera gånger. Då man placerar ett element k på rätt position i B, så kommer antalet osorterade element som är mindre än eller lika med k att minska med ett. Vi kan således lösa det problemet genom att sätta $C[k] = C[k]-1$ då elementet k placeras i B (se Tabell 5.8). Låt oss nu på basis av ovanstående resonemang formulera uppräkningssorteringsalgoritmen som pseudokod.

```
uppräkningssortering(A, m):
    for i = 1 to m:
        C[i] = 0
    for i = 1 to length(A):
        C[A[i]] = C[A[i]] + 1
    for i = 2 to m:
        C[i] = C[i] + C[i-1]
    for i = 1 to length(A):
        k = A[i]
        B[C[k]] = k
        C[k] = C[k] - 1
    for i = 1 to length(A):
        A[i] = B[i]
```

A[i]	**B**							**C**				
								2	3	3	6	7
2			2					2	**2**	3	6	7
4			2			4		2	2	3	**5**	7
1		1	2			4		**1**	2	3	5	7
1	1	1	2			4		**0**	2	3	5	7
5	1	1	2			4	5	0	2	3	5	**6**
4	1	1	2		4	4	5	0	2	3	**4**	6
4	1	1	2	4	4	4	5	0	2	3	**3**	6
								1	*2*	*3*	*4*	*5*

Tabell 5.8: Exempel över hur uppräkningssortering opererar då inputen är $A = [2, 4, 1, 1, 5, 4, 4]$. Elementet $A[i]$ som placeras i B är understrykt. Värdet av elementet i C skrivet i fet stil minskar med 1 under samma iteration. Slutligen innehåller B det sorterade fältet.

För att bestämma algoritmens komplexitet bör man undersöka de fem for-slingorna och avgöra vilken eller vilka som är mest tidskrävande. Alla operationer innanför slingorna är elementära operationer, så endast antalet iterationer är relevant. Den första och tredje slingan bidrar således med komplexiteten $O(m)$. Låt n vara antalet element i inputfältet, det vill säga

$n = length(A)$. Då bidrar den andra, fjärde och femte slingan med komplexiteten $O(n)$.

Sats 5.20. *Tidskomplexiteten för uppräkningssortering är $O(n + m)$, där n är antalet element i fältet som ska sorteras, och fältet består av heltal i intervallet $[1, m]$.*

Anmärkning 9. I praktiken används uppräkningssortering oftast då m och n är av samma storleksordning [3], och då är dess tidskomplexitet $O(n + n) = O(n)$.

Uppräkningssorteringsalgoritmen som helhet är ganska komplex, då den består av flera små komponenter som alla måste pusslas ihop på rätt sätt. Däremot är de enskilda stegen i algoritmen tydliga och konkreta, och kunde eventuellt lämpa sig som övningsuppgifter i MAA11.

Övning 5.21. Skapa en funktion *räknare(A)* vars input är ett fält A med heltal $0 \leq n \leq 99$, och vars output är ett fält C, så att $C[k]$ anger antalet gånger elementet k förekommer i A.

5.9 Sortering i Python

I praktiken då man behöver sortera en mängd implementerar man inte sin egen sorteringsalgoritm, utan man använder istället programspråkets inbyggda funktioner för sortering. Pythons standardbibliotek innehåller ett par olika möjligheter för att sortera bland annat listor.

Funktionen *sorted(A)* skapar och returnerar en ny sorterad lista. Den ändrar alltså inte på objektet A.

```
1 A = [3,42,0,3,4]
2 print(sorted(A))
3 print(A)
```

```
[0, 3, 3, 4, 42]
[3,42,0,3,4]
```

Om man vill sortera en lista på plats, det vill säga genom att manipulera själva `list`-objektet, kan man istället använda `list`-metoden *sort()*.

```
1 import random
2 A = list(range(10))
3 random.shuffle(A)
4 print(A)
5 A.sort()
6 print(A)
```

```
[7, 1, 0, 3, 2, 9, 4, 8, 6, 5]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Funktionerna fungerar även direkt på listor av strängar, och sorterar då strängarna i alfabetisk ordning.

```
1 namnlista = ['Charlie', 'Alice', 'Aaron', 'Bob']
2 print(sorted(namnlista))
```

```
['Aaron', 'Alice', 'Bob', 'Charlie']
```

De inbyggda funktionerna använder sig av *Timsort*-algoritmen, som Tim Peters designade för Python 2002. Idén bakom Timsort är att hitta redan monotona intervall av fältet som ska sorteras, och sedan sammanfoga intervallen [9].

Övning 5.22. Skapa en funktion *sök(A,x)* som returnerar *true* om fältet A innehåller elementet x, och annars *false*. Du får anta att A är ett sorterat fält, och funktionens komplexitet bör vara $O(\log(n))$.

Övning 5.23. Anta att *namnlista* är en lista av strängar i formen *"Förnamn Efternamn"*. Skapa en funktion *ordnaEnligtEfternamn* som returnerar listan så att namnen är i alfabetisk ordning enligt efternamnen.

Sammanfattning

Vi har konstaterat att sorteringsalgoritmerna är en viktig del då det gäller lärandet av olika algoritmer. Olika sorteringsalgoritmer använder sig av olika viktiga koncept för programmeringen. Sådana är bland annat datastrukturer som fält och träd, samt programmeringstekniker med slingor, konditionsvillkor och rekursion. Å andra sidan är det klart att det uppstår många utmaningar då man sätter sig in i de specifika sorteringsalgoritmerna. Även en del av de enklaste sorteringsalgoritmerna innehåller svåra moment.

En annan utmaning är förstås att studieavsnittet MAA11 innehåller mycket annat än sorteringsalgoritmer, och undervisningstiden är väldigt begränsad. Det är då svårt att gå in på djupet inom ett visst område.

Då läroplanen är ny finns det för tillfället inte så mycket forskning och material om det här ämnet. Enligt mig saknas det alltså praktiska och konkreta lösningar för att hjälpa matematiklärare att planera undervisningen för studieavsnittet MAA11. Man kunde således forska mera i hur gymnasister lär sig programmering. Samarbete med högskolor skulle också behövas för att kunna avgöra vilka behov som finns.

Litteraturförteckning

[1] Jane Koivisto. Algoritmisen ajattelun kehittäminen ohjelmoinnin ja kielentämisen avulla matematiikan opetuksessa. Master's thesis, 2019.

[2] Utbildningsstyrelsen. Grunderna för gymnasiets läroplan 2019. 2019.

[3] TH Cormen, CE Leiserson, RL RIVEST, and C Stein. Introduction to algorithms third edition ed. *Massachusetts London, England*, 2009.

[4] Jan Cuny, Larry Snyder, and Jeannette M Wing. Demystifying computational thinking for non-computer scientists. *Unpublished manuscript in progress, referenced in http://www. cs. cmu. edu/˜ CompThink/resources/TheLinkWing. pdf*, 2010.

[5] Leila Ribeiro, Daltro José Nunes, Marcia Kniphoff da Cruz, and Ecivaldo de Souza Matos. Computational thinking: Possibilities and challenges. In *2013 2nd Workshop-School on Theoretical Computer Science*, pages 22–25. IEEE, 2013.

[6] Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[7] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. Why complicate things? introducing programming in high school using python. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 71–80, 2006.

[8] Lars-Erik Janlert and Torbjörn Wiberg. *Datatyper och algoritmer*. Studentlitteratur, 2000.

[9] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to timsort. *URL https://hal-upec-upem. archives-ouvertes. fr/hal-01212839, working paper or preprint*, 2015.