**Laboration 1:   
Algoritmer & Tidskomplexitet**

**Författare:** Isak Lagerberg

Mattias Arvidsson

Joakim Sandström

**Nivå**: Grundnivå  
**Kursnamn:** Algoritmer och mjukvarudesign **Kurskod:** GMI24H **Högskolepoäng:** 7,5HP **Handledare**: Elin Ekman  
**Institution:** Institutionen för information och teknik  
**Examinator:** Elin Ekman **Examinationsdatum:** 2025-04-11

**Antal Ord:** 860

Innehållsförteckning **Abstract  
Keywords  
Sammanfattning**

[1. Inledning 0](#_Toc194929392)

[1.1. Bakgrund 0](#_Toc194929393)

[1.1.1. Syfte 0](#_Toc194929394)

[2. Metod 1](#_Toc194929395)

[2.1. Implementation 2](#_Toc194929396)

[2.1.1. Datainsamling 2](#_Toc194929397)

[3. Resultat 3](#_Toc194929398)

[3.1. Deluppgift 1 – Räkna förekomster i en array 3](#_Toc194929399)

[3.2. Deluppgift 2 – Vända en array 6](#_Toc194929400)

[3.2.1. O(N) algoritmen 6](#_Toc194929401)

[3.2.2. O(N^2) algoritmen 7](#_Toc194929402)

[3.2.1. Övergripande jämförelse 8](#_Toc194929403)

[4. Diskussion 9](#_Toc194929404)

[4.1. Deluppgift 1 – Räkna förekomster i en array 9](#_Toc194929405)

[4.2. Deluppgift 2 – Vända en array 9](#_Toc194929406)

[5. Slutsats 9](#_Toc194929407)

**Referenser  
Bilagor**

1. Inledning

I denna laboration har vi genomfört två deluppgifter med syftet att undersöka hur algoritmers indata och komplexitet påverkar exekveringstiden samt minnesanvändningen. Den första deluppgiften går ut på att räkna antalet förekomster av ett specifikt tal i en array, medan den andra deluppgiften handlar om att vända ordningen på elementen i en array. Genom att implementera lösningarna i C# med .NET har vi inte bara studerat de teoretiska aspekterna, såsom tidskomplexitet (O(n) respektive O(n²)), utan även genomfört praktiska mätningar med hjälp av DateTime och TimeSpan.

* 1. Bakgrund

Vid design av mjukvara är det avgörande att förstå hur olika algoritmer presterar beroende på datastruktur och inputstorlek. Denna laboration fokuserar på två centrala operationer: att räkna hur många gånger ett specifikt tal förekommer i en lista samt att vända ordningen på en array.

* + 1. **Syfte**

Syftet med att undersöka hur implementationer från ”ax till limpa” utan hjälp av inbyggda C#-funktioner påverkar exekveringstid och komplexitet. Genom att spara resultatet till .csv och analysera i Excel kan vi dra slutsatser kring tidskomplexitet.

1. Metod

För att genomföra laborationen implementerades två olika algoritmer med särskilda mål och angreppssätt:

1. **Deluppgift 1 – Räkna förekomster i en array**
   1. **Problemformulering:** Utveckla en algoritm som, givet en array och ett angivet tal, räknar hur många gånger talet förekommer.
   2. **Implementering:** Algoritmen itererar genom hela arrayen och ökar en räknare varje gång måltalet hittas. För att säkerställa reproducerbara testfall användes Random() med en fixerad seed vid generering av populerade arrayer.
   3. **Prestandamätning:** Exekveringstiden mättes med DateTime.Now() samt med TimeSpan-klassen varpå differansen beräknades från starttid till sluttid.
2. **Deluppgift 2 – Vända en array**
   1. **Problemformulering:** Skapa en algoritm som vänder ordningen på elementen i en array. Två metoder testades:
      1. Första metoden, baserad på given pseudokod, som beräknas ha en tidskomplexitet på O(n²) då den genomför upprepade operationer för att flytta element.
      2. En förbättrad in-place-algoritm, som byter plats på elementen från början och slutet av arrayen utan att använda någon extra datastruktur, vilket ger en minneskomplexitet på O(1) och en linjär tidskomplexitet O(n).
   2. **Implementering:** Båda algoritmerna utvecklades i C#. Den in-place-metoden byter element med hjälp av en temporär variabel i en loop som körs n/2 gånger, där n är antalet element.
   3. **Prestandamätning:** Mätningar genomfördes med DateTime för att bekräfta de teoretiska beräkningarna. Kurvor och diagram över exekveringstider ritas upp för att tydligt demonstrera skillnaderna i prestanda mellan den O(n²)-algoritm och den optimerade O(n)-metoden.

Genom att jämföra den teoretiska komplexiteten med de praktiska mätningarna får vi en tydlig bild av hur algoritmval och implementeringsmetod påverkar både exekveringstid och minnesanvändning.

* 1. Implementation

Koden består av tre klasser:

Program.cs: Startar programmet och låter användaren välja vilken uppgift som ska köras.

OccurrenceCounter.cs: Generear array, räknar förekomster av ett valt tal, mäter och sparar tiden det tog för algoritmen att räkna ut antalet förekomster.

Reverse.cs: Vänder arrayen med hjälp av en snabb O(n)-algoritm, med en minneskomplexitet på O(1), eller en långsam O(n^2)-algoritm, med en minneskomplexitet på O(1), beroende på användarens val av metod.

De mätta tiderna för varje körning skrivs till tre olika .csv-filer: PerformanceCounterData.csv, PerformanceReverseOrderNormalData.csv samt PerformanceReverseOrderSlowData.csv.

* + 1. Datainsamling

Varje algoritm kördes med olika liststorlekar men med samma seed (frö). Exekveringstiden som vi mäter i millisekunder, liststorlek, seed samt förekomster loggas. Datan analyseras sedan i Excel där datan omhändertas och sammanställs till diagram för att identifiera komplexitetstrender. Stapeldiagram och scatter plots användes för att visualisera tidssökningen vid ökad inputsstorlek.

1. Resultat
   1. Deluppgift 1 – Räkna förekomster i en array

|  |  |  |  |
| --- | --- | --- | --- |
| **Listlängd** | **O(N)** | **Förekomster** | **Seed** |
| 10 | 4,2499 | 3 | 101 |
| 100 | 3,2162 | 14 | 101 |
| 1000 | 8,6387 | 85 | 101 |
| 10000 | 4,681 | 945 | 101 |
| 100000 | 3,956 | 9835 | 101 |
| 1000000 | 7,7001 | 99555 | 101 |

Tabell 1: Exekveringstider för förekomstalgoritmen

Figur 1: Tidskomplexitet för förekomstalgoritmen (första 6 listorna)

|  |  |  |  |
| --- | --- | --- | --- |
| **Listlängd** | **ms** | **Förekomster** | **Seed** |
| 10 | 4,2499 | 3 | 101 |
| 100 | 3,2162 | 14 | 101 |
| 1000 | 8,6387 | 85 | 101 |
| 10000 | 4,681 | 945 | 101 |
| 100000 | 3,956 | 9835 | 101 |
| 1000000 | 7,7001 | 99555 | 101 |
| 10000000 | 27,9454 | 999407 | 101 |
| 100000000 | 205,8482 | 10002198 | 101 |
| 1000000000 | 2381,5983 | 99989033 | 101 |

Tabell 2: Exekveringstider för förekomstalgoritmen

Figur 2: Exekveringstid för förekomstalgoritmen vid alla listor

|  |  |  |  |
| --- | --- | --- | --- |
| **Körning** | **Nummer** | **Förekomster** | **Seed** |
| 1 | 2 | 3 | 101 |
| 2 | 2 | 14 | 101 |
| 3 | 2 | 85 | 101 |
| 4 | 2 | 945 | 101 |
| 5 | 2 | 9835 | 101 |
| 6 | 2 | 99555 | 101 |
| 7 | 2 | 999407 | 101 |
| 8 | 2 | 10002198 | 101 |
| 9 | 2 | 99989033 | 101 |

Tabell 3: Förekomster av tal 2 vid olika körningar

Figur 3: Förekomster av värde 2 över flera körningar

* 1. Deluppgift 2 – Vända en array
     1. O(N) algoritmen

|  |  |  |
| --- | --- | --- |
| **Listlängd** | **ms** | **Seed** |
| 100 | 2,2727 | 101 |
| 1000 | 2,6244 | 101 |
| 10000 | 2,2121 | 101 |
| 100000 | 2,362 | 101 |
| 1000000 | 5,3525 | 101 |

Tabell 4: Exekveringstider för ReverseOrder-algoritmen (O(n))

Figur 4: Tidskomplexitet för ReverseOrder-algoritmen

* + 1. O(N^2) algoritmen

|  |  |  |
| --- | --- | --- |
| **Listlängd** | **ms** | **Seed** |
| 100 | 6,1425 | 101 |
| 1000 | 2,6346 | 101 |
| 10000 | 58,6069 | 101 |
| 100000 | 17117,3343 | 101 |
| 1000000 | 566751,0627 | 101 |

Tabell 5: Exekveringstider för ReverseOrderSlow-algoritmen (O(n²))

Figur 5: Exekveringstid för ReverseOrderSlow-algoritmen

* + 1. Övergripande jämförelse

Figur 6: Jämförelse mellan ReverseOrder (O(n)) och ReverseOrderSlow (O(n²))

1. Diskussion
   1. Deluppgift 1 – Räkna förekomster i en array

Figur 2 visar en trend som påminner om linjär tidskomplexitet, även om den inte följs helt strikt – vilket framgår tydligare i figur 1. Avvikelserna kan förklaras av flera faktorer, såsom systemets overhead, JIT-kompilering, garbage collection, hur data organisering i arrayen ser ut samt hur hanteringen av konstanter och instansieringar i programmet omhändertas. Dessa aspekter påverkar exekveringstiden och bidrar till att ökningen inte är helt jämn (något som även är relevant i deluppgift 2). Trots detta framträder en övergripande proportionell ökning, vilket tyder på att algoritmen i stort sett skalar linjärt med datamängden – särskilt när mindre variationer och externa faktorer bortses från.

* 1. Deluppgift 2 – Vända en array

Vid jämförelse av algoritmerna O(N) och O(N^2) för att vända om en array, visade det sig att O(N)-algoritmen presterade effektivt även för stora arrayer, med en stabil och linjär ökning i exekveringstid. För arrayer med över 1 000 000 element ökade exekveringstiden fortfarande, men påverkades mindre än O(N^2)-algoritmen. Denna senare algoritm hade en rimlig prestanda för små arrayer, men när listan växte över 10 000 element ökade exekveringstiden kraftigt, och för 100 000 och 1 000 000 element blev den mycket långsam. Detta är typiskt för algoritmer med kvadratisk tidskomplexitet, där ökning i storlek leder till en exponentiell ökning i operationer.

1. Slutsats

Avslutningsvis kan vi konstatera att algoritmerna följer deras tilldelade big O notation med hänsyn till mindre variationer, externa faktorer eller påverkande konstanter. De linjära algoritmer som behandlats visar på en jämn och skalbar prestanda, medan den kvadratiska snabbt blev ineffektiv vid större datamängder. Detta understryker vikten av att välja rätt algoritm vid rätt tillfälle med hänsyn till problemets storlek och krav på effektivitet.

# Referenser

Dalarna University. (n.d.). *Föreläsning om algoritmkomplexitet* [Video]. <https://play.du.se/media/t/0_8vs94yo5>

Dalarna University. (n.d.). *Föreläsning om implementation av algoritmer* [Video]. <https://play.du.se/media/t/0_9mcnzpfj>

Stephens, R. (2019). *Essential Algorithms* (2 uppl.). John Wiley & Sons Inc.