Sortera en array

Nipun Thianwan

VT 2025

1 Introduction

Denna uppgift handlar om att undersöka och jämföra olika sökalgoritmer för att förstå deras effektivitet och begränsningar. Fokus ligger på att analysera deras prestanda i fasta arrayer där inga nya element tillkommer. För varje algoritm ska beräkningstiden uttryckas som en funktion av arrayens storlek för att identifiera dess styrkor och svagheter.

2 the not so efficient

I denna deluppgift fick skribenten en ofullständig kod som behövde kompletteras. Den slutgiltiga kompletteringen resulterade i följande fullständiga kod:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIdx];
        arr[minIdx] = temp;
    }
}
```

Selection sort är en enkel algoritm som fungerar bäst för små arrayer. Den börjar med att välja ett index vid första positionen och söker sedan efter det minsta elementet i resten av arrayen. Om ett mindre värde hittas byts det ut med indexvärdet, och processen upprepas genom att gå vidare till nästa position. Sorteringen fortsätter tills hela arrayen är ordnad. Algoritmen är effektiv för små dataset men blir långsam och ineffektiv för stora mängder data.

2.1 benchmark 1

Selection Sort fungerar genom att: först gå igenom listan och hitta det minsta elementet → O(n). sen byta plats med första osorterade elementet. Efter det upprepa processen för varje index i listan. Koden kördes flera gånger med olika arraystorlekar, vilket resulterade i följande resultat: Dessa resultat bekräftar att Selection Sort har en beräknad

Array Size (n)	**Execution Time (ms)**
100	0
500	1
1000	3
5000	18
10000	41

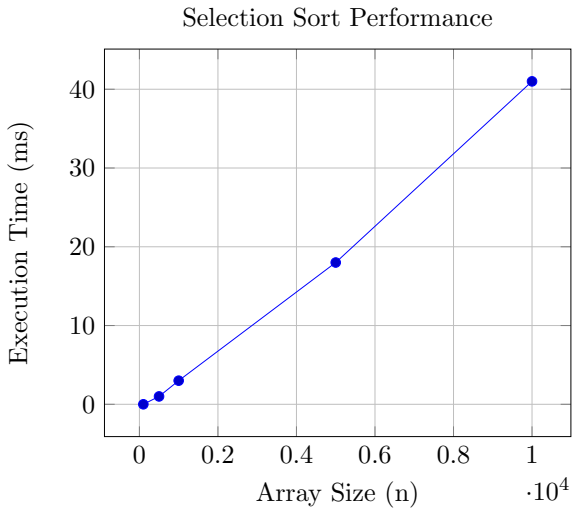
Tabell 1: Exekveringstid för Selection Sort vid olika arraystorlekar

Tidskomplexitet av $O(n^2)$. Om vi ökar antalet element med en faktor 10, bör exekveringstiden öka ungefär 100 gånger, eftersom algoritmen gör jämförelser för varje element mot resten av de osorterade elementen.

I vårt experiment observerade vi att tiden inte växer exakt enligt $O(n^2)$. Exempelvis, när vi gick från 1000 till 10000 element, ökade tiden från 3 ms till 41 ms, vilket endast är ungefär 13,67 gånger längre. Detta beror på att moderna processorer och cache-optimeringar gör att exekveringstiden påverkas av andra faktorer än enbart den teoretiska komplexiteten.

För att ytterligare visualisera prestandan av Selection Sort ritar vi en graf som visar sambandet mellan arraystorlek och exekveringstid. Grafen i Figur 1 visar att tillväxten följer en kvadratisk kurva, vilket stöder vår analys.

Vi har bekräftat att Selection Sort har en kvadratisk tidskomplexitet, vilket gör den ineffektiv för stora dataset. Resultaten visar att exekveringstiden ökar drastiskt när vi ökar antalet element i arrayen.



Figur 1: Exekveringstid för Selection Sort

3 slightly more complicated

I nästa deluppgift skulle vi implementera insertion sort. Här pekar vårt index på ett värde och jämför det med nästa värde i arrayen. Om det senare är mindre, flyttas det framåt. Denna process upprepas genom hela arrayen. Likt föregående uppgift behövde vi komplettera en given kod. Lösningen blev då komplett och är snabbare än selection sort. Den slutgiltiga, fullständiga koden efter komplettering är:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && arr[j] < arr[j - 1]; j--) {
            swap(&arr[j], &arr[j - 1]);
        }
    }
}
```

3.1 benchmark 2

Koden kördes flera gånger med olika arraystorlekar, vilket resulterade i följande utfall:

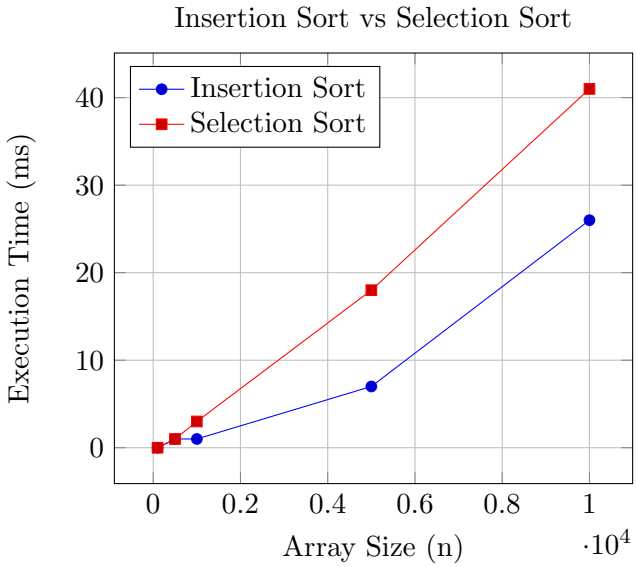
Resultaten visar att Insertion Sort är betydligt snabbare än Selection Sort för alla arraystorlekar. Skillnaden är särskilt tydlig vid större dataset. Detta beror på att Insertion Sort gör färre onödiga jämförelser och byten när arrayen redan är delvis sorterad.

För att ytterligare visualisera prestandaskillnaden mellan de två algoritmerna, presenterar vi exekveringstiden i en graf. Grafen i Figur 2 visar att tillväxten för Selection Sort följer en tydligare kvadratisk trend jämfört med Insertion Sort.

Insertion Sort visade sig vara överlägsen Selection Sort i samtliga tester, särskilt för större dataset. Detta beror på att Selection Sort alltid gör $O(n^2)$ operationer, medan

Array Size (n)	Insertion Sort (ms)	Selection Sort (ms)
100	0	0
500	1	1
1000	1	3
5000	7	18
10000	26	41

Tabell 2: Exekveringstid (ms) för Insertion Sort och Selection Sort



Figur 2: Jämförelse av exekveringstid mellan Insertion Sort och Selection Sort

Insertion Sort kan prestera bättre när arrayen är delvis sorterad. Resultaten bekräftar den teoretiska analysen och visar att Insertion Sort bör föredras över Selection Sort för små till medelstora listor, särskilt när data ofta är nästan sorterad.

4 completely different

Mergesort fungerar genom att dela upp en array i två delar, sortera dem separat och sedan slå ihop dem till en sorterad array. Namnet kommer från denna sammanslagningsprocess. För att slå ihop de sorterade delarna används en temporär lagring.

Principen kan liknas vid att sortera en delad kortlek: de minsta korten ligger överst i varje hög, och man jämför de översta korten, lägger det minsta först och fortsätter tills alla kort är sorterade.

Den slutgiltiga, fullständiga koden efter komplettering är:

```
void merge(int *org, int *aux, int lo, int mid, int hi) {
    for (int i = lo; i <= hi; i++) {
```

```
        aux[i] = org[i];
    }

    int i = lo;
    int j = mid + 1;

    for (int k = lo; k <= hi; k++) {
        if (i > mid) {
            org[k] = aux[j++];
        } else if (j > hi) {
            org[k] = aux[i++];
        } else if (aux[i] <= aux[j]) {
            org[k] = aux[i++];
        } else {
            org[k] = aux[j++];
        }
    }
}

void sort(int *org, int *aux, int lo, int hi) {
    if (lo >= hi) return;

    int mid = (lo + hi) / 2;

    sort(org, aux, lo, mid);
    sort(org, aux, mid + 1, hi);

    merge(org, aux, lo, mid, hi);
}
```

Dela upp: sort delar arrayen i mindre delar tills varje del innehåller ett enda element. Sortera: Varje del sorteras rekursivt. Slå ihop: merge kombinerar de sorterade delarna genom att jämföra och ordna elementen i rätt ordning.

4.1 benchmark 3

Koden kördes flera gånger med olika arraystorlekar, vilket resulterade i följande utfall:

tid (ms)	**n-värde**
0	100
0	500
1	1000
5	5000
10	10000

Tabell 3: Exekveringstid för Merge Sort

Benchmarkresultaten bekräftar att Merge Sort är den mest effektiva algoritmen för

stora dataset tack vare sin rekursiva struktur och $\mathcal{O}(n \log n)$ tidskomplexitet. Genom att dela upp, sortera och slå ihop mindre delar rekursivt, uppnår den bättre prestanda än både Insertion Sort och Selection Sort. Insertion Sort presterar bättre än Selection Sort, särskilt vid delvis sorterade listor, men båda har $\mathcal{O}(n^2)$ komplexitet, vilket gör dem ineffektiva för stora dataset. Merge Sort är därför det bästa valet när snabb och konsekvent sortering krävs.