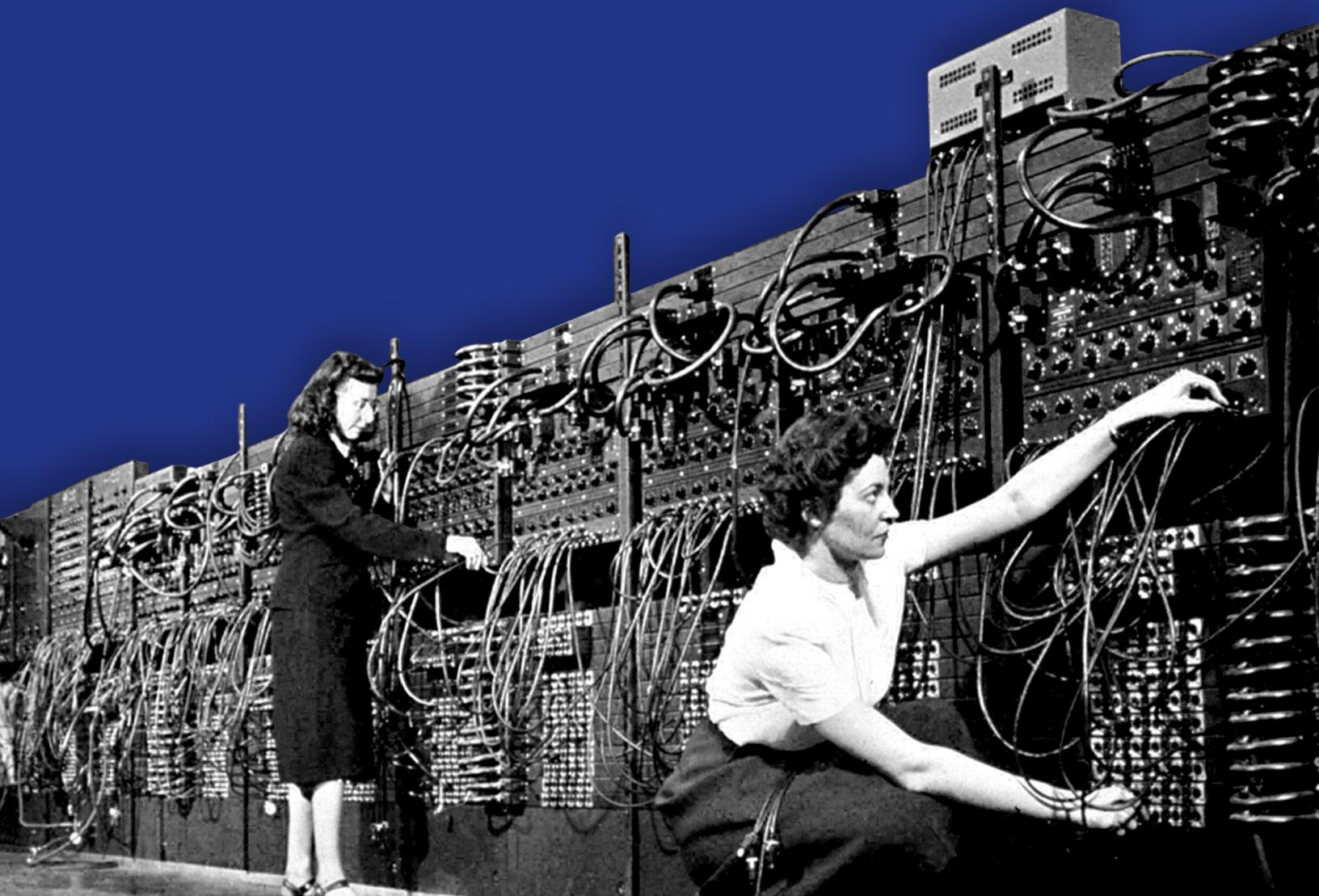




ANALYS AV
SORTERINGSALGORITMER
MED C#

ERIK V. NORBERG



Erik Viking Norberg 2019
erik.norberg@europaskolan.com
2020-20-20

"Premature optimization is the root of all evil."
– D.E. Knuth, The Art of Computer Programming, Volume 1

"In some ways, programming is like painting. You start with a blank canvas and certain basic raw materials. You use a combination of science, art, and craft to determine what to do with them."

– Andrew Hunt, The Pragmatic Programmer: From Journeyman to Master

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Innehåll

| | |
|---|-----------|
| 1 Inledning | 3 |
| 1.1 Introduktion | 3 |
| 1.2 Syfte och metod | 4 |
| 1.3 Bakgrund | 4 |
| 1.3.1 Ordo | 5 |
| 1.3.2 Fundamental C# | 6 |
| 1.3.3 Grundläggande algoritmer | 8 |
| 1.3.4 Divide-and-conquer algoritmer (#2) | 14 |
| 1.3.5 'Jämförelse på plats' algoritmer (#3) | 17 |
| 1.3.6 Övriga algoritmer (#4) | 17 |
| 2 Metod | 18 |
| 3 Resultat | 19 |
| 4 Diskussion och slutsatser | 20 |
| 5 Källförteckning | 21 |

Kapitel 1

Inledning

1.1 Introduktion

Du ska till ett bibliotek och hämta en bok. När du kommer fram till biblioteket börjar du kolla efter den. Men inga av böckerna verkar stå i någon ordning. Då inser du att alla böckerna i biblioteket inte är sorterade. Det är huller om buller. Du frågar då bibliotekarien varför det inte är sorterade. Hon svarar att det tar för lång tid för henne att sortera alla böcker eftersom det har så många, så hon har inte gjort det. Du, med ditt genialiska intellekt, kommer snabbt på att man kan skriva ett dataprogram som sorterar all böcker snabbare än vad någon mänskliga kan göra. Du utbrister då heureka! Sedan springer du hem och sätter dig framför din dator för att skriva dataprogrammet. Men du inser snabbt att det finns många, många, väldigt många olika sätt att sortera böckerna. Du kommer på ett flertal sätt man kan göra det men du vet inte vilken metod du ska använda. Du ville ju att programmet skulle vara snabbt så du bestämmer dig för att skriva alla metoder du har kommit på för att kunna se vilken som är snabbast. Detta är vad mitt gymnasiearbete kommer att handla om. Vilken metod (eller sorteringsalgoritm som det kallas) är mest effektiv när det kommer till sorteringsalgoritm.

Inom datavetenskap är en sorteringsalgoritm en algoritm som sorterar element i en lista till en viss ordning. De mest vanliga ordningarna att sortera till är numerisk ordning eller lexikografisk ordning. Effektiv sorteringsalgoritm är viktigt för att optimera effektiviteten av andra algoritmer (till exempel sök- och sammanslagningsalgoritmer) som kräver att dess givna data är i sorterad ordning. Sortering är också ofta användbart för kanonisering av data och för att producera mänsklig läsbar data. Mer formellt måste den sorterade datan från en sorteringsalgoritm uppfylla två villkor:[13, 6]

1. Den sorterade datan är i icke minskande ordning (varje element är inte mindre än föregående element efter den önskade ordningen);
2. Resultatet är en permutation (en omordning som behåller alla original element) av den givna datan.

Vidare lagras ingångsdata ofta i en matris, vilket tillåter slumpmässig åtkomst av dess element, snarare än en lista, som endast tillåter sekventiell åtkomst; även om många algoritmer kan tillämpas på vilken typ av data som helst efter lämplig modifiering.

Redan från början av datavetenskapen har sorteringsalgoritmer varit mycket forskningsämnen, kanske på grund av att det är ett problem som är lätt att konceptuellt förstå men är väldigt svårt att lösa. Blandet det första personerna som skrev sorteringsalgoritmer är Betty Holberton som skrev ENIAC och UNIVAC. En av det mest kända sorteringsalgoritmerna ”*Bubble sort*” skrevs redan så tidigt som 1956. Detta innebär att ämnet redan är väldigt väl undersökt. Så jag kommer inte kunna skriva någon ny sorteringsalgoritm men jag kommer kunna analysera det som redan finns. [13] Mer exakt kommer jag analysera: *Selection sort*, *Shell-sort*, *Insertion Sort*, *Merge Sort*, *Quick Sort*, *Heap Sort*, *Bubble Sort*, *Comb sort* och *Bidirectional Bubble Sort*.

1.2 Syfte och metod

Syftet med arbetet är att bestämma den snabbaste och mest effektiva sorteringsalgoritmen. Detta kommer redovisat genom att först visa hur man kan skriva sorteringsalgoritmerna i programmeringsspråket C#. Det är viktigt att man förstår hur syntaxen i C# fungerar så det kommer läggas fokus på hur den fungerar och vad den gör. Om man inte förstår koden är det svårt för en att förstå resten av arbetet eftersom C# är grunden för arbetet. Följt av detta kommer matematiken bakom algoritmernas ordo (se sida 5) förklaras så att man från ett matematiskt perspektiv kan jämföra algoritmerna.

Sorteringsalgoritmer i sin helhet öppnar upp många intressanta dörrar inom datavetenskapen. Men det är ett tråsk av information och data som är omöjlig att utforska på en och samma gång. För att undvika det tråsket kommer jag begränsa arbetet till enbart så kallade ”*Comparison sorting algorithms*” och inte ”*Non-Comparison sorting algorithms*” (eller andra former av sorteringsalgoritmer).

Sorteringsalgoritmer handlar inte bara om optimering utan också och om minneshantering inom RAM minnet. Särskilt dom sorteringsalgoritmer som är *på plats*. Det innebär att det bara behöver $\mathcal{O}(1)$ eller $\mathcal{O}(\log n)$ minne (beteckningen \mathcal{O} förklaras mer på sida 5) utöver det element som sorteras och det behöver inte skapa temporära platser att spara datan i RAM minnet som andra algoritmer behöver göra. [2] Dock så är frågan om minne inte viktigt för arbetet så jag kommer undvika frågan om minne.

1.3 Bakgrund

För att man ska kunna analysera algoritmerna måste man först förstå ett par fundamentala saker. Ordo, så att man kan jämföra dom matematiskt, C# syntax, så man förstår implementationerna av algoritmerna, och visa termer som kommer uppstå ett flertal gånger i arbetet.

1.3.1 Ordo

Ordo (*Ordning* på latin. Utalat 'or do Ω) är ett begrepp inom matematiken och även datavetenskapen som används för att ge ett mått på hur beräkningsmässigt tiden en funktion är. Detta betecknas med ett versalt \mathcal{O} . När en funktion beskrivs med ordo så beskrivs vanligtvis en övre gräns för en funktions tillväxthastighet. Det vill säga hur snabbt y ökar med ett ökande x . Till exempel betecknar $\mathcal{O}(x^2)$ och $\mathcal{O}(\sqrt{x})$ något som växer lika fort som x^2 respektive \sqrt{x} då x ökar. Detta används särskilt inom datavetenskapen när man beskriver hastigheten av olika algoritmer, som sorteringsalgoritmer. [15, 7] Associerat med \mathcal{O} är ett par andra notationer som använder symbolerna o , Ω , ω och Θ som beskriver andra gränser av asymptotisk tillväxt. Dessa kommer vi gå igenom mer i detalj snart.

En mer formell definitionen av ordo lyder: Låt f vara en reell eller komplex funktion och g vara en reell funktion. Låt båda funktionerna vara definierade med någon begränsad delmängd av det positiva reella talen och $g(x)$ vara strikt positivt för stora x . Man skriver [12]

$$f(x) = \mathcal{O}(g(x)) \quad \text{när } x \rightarrow \infty$$

om absolutbeloppet av $f(x)$ är en positiv konstant multiple av $g(x)$ för alla stora värden av x . Alltså är $f(x) = \mathcal{O}(g(x))$ om det existerar ett positivt reellt nummer M och reellt nummer x_0 så att

$$|f(x)| \leq Mg(x) \quad \text{för alla } x \leq x_0.$$

I många sammanhang lämnas antagandet att vi är intresserade av tillväxthastigheten när variabeln x går mot oändlighet, då skriver man mer enkelt

$$f(x) = \mathcal{O}(g(x)).$$

Ordo kan även användas för att beskriva betendet av f nära ett reelt nummer a (ofta $a = 0$). Man skriver

$$f(x) = \mathcal{O}(g(x)) \quad \text{när } x \rightarrow a$$

om det existerar ett positivt nummer δ och M sådan att för alla x med $0 < |x - a| < \delta$ så är

$$|f(x)| \leq Mg(x).$$

Eftersom $g(x)$ är valt att vara noll för värden på x tillräckligt nära a , kan båda dessa definitioner förenas med användning av *limit superior*:

$$f(x) = \mathcal{O}(g(x)) \quad \text{när } x \rightarrow a$$

om

$$\limsup_{x \rightarrow a} \frac{|f(x)|}{g(x)} < \infty.$$

Det tidigare nämnda notationerna o , Ω , ω och Θ kan man lätt förstå i tabellen under då det inte krävs några formella bevis för att förklara dem, eftersom det

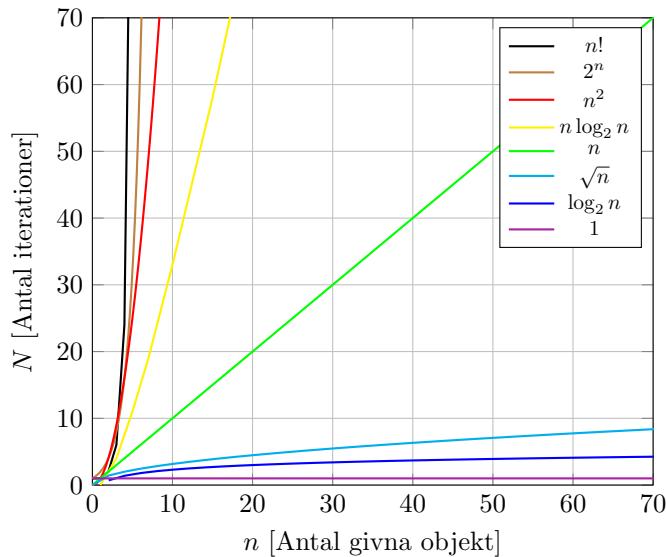
| Notation | I ord |
|------------------------------|-------------------------------------|
| $f(n) \in \mathcal{O}(g(n))$ | f växer högst lika snabbt som g |
| $f(n) \in \Omega(g(n))$ | f växer minst lika snabbt som g |
| $f(n) \in \Theta(g(n))$ | f växer lika snabbt som g |
| $f(n) \in o(g(n))$ | f växer längsammare än g |
| $f(n) \in \omega(g(n))$ | f växer snabbare än g |

Tabell 1.1: Ordo och relaterade notationer

är så lika \mathcal{O} .

Som nämnt innan beskriver ordo hur beräkningsmässigt tung en funktionär. Detta är väldigt viktigt när det kommer till analys och optimering av olika algoritmer, främst när det ska jämföras med varandra. Om vi har funktionen $f(x) = 6x^4 - 2x^3 + 5$ och vi vill förenkla den med ordo så vi kan se hur den växer när x går mot oändligheten så börjar vi med att kolla på termerna av f , nämligen $6x^4$, $-2x^3$ och 5 . Utav dessa tre termer har termen med den största exponenten med avseende på x snabbast tillväxthastighet, nämligen $6x^4$. $6x^4$ är en produkt av 6 och x^4 där första termen inte är beroende av x . Att utelämna denna term resulterar i en förenklad formen x^4 . Således säger vi att $f(x) = \mathcal{O}(x^4)$.

Det finns några par tillväxthastigheter som man stöter på oftast när man talar om ordo. Det är dessa följande:



1.3.2 Fundamental C#

C# (C-Sharp) är ett programmeringsspråk utvecklat av Microsoft som körs på .NET Framework [11]. C# är i första hand ett objektorienterat programspråk.

Även kallad ett OOP (från engelskans *Object Oriented Programming*). Det är varför detta arbete skrivs i C#. Då alla sorteringsalgoritmer kan skrivas som var sitt objekt. Ett objekt är en instans av samlad syntax som används för att utföra en given operation. Exempelvis kan ett objekt kallas på för att göra en viss beräkning eller för att sammanställa data. En objekt i C# kan se ut såhär:

```
1 public void multiplyByTwo(int n)
2 {
3     return n * 2;
4 }
```

Denna funktion tar in ett heltal, `int n`, och ger tillbaka ett heltal dubbelt så stort som `n`, `return n * 2`.

En funktion i C# måste inte bara behandla aritmetik. En funktion kan även behandla arrays (listor) vilket är väldigt användbart när man vill sortera. Exempelvis om vi har listan

```
int[] lista = {1, 2, 5, 3, 4};
```

och vi vill byta plats på fyran och femman för att få listan i storleksordning kan vi skriva en funktion som denna:

```
1 static void swap(int x, int y, int[] arr)
2 {
3     int temp = arr[x];
4     arr[x] = arr[y];
5     arr[y] = temp;
6 }
```

Den tar in två positioner `x` och `y` i en given lista `arr` och byter plats på dom. Den gör detta genom att spara heltalet på position `x` i `arr` inuti en temporär variabel `temp`. När `x` är sparad i `temp` överskider funktionen `x` med `y`. Sist överskider funktionen `y` med `temp`. Då har det två positionerna byt plats. Vi kallar på funktionen för att byta plats på fyran och trean i `lista`:

```
swap(3, 5, lista);
```

och vi får att listan nu är

```
[1, 2, 3, 4, 5]
```

Observera att `swap` inte har en `return`, såsom `multiplyByTwo` har på rad 3. Detta är då för att `swap` inte ger tillbaka en ny lista. Den ändrar den givna arrayn utan att skapa en ny array medan `multiplyByTwo` tar in ett heltal och ger tillbaka ett nytt heltal. Observera även att när vi kallar på `swap` ger vi den 3 och 4 då det är positionerna av numrenna vi vill byta plats på, inte för att det är 3 och 4 vi vill byta plats på. Om vi istället ville byta plats på 1 och 3 hade vi skrivit

```
swap(1, 4, lista);
```

Vilket hade gett

```
[3, 2, 4, 1, 5]
```

Funktion `swap` kommer användas för varje sorteringsalgoritm som arbetet kommer analysera. Alltså varje gång en sorteringsalgoritm kollar på `swap` kommer den kalla koden som är beskriven ovan.

1.3.3 Grundläggande algoritmer

Sorteringsalgoritmer kan vara väldigt komplexa och effektiva med bekostnad av att vara väldigt svåra att förstå. Docks så kan sorteringsalgoritmer vara väldigt enkla men med bekostnad av att vara mindre effektiva. Ett par enkla sorteringsalgoritmer vi kommer kolla på nu är *Bubblesort*, *Cocktailsort* och, *Shellsort*. Vi börjar med Bubblesort.

Bubblesort är bland det minst effektiva sorteringsalgoritmerna men den är ändå en av det mest kända sorteringsalgoritmerna eftersom den är sorteringsalgoritmernas "Hello World". Den är väldigt lätt att förstå både praktiskt och teoretiskt. Därför används den mest som ett pedagogiskt verktyg [3]. Bubblesort fungerar genom att jämföra intilliggande element i en lista. Om elementen som jämförs ligger i fel ordning byter det plats så att det större elementet rör sig uppåt i listan. Denna process fortsätter tills det största elementet i listan är högst upp i listan. När detta är klart börjar algoritmen om men för det näst största elementet i listan. Detta repeteras till listan är sorterad. Låt oss säga att vi har en array

```
int[] lista = {5, 1, 4, 2, 8};
```

som ska sorteras från minst till störst med Bubblesort. Det kommer bara ta tre iterationer för bubblesort att sortera arrayen (elementen skrivna i fetstil jämförs);

Första iterationen:

$$\begin{aligned} [5, \mathbf{1}, 4, 2, 8] &\rightarrow [1, \mathbf{5}, 4, 2, 8] \\ [1, \mathbf{5}, 4, 2, 8] &\rightarrow [1, \mathbf{4}, \mathbf{5}, 2, 8] \\ [1, 4, \mathbf{5}, \mathbf{2}, 8] &\rightarrow [1, 4, \mathbf{2}, \mathbf{5}, 8] \\ [1, 4, 2, \mathbf{5}, 8] &\rightarrow [1, 4, 2, \mathbf{5}, 8] \end{aligned}$$

Andra iterationen:

$$\begin{aligned} [1, \mathbf{4}, 2, 5, 8] &\rightarrow [1, \mathbf{4}, 2, 5, 8] \\ [1, \mathbf{4}, \mathbf{2}, 5, 8] &\rightarrow [1, \mathbf{2}, \mathbf{4}, 5, 8] \\ [1, 2, \mathbf{4}, \mathbf{5}, 8] &\rightarrow [1, 2, \mathbf{4}, \mathbf{5}, 8] \end{aligned}$$

Vid detta laget är arrayen redan sorterad men det vet inte algoritmen. Den måste iterera över arrayen utan att behöva byta några element för att veta att den är klar.

Tredje iterationen:

$$\begin{aligned} [1, \mathbf{2}, 4, 5, 8] &\rightarrow [1, \mathbf{2}, 4, 5, 8] \\ [1, \mathbf{2}, \mathbf{4}, 5, 8] &\rightarrow [1, \mathbf{2}, \mathbf{4}, 5, 8] \end{aligned}$$

Denna algoritm är lätt att implementera i C# då den inte kräver mycket kod för att utföras:

```

1 static void bubbleSort(int[] array)
2 {
3     items = array.Length;
4     for (int i = 0; i < items; i++)
5     {
6         for (int j = 0; j < items - i - 1; j++)
7         {
8             if (array[j] > array[i])
9             {
10                 swap(i, j, array);
11             }
12         }
13     }
14 }
```

Vi initialiseras funktionen på rad 1. Vi kallar funktionen för `bubbleSort` och säger även att den bara kan ta emot en endimensionell array. Väl inuti funktionen säger vi att `items` ska vara lika med hur många element som finns i `array` när den är given. Detta är inte nödvändigt men det gör koden mer läsbar. Nu vill vi kunna iterera över `array` lika många gånger som det finns element i `array` men för varje iteration vill vi iterera över ett element mindre. Detta gör vi med `for` looparna på rad 4 och 6. Första `for` loopen på rad 4 loopar över alla element i `array` och på samma gång ökar värdet av `i` tills den är lika stor som `items`. Inuti den `for` loopen är en nested¹ `for` loop på rad 6 som också itererar över alla element i `array` förutom det `i` sista elementen. Med detta kan vi jämföra elementen på plats `i` och `j` för att se vilken som är störst. Om `j` är större än `i` bytter det plats, detta gör vi på rad 8 till 11.

Nu kan vi kalla på `bubbleSort` för att sortera `lista`:

```
bubbleSort(lista);
```

Vilket sorterar `lista` så att den nu är:

```
[1, 2, 4, 5, 8]
```

Analysen av Bubblesort är inte särskilt svår heller. Om vi har en array som är n element lång så kommer Bubblesort behöva jämföra $(n - 1)$ element första iteration. Andra iterationen kommer den behöva jämföra $(n - 2)$ element. Tredje Iterationen kommer den behöva jämföra $(n - 3)$ element och så vidare. Detta innebär att den totala mängden jämförelser bubblesort kommer behöva göra är

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \sum \frac{n(n - 1)}{2}$$

Vilket betyder att bubblesort har ett ordo av $\mathcal{O}(n^2)$.

Som sagt är Bubblesort inte särskilt effektiv. Man kan optimera koden så att den blir lite mer effektiv men det gör ingen jättestor skillnad. Man kan dock förbättra själva algoritmen. Första sättet man kanske kommer på för att

¹Det är en loop innuti en annan loop. Nästlade loop'ar är inte direkt ovanliga speciellt vid algoritmer som behandlar data. Exempel kan vara vid uppritning av kartor i grafiska spel eller sorterings av data.

förbättra algoritmen är genom att ändra hur den itererar genom en array. Istället för att varje iteration ska genomföras från vänster till höger en gång i en array innan nästa iteration kan varje iteration genomföras från vänster till höger och sedan tillbaka. Detta minskar hur många iterationer som krävs för att sortera en array eftersom den inte bara flyttar det största elementen till toppen av listan, den flyttar även samtidigt den minsta elementet till botten av listan per iteration. Denna typ av sorteringsalgoritm kallas för *Bidirectional Bubblesort* eller *Cocktailsort* som den är mer känd som. Låt oss säga att vi har samma array som för Bubblesort

```
int[] lista = {5, 1, 4, 2, 8};
```

som ska sorteras med Cocktailsort. Det kommer bara ta två iterationer för Cocktailsort att sortera arrayen (elementen skrivna i fetstil jämförs);

Första iterationen:

```
[5, 1, 4, 2, 8] → [1, 5, 4, 2, 8]  
[1, 5, 4, 2, 8] → [1, 4, 5, 2, 8]  
[1, 4, 5, 2, 8] → [1, 4, 2, 5, 8]  
[1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]  
[1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]  
[1, 4, 2, 5, 8] → [1, 2, 4, 5, 8]  
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]
```

Vid detta laget är arrayen redan sorterad men det vet inte algoritmen. Den måste iterera över arrayen utan att behöva byta några element för att veta att den är klar.

Andra iterationen:

```
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]  
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]  
[1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]
```

Denna algoritm är inte mycket svårare att implementera i C# än vad Bubblesort är:

```
1 static void cocktailSort(int[] array)
2 {
3     int start = 0;
4     int end = array.Length;
5     bool swapped = true;
6
7     while (swapped)
8     {
9         for (int i = start; i < end - 1; i++)
10        {
11            if (array[i] > array[i + 1])
```

```

12     {
13         swap(i, i + 1, array);
14         swapped = true;
15     }
16 }
17 if (swapped == false)
18 {
19     break;
20 }
21 swapped = false;
22 end -= 1;
23 for (int i = end - 1; i >= start; i--)
24 {
25     if (array[i] > array[i + 1])
26     {
27         swap(i, i + 1, array);
28         swapped = true;
29     }
30 }
31 start += 1;
32 }
33 }
```

Vi initialiseras funktionen på rad 1. Vi kallar funktionen för `cocktailSort` och säger även att den bara kan ta emot en endimensionell array. Väl inuti funktionen säger vi att `start` och `end` ska vara 0 respektive längden av `arr`. Dessa variabler kommer avgöra vart initierandet kommer börja och sluta. Vi skapar även en boolean `swapped` som kommer avgöra om `arr` är sorterad. På rad 7 börjar en `while` loop som kommer looppa tills `swapped` är falsk. Det första som händer i `while` loopen är att `for` loopen på rad 9 itererar över `arr` från vänster (med `start` på `start` positionen) till höger (med slut vid `end` positionen) och jämför elementet den är på med elementet till höger och byter dom ifall det behövs. Den sätter även `swapped` som san ifall den behövde byta plats på två element. Följt av den `for` loopen är det ett `if` statement på rad 17 som stannar `while` loopen om `swapped` är falsk. Det vill säga listan är sorterad. Men om den inte är sorterad kommer nästa `for` loop på rad 23 att börja. Den fungerar samma som `for` loopen på rad 9 förutom att det itererar över `arr` från vänster till höger istället. Observera att den loopen kommer ignorera objektet längsta till höger i `arr` på grund av `start` och `end` på rad 22 och 31.

Nu kan vi kalla på `cocktailSort` för att sortera `lista`:

```
cocktailSort(lista);
```

Vilket sorterar `lista` så att den nu är:

```
[1, 2, 4, 5, 8]
```

Tidskomplexiteten av Cocktailsort är samma som för Bubblesort. En iteration av Cocktailsort är samma som två iterationer av Bubblesort. Om vi har en array som är n element lång så kommer första iterationen av Cocktailsort behöver jämföra först $(n - 1)$ element och sedan $(n - 2)$ element (detta är ju då på grund av det två `for` looparna på raderna 9 och 23). Andra iterationen kommer Coctailsort behöva jämföra $(n - 3)$ sedan $(n - 4)$ och så vidare. Detta betyder

att den totala summan av iterationer för n element är

$$(n - 1) + (n - 2) + \cdots + 2 + 1 = \sum \frac{n(n - 1)}{2}.$$

Vilket betyder att Coctailsort har ett ordo av $\mathcal{O}(n^2)$.

Både Bubblesorts och Coctailsorts hastighet lider främst på grund av att små värden i början eller slutet i deras respektive listor eftersom det tar längst tid att flytta till rätt plats. Såna värden kallas ibland för *turtles*. Stora värden nära början eller slutet av en lista kan även kallas för *rabbits* eftersom det går snabbast att flytta till rätt plats. Combsorts (vilket är den sista grundläggande algoritmen vi kommer kolla på) syfte är att eliminera rabbits och turtles [1]. När två element jämförs med bubblesort har de alltid ett mellanrum (avstånd från varandra) av ett. Ideen med combsort är att mellanrummet kan vara mer än ett. Den inre loopen av bubblesort som byter plats på elementen är modifierad sådan att mellanrummet mellan den jämförda elementen är minskad med en *krympnings faktor* k [4]. Låt oss säga att vi har en array

```
int [] lista = {5, 1, 4, 2, 8};
```

som ska sorteras med Coctailsort. Mellanrummet mellan det jämförda elementen är lika med hur många element som finns i listan. Vi säger även att $k = 1$. Det vill säga att mellanrummet kommer bli ett mindre per iteration. Det betyder att den första iterationen kommer se ut så här (elementen skrivna i fetstil jämförs):

Första iterationen:

$$[5, 1, 4, 2, 8] \rightarrow [5, 1, 4, 2, 8]$$

Det yttersta elementen jämförs eftersom mellanrummet börjar med att vara lika stort som lista är lång.

Andra iterationen:

$$\begin{aligned} [5, 1, 4, \mathbf{2}, 8] &\rightarrow [2, 1, 4, \mathbf{5}, 8] \\ [2, \mathbf{1}, 4, 5, 8] &\rightarrow [2, 1, 4, \mathbf{5}, 8] \end{aligned}$$

Här är mellanrummet ett mindre eftersom vi valde att k skulle vara ett. Observera att k vanligtvis inte är ett. k brukar vanligast vara lika med 1.3. Detta går vi mer in på detalj snart.

Tredje iterationen:

$$\begin{aligned} [2, 1, \mathbf{4}, 5, 8] &\rightarrow [2, 1, \mathbf{4}, 5, 8] \\ [2, 1, 4, \mathbf{5}, 8] &\rightarrow [2, 1, 4, \mathbf{5}, 8] \\ [2, 1, 4, 5, \mathbf{8}] &\rightarrow [2, 1, 4, 5, 8] \end{aligned}$$

Fjärde iterationen:

$[2, 1, 4, 5, 8] \rightarrow [1, 2, 4, 5, 8]$
 $[1, 2, 4, 5, 8] \rightarrow [1, 2, 4, 5, 8]$
 $[1, 2, 4, 5, 8] \rightarrow [1, 2, 4, 5, 8]$
 $[1, 2, 4, 5, 8] \rightarrow [1, 2, 4, 5, 8]$

Denna sista iteration iteration är exakt samma som bubblesort då k har minskat mellanrummet till bara ett.

Innan vi kan börja skriva combsort måste vi skriva en funktion som kan krympa mellanrummet mellan det elementen som jämförs. Vi börjar med att initialisera k som en variabel

```
int k = array.Length
```

Vi initialiseras den lika lång som `items`. Nu kan vi skriva en funktion som krymper mellanrummet med k :

```
1 static int getNextGap(int gap)
2 {
3     gap = (gap * 10) / 13;
4     if (k < 1)
5     {
6         return 1;
7     }
8     return gap;
9 }
```

Denna funktion kommer krympa mellanrummet med 1.3. Det kan verka slumpmässigt att k ska vara 1.3 men det har visat sig att 1.3 är det värde som ger optimalt hastighet för att sortera en array oavsett arrayens storlek. Vi kan nu skriva combsort.

```
1 static void combSort(int[] array)
2 {
3     bool swapped = true;
4
5     while (gap != 1 || swapped == true)
6     {
7         gap = getNextGap(gap);
8         swapped = false;
9
10        for (int i = 0; i < items - gap; i++)
11        {
12            if (list[i] > list[i + gap])
13            {
14                swap(i, i + gap, list);
15                swapped = true;
16            }
17        }
18    }
19 }
```

Först på rad 3 har vi en `bool` kallad `swapped`. Den används för att avgöra om listan är sorterad eller inte då på rad 5 används den som ett argument för `while`

loopen. `while` loopen kommer att loopa så länge `gap` $\neq 1$ och `swapped` är sann. Det innebär att `while` loopen kommer sluta när arrayen är sorterad. Först i `while` loopen bestäms mellanrummet mellan det jämförda variablerna med `getNextGap` funktionen. Varje loop i `while` loopen kommer mellanrummet krympa med 1.3. Sen kommer en `for` loop gör självaste sorteringen. Observera hur lik `bubbleSort` den är. Det enda som skiljer dom åt är k och `swapped`.

Vi kan nu kalla på `combSort` för att sortera `lista`:

```
combSort(lista);
```

Vilket sorterar listan så att den nu är:

```
[1, 2, 4, 5, 8]
```

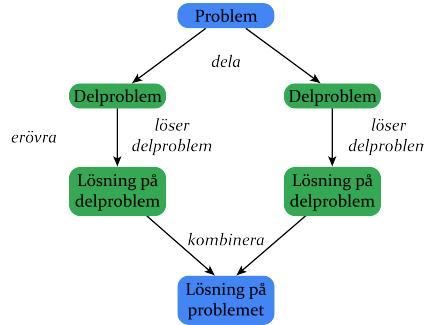
Tidskomplexiteten av Combsort är rätt så knepig att bevisa. Dock har Paul Vitanyi bevisat med Kolmogorov complexity att den har en tidskomplexitet av $\mathcal{O}(n^2)$ på sida 16 i hans artikel “*Analysis of Sorting Algorithms by Kolmogorov Complexity*” [14].

1.3.4 Divide-and-conquer algoritmer (#2)

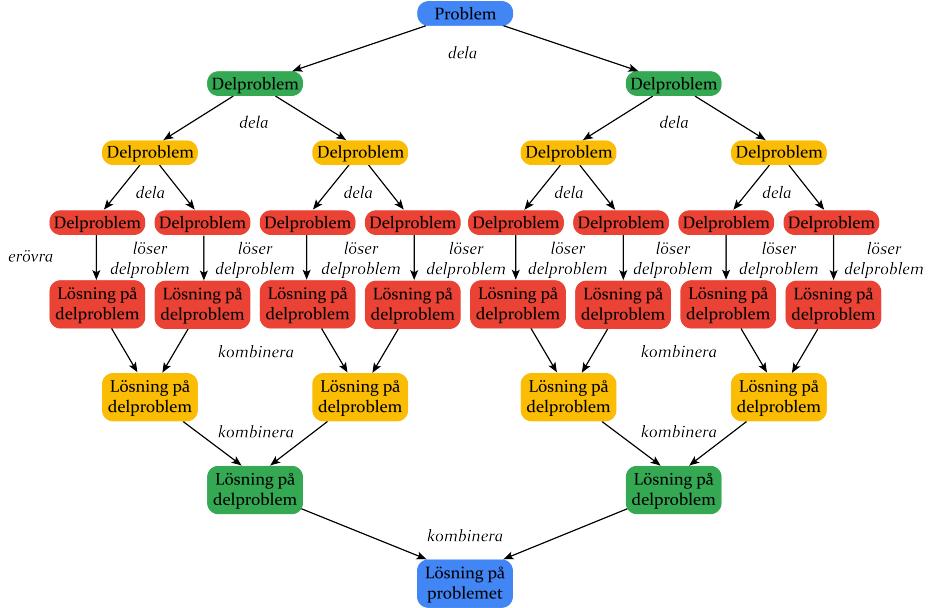
Det algoritmerna vi har kollat på hittills har alla varit väldigt simpla och ineffektiv, vilket vi har sett då alla har haft $\mathcal{O}(n^2)$. Så nu ska vi kolla på Quicksort och Mergesort. Båda är betydligt mycket mer komplexa algoritmer men det är båda väldigt eleganta lösningar till sorteringsproblem. Båda algoritmerna är så kallade “Divide-and-conquer” algoritmer (förkortat DAC). I datavetenskap är DAC ett algoritmiskt paradigm. En DAC algoritm rekursivt delar upp ett problem till två eller flera delproblem av samma eller relaterad typ, tills delproblemen går att lösa omedelbart utan flera delningar. Lösningarna av alla delproblem kombineras sedan för att ge lösning på det ursprungliga problemet [5]. Eftersom DAC löser delproblem rekursivt måste varje delproblem vara mindre än originalproblem, och det måste finnas ett grundläggande fall för delproblem. Man kan tänka sig att DAC algoritmer har tre steg:

1. **Dela** upp problemet i ett antal delproblem som är mindre fall av samma problem.
2. **Erövra** delproblemen genom att lösa dem rekursivt.
3. **Kombinera** det lösta delproblemen tills man löst original problemet.

Om en DAC algoritm skulle lösa ett problem med en rekursioner skulle det se ut så här:



Observera att detta är i fallet då DAC algoritmen skapar två delproblem utav det föregående problemet. Om DAC algoritm skulle lösa ett problem med två rekursioner istället för en skulle det se ut så här: [8]



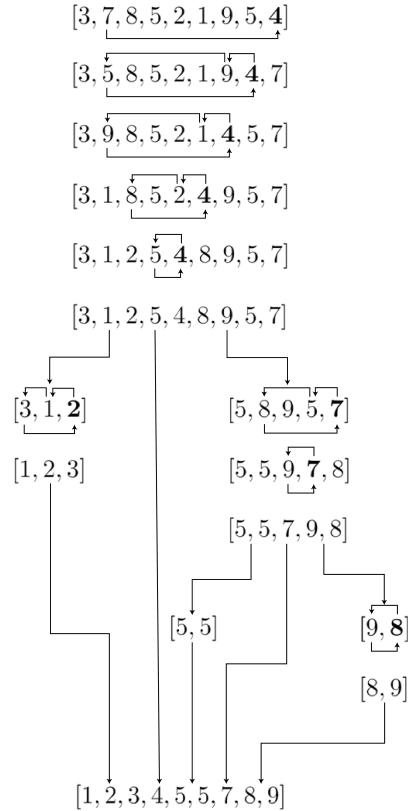
Den första divide-and-conquer algoritmen vi kommer kolla på är Quicksort. Quicksort-algoritmen utvecklades 1959 av Tony Hoare medan han var gäststudent vid Moscow State University. Vid den tiden arbetade Hoare på ett maskinöversättningsprojekt för National Physical Laboratory. I en del av översättningsprocessen behövde han sortera orden i det ryska meningarna innan han kunde kolla upp dom individuella orden i ett rysk-engelskt lexikon (lexikonet var i alfabetisk ordning på magnetisk tejp) [9]. Efter han insåg att hans första idé, insertionsort, skulle vara för långsam för att sortera orden kom han på ideen för quicksort [10]. Quicksort fungerar genom att först dela en given array i två mindre sub-arrayer: det mindre elementen och det större elementen. Efter det sorterar den sub-arrayerna rekursivt. Stegen för sorteringen är det följande:

1. Välj ett element, kalla den för *the pivot*, från arrayen.
2. Partitionering: ändra ordningen på arrayen sådan att element med mindre värden än the pivot kommer innan the pivot och vice versa för det större elementen (element med samma värde som the pivot kan gå till båda sidor). Efter partitioneringen är the pivot på dess slutliga position.
3. Tillämpa stegen ovan rekursivt på sub-arrayerna med element med mindre värden och på sub-arrayerna med element med större värden.

Basfallet för reduktionen är en array av storlek noll eller ett då det är sorterade per definition, så det kräver ingen sortering. Hur man väljer the pivot och hur man partitionerar arrayen kan göras på ett flertal sätt; vilket har stor påverkan algoritmens hastighet. Låt oss säga att vi har en array

```
intp[] lista = {3, 7, 8, 5, 2, 1, 9, 5, 4};
```

som ska sorteras med quicksort. Den fulla sortering skulle se ut såhär (the pivot är skrivet i fetstil)



1.3.5 'Jämförelse på plats' algoritmer (#3)

1.3.6 Övriga algoritmer (#4)

Kapitel 2

Metod

Kapitel 3

Resultat

Kapitel 4

Diskussion och slutsatser

Kapitel 5

Källförteckning

Litteratur

- [1] Bronislava Brejová. *Analyzing variants of Shellsort*. 2021. URL: <https://www.sciencedirect.com/science/article/abs/pii/S0020019000002234?via%3Dihub> (hämtad 2001).
- [2] Cpp.edu. *CS241: Data Structures and Algorithms II*. 2020. URL: <https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm>.
- [3] En.wikipedia.org. *Bubble sort*. 2020. URL: https://en.wikipedia.org/wiki/Bubble_sort (hämtad 2020-11-25).
- [4] En.wikipedia.org. *Comb sort*. 2020. URL: https://en.wikipedia.org/wiki/Comb_sort (hämtad 2021-01-24).
- [5] En.wikipedia.org. *Divide-and-conquer algorithm*. 2021. URL: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm.
- [6] GeeksforGeeks. *Sorting Algorithms*. 2020. URL: <https://www.geeksforgeeks.org/sorting-algorithms/>.
- [7] Sal Khan. *Big O notation*. 2020. URL: <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation> (hämtad 2020-11-16).
- [8] Sal Khan. *Divide and conquer algorithms*. 2021. URL: <https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms>.
- [9] L Shustek. *Interview: An interview with C.A.R. Hoare*. 2009. URL: <https://dl.acm.org/doi/10.1145/1467247.1467261>.
- [10] L Shustek. *Quicksort*. 2021. URL: <https://en.wikipedia.org/wiki/Quicksort>.
- [11] W3schools.com. *C Tutorial (C Sharp)*. 2020. URL: <https://www.w3schools.com/cs/> (hämtad 2020-11-23).
- [12] Wikipedia.org. *Big O notation*. 2020. URL: https://en.wikipedia.org/wiki/Big_O_notation (hämtad 2020-11-16).
- [13] Wikipedia.org. *Sorting algorithm*. 2020. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.
- [14] Paul Vitanyi. *Analysis of Sorting Algorithms by Kolmogorov Complexity (A Survey)*. 2021. URL: <https://hurna.io/assets/files/algorithms/sort/sorting.pdf> (hämtad 2001).

- [15] yourbasic.org. *Big O notation: definition and examples*. 2020. URL: <https://yourbasic.org/algorithms/big-o-notation-explained/> (hämtad 2020-11-16).

