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

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 böckerna snabbare än vad någon människa 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 sortering.
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 sortering 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:[6, 2]

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 sortering attraherat mycket forskning, 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. [6] 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 analyserna hastigheten av det nio olika algoritmerna som nämndes tidigare. Detta kommer redovisat genom att först visa hur man kan skriva algoritmerna i programmeringsspråket C#. Det är viktigt att man förstår hur syntaxen i C# fungerar så det kommer läggas stort fokus på hur syntaxen fungerar och vad den gör. Om man inte förstår programmet är det svårt för en att förstå resten av arbetet då C# är grunden för resten av gymnasiearbetet. Följt av detta kommer matematiken bakom algoritmernas ordo (se sida ???) förklaras. Då matematiken bakom algoritmerna är komplex är det inte i fokus då det inte är lika viktigt för arbetets syfte. Detta innebär att det matematiken kommer förklaras men kommer inte gå in på en djupare nivå än vad som är nödvändigt.
Sorteringsalgoritmer i sin helhet öppnar upp många intressanta dörrar inom datavetenskapen. Men det är ett träsk av information och data som är omöjlig att analysera på en och samma gång. För att undvika det träsket kommer jag begränsa arbetet till enbart så kallade *"Comparison sorting algorithms"* och inte *"Non-Comparison sorting algorithms"*[1] (eller andra former av sorteringsalgoritmer)[2].
Sorteringsalgoritmer handlar inte bara om optimering utan också och om minneshantering inom RAM minnet. Särskilt är vissa sorteringsalgoritmer *på plats.* Det innebär att det bara behöver $\mathcal{O}(1)$ eller $\mathcal{O}(\log n)$ minne (beteckningen $\mathcal{O}$ förklaras mer på sida ???) utöver det objekt som sorteras och det behöver inte skapa temporära platser att spara datan i RAM minnet som andra algoritmer behöver göra. [1] 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 förstå algoritmerna måste man först förstå ett par fundamentala saker. Ordo, så att man förstå matematiken, C# syntax, så man förstår programmen, och visa termer som kommer uppstå ett flertal gånger i arbetet.

1.3.1 Ordo

Ordo (*Big O Notation* på engelska och latin för *ordning*) är ett begrepp inom matematiken och även datavetenskapen som används för att ge ett mått på hur beräkningsmässigt tung en funktion är. Detta betecknas med ett versalt $\mathcal{O}$. När en funktion beskrivs med ordo så beskrivs vanligtvist bara en övre gräns för funktionens tillväxthastighet. Associerat med $\mathcal{O}$ är ett par andra notationer som

[1] Exempel på sådana algoritmer är: Pigeonhole sort, Bucket sort, MSD Radix sort, etc.
[2] Exempel på sådana algoritmer är: Bead sort, Simple Pancake sort, Spaghetti sort, etc.

använder symbolerna o, Ω, ω och Θ som beskriver andra gränser av asymptotisk tillväxthastighet. Dessa kommer vi gå igenom mer i detalj snart. Till exempel betecknar $\mathcal{O}(n^2)$ och $\mathcal{O}(\sqrt{n})$ något som växer lika fort som n^2 respektive $\sqrt{n}$ då n ökar. Detta används särskilt inom datavetenskapen när man beskriver hastigheten av olika algoritmer, som sorteringsalgoritmer. [7, 3]
Den formella definitionen 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 [5]

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

om absolutbeloppet av $f(x)$ är 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 number M och reellt numer 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 number a (ofta $a = 0$). Man skriver

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

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

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

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*:

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

om

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

Det tidigare nämnda relaterade 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 inte är lika viktiga för arbetet.

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 långsammare än g
$f(n) \in \omega(g(n))$	f växer snabbare än g

1.3.2 Fundamental C#

C# (C-Sharp) är ett programmeringsspråk utvecklat av Microsoft som körs på .NET Framework. [4] C# är i första hand ett objektorienterat programspråk. Även kallad ett OOP[3]. Det är varför detta arbete skrivs inom 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:

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

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, 4, 3, 5};
```

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

```
static void swap(int x, int y, int[] array)
{
  int temp = array[x];
  array[x] = array[y];
  array[y] = temp;
}
```

Den tar in två positioner `x` och `y` i en given lista `array` och byter plats på dom. Den göra detta genom att spara heltalet på position `x` i `array` inuti en temporär variabel `temp`. När `x` är sparad i `temp` överskrider funktionen `x` med `y`. Sist överskrider 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, 4, lista);
```

och vi får att listan nu är

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

[3]Från engelskans *Object Oriented Programming*

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 medans `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 numrena 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 hänvisar till `swap` kommer den köra koden som beskrivs ovan.

1.3.3 Grundläggande algoritmer (#1)

Sorteringsalgoritmer kan vara väldigt komplexa och effektiva medans det är väldigt konceptuellt svåra att förstå. Docks så kan sorteringsalgoritmer vara väldigt simpla men med bekostnad av att vara mindre effektiva. Ett par simpla sorteringsalgoritmer vi kommer kolla på nu är *Bubblesort*, *Bidirectional Bubblesort* och, *Shellsort*. Vi börjar med Bubblesort.
Bubblesort är bland det minst effektiva sorteringsalgoritmerna. Men den är ändå den mest kända sorteringsalgoritm eftersom den är sorteringsalgoritmernas *"Hello World"*. Den är väldigt lätt att förstå både praktiskt och teoretiskt. Därför används den mest som ett pedagogiskt verktyg.[**8**] 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:

$$[\mathbf{5}, \mathbf{1}, 4, 2, 8] \rightarrow [\mathbf{1}, \mathbf{5}, 4, 2, 8]$$
$$[1, \mathbf{5}, \mathbf{4}, 2, 8] \rightarrow [1, \mathbf{4}, \mathbf{5}, 2, 8]$$
$$[1, 4, \mathbf{5}, \mathbf{2}, 8] \rightarrow [1, 4, \mathbf{2}, \mathbf{5}, 8]$$
$$[1, 4, 2, \mathbf{5}, \mathbf{8}] \rightarrow [1, 4, 2, \mathbf{5}, \mathbf{8}]$$

Andra iterationen:

$$[\mathbf{1}, \mathbf{4}, 2, 5, 8] \rightarrow [\mathbf{1}, \mathbf{4}, 2, 5, 8]$$
$$[1, \mathbf{4}, \mathbf{2}, 5, 8] \rightarrow [1, \mathbf{2}, \mathbf{4}, 5, 8]$$
$$[1, 2, \mathbf{4}, \mathbf{5}, 8] \rightarrow [1, 2, \mathbf{4}, \mathbf{5}, 8]$$
$$[1, 2, 4, \mathbf{5}, \mathbf{8}] \rightarrow [1, 2, 4, \mathbf{5}, \mathbf{8}]$$

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

Tredje iterationen:

$$[\mathbf{1}, \mathbf{2}, 4, 5, 8] \rightarrow [\mathbf{1}, \mathbf{2}, 4, 5, 8]$$
$$[1, \mathbf{2}, \mathbf{4}, 5, 8] \rightarrow [1, \mathbf{2}, \mathbf{4}, 5, 8]$$
$$[1, 2, \mathbf{4}, \mathbf{5}, 8] \rightarrow [1, 2, \mathbf{4}, \mathbf{5}, 8]$$
$$[1, 2, 4, \mathbf{5}, \mathbf{8}] \rightarrow [1, 2, 4, \mathbf{5}, \mathbf{8}]$$

Denna algoritm är lätt att implementera i C# då den inte kräver många logiska operationer för att utföras:

```
static void bubbleSort(int[] array)
{
  items = array.Length;
  for (int i = 0; i < items; i++)
  {
    for (int j = 0; j < items - 1; j++)
    {
      if (array[j] > array[i])
      {
        swap(i, j, array);
      }
    }
  }
}
```

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

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

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

5 Källförteckning

Referenser

[1] Cpp.edu. *CS241: Data Structures and Algorithms II*. 2020. URL: https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm.

[2] GeeksforGeeks. *Sorting Algorithms*. 2020. URL: https://www.geeksforgeeks.org/sorting-algorithms/.

[3] khanacademy.org. *Big O notation*. 2020. URL: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation (hämtad 2020-11-16).

[4] W3schools.com. *C Tutorial (C Sharp)*. 2020. URL: https://www.w3schools.com/cs/ (hämtad 2020-11-23).

[5] Wikipedia.org. *Big O notation*. 2020. URL: https://en.wikipedia.org/wiki/Big_O_notation (hämtad 2020-11-16).

[6] Wikipedia.org. *Sorting algorithm*. 2020. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.

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