Arrayer och prestanda

Nipun Thianwan

VT 2025

Introduktion

Detta är den första rapporten för kursen ID1021, som ges vid KTH. Kursens syfte är att ge en djupare förståelse för grundläggande algoritmer och datastrukturer.

I den inledande uppgiften analyserades och presenterades tidsåtgången för olika kodsnuttar. Dessa innehöll tre operationer: *Random access*, *Search* och *Duplicates*. För de två senare operationerna utvecklades dessutom polynom som bäst beskriver deras beteende.

Rapporten skrivs i Overleaf för att ge möjlighet att bekanta sig med latex

Körtid - Random access

Vår första uppgift var att mäta körtiden för att utföra en operation. Det skulle även undersökas hur exakt denna mätning kunde genomföras. För att möjliggöra detta tillhandahölls följande kod:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

long nano_seconds(struct timespec *t_start, struct timespec *
    t_stop) {
    return (t_stop->tv_nsec - t_start->tv_nsec) +
           (t_stop->tv_sec - t_start->tv_sec) * 1000000000;
}

int main() {
    struct timespec t_start, t_stop;
    for (int i = 0; i < 10; i++) {
        clock_gettime(CLOCK_MONOTONIC, &t_start);
        clock_gettime(CLOCK_MONOTONIC, &t_stop);
        long wall = nano_seconds(&t_start, &t_stop);
```

```
        printf("%ld ns\n", wall);
    }
}
```

Resultatet visade högre initiala tider (48–57 ns) följda av stabila tider mellan 14–17 ns. De högre initiala tiderna beror på kallstartseffekten, där data laddas in i cachen och processorn optimerar åtkomsten. Efter stabiliseringen visar de lägre tiderna den faktiska prestandan för `clock_gettime`-anropen. Små variationer orsakas av operativsystemets samtidiga processer. Slutsatsen är att stabiliserade mätvärden ger mer meningsfulla resultat.

När följande kod körs:

```
int main() {
    struct timespec t_start, t_stop;
    int given[] = {1,2,3,4,5,6,7,8,9,0};
    int sum = 0;
    for(int i = 0; i < 10; i++) {
        clock_gettime(CLOCK_MONOTONIC, &t_start);
        sum += given[i];
        clock_gettime(CLOCK_MONOTONIC, &t_stop);
        long wall = nano_seconds(&t_start, &t_stop);
        printf("one operation in %ld ns\n", wall);
    }
}
```

Resultaten visade högre initiala tider (51–57 ns) och stabila tider mellan 15–18 ns. Slutsatsen är att cacheuppvärmning orsakar de högre initiala tiderna, medan stabila tider representerar den faktiska prestandan.

```
int main() {
    struct timespec t_start, t_stop;
    int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int sum = 0;

    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < 1000; i++) {
        sum += array[rand() % 10];
    }
    clock_gettime(CLOCK_MONOTONIC, &t_stop);

    long wall = nano_seconds(&t_start, &t_stop);
    printf("%ld ns\n", wall / 1000);
}
```

Programmet mäter genomsnittstiden för 1000 slumpmässiga arrayåtkomster. Resultaten varierar mellan 5 ns och 11 ns. Detta visar att processorns cache används effektivt, vilket möjliggör snabba arrayåtkomster efter initial optimering. Variationerna beror huvudsakligen på operativsystemets schemaläggning och samtidiga bakgrundsprocesser, men påverkar resultaten marginellt. Användningen av en liten array på 10 element förstärker

cacheoptimeringen, medan slumpberäkningen (`rand() % 10`) introducerar försumbar overhead.

I nästa uppgift skulle vi köra en kodsnutt av typen "Benchmark" flera gånger. Som student fick vi välja hur resultaten skulle rapporteras: antingen som medelvärde, median eller minsta värde. I denna rapport valde skribenten att analysera minsta värdet. Efter att ha kört koden flera gånger kunde en slutsats dras baserat på de observerade resultaten. När den modifierade Benchmarken"kördes, denna:

```
long bench(int n, int loop) {
    int *array = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) array[i] = i;

    int *indx = (int *)malloc(loop * sizeof(int));
    for (int i = 0; i < loop; i++) indx[i] = rand() % n;

    int sum = 0;
    struct timespec t_start, t_stop;

    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < loop; i++) sum += array[indx[i]];
    clock_gettime(CLOCK_MONOTONIC, &t_stop);

    long wall = nano_seconds(&t_start, &t_stop);
    free(array);
    free(indx);
    return wall;
}
```

Resultaten visar att tiden per arrayoperation ökar med arraystorleken. För små arrayer, som 1000 eller 2000 element, var åtkomsttiden stabil runt 14–16 ns, medan större arrayer, som 32000 element, nådde cirka 35 ns. Ökningen beror på cacheminnets begränsningar, där små arrayer ryms i cachen medan större arrayer kräver långsammare åtkomst till huvudminnet.

Genom att fokusera på minsta värdet kunde den optimala prestandan identifieras, fri från brus som orsakas av bakgrundsprocesser. Detta gör metoden tillförlitlig för att analysera prestanda i enkla operationer.

Search

I nästa uppgift analyserades hur exekveringstiden för en sökaloritm påverkas av arrayens storlek. Genom att mäta tidsåtgången vid sökning efter slumpmässiga nycklar undersöktes algoritmens prestanda.

```
int main(int argc, char *argv[]) {
    int sizes[] = {100, 200, 400, 800, 1600}; // Different sizes
       of the array
    int loop = 1000; // Number of search operations
    int k = 10; // Number of iterations for averaging
```

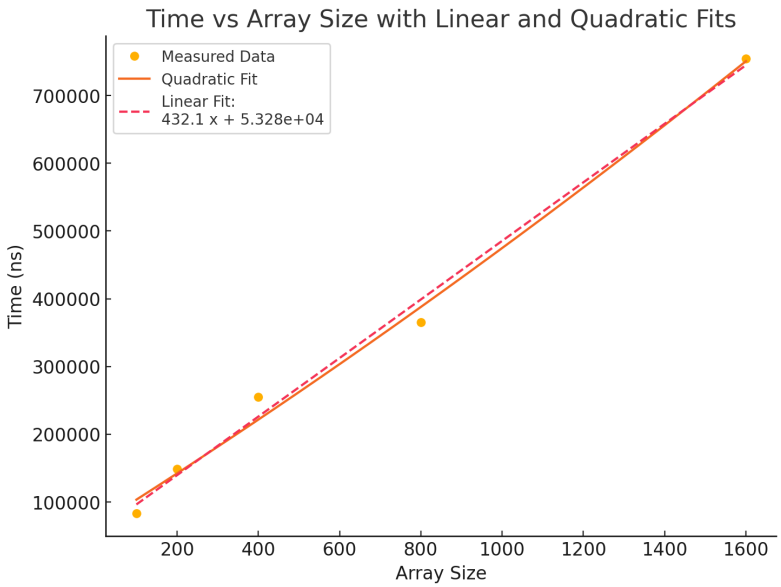
```
    printf("Array Size\tTime (ns)\n");
    printf("--------------------------\n");

    for (int i = 0; i < sizeof(sizes) / sizeof(sizes[0]); i++) {
        int n = sizes[i];
        long total_time = 0;

        for (int j = 0; j < k; j++) {
            total_time += search(n, loop); // Call the search
                function multiple times
        }

        long average_time = total_time / k; // Calculate average
            time
        printf("%d\t\t%ld ns\n", n, average_time); // Print the
            results in table format
    }
```

Den mäter tiden det tar att utföra linjärsökning i arrays av olika storlekar och presenterar resultaten i en tabell. Arraystorlekarna som testas är 100, 200, 400, 800 och 1600, och för varje storlek söks 1000 nycklar i arrayen. Varje storlek testas 10 gånger för att beräkna ett genomsnitt av söktiden i nanosekunder. Funktionen `search` används för att skapa arrayen, generera nycklar, utföra sökningar och mäta tiden för varje iteration. Resultaten för varje storlek sammanställs och skrivs ut i tabellform, där både arraystorlek och genomsnittlig tid visas. Syftet är att analysera hur linjärsökningens prestanda påverkas av arrayens storlek.

Array Size	**Time (ns)**
100	83235
200	148759
400	254769
800	364986
1600	754182

Tabell 1: resultat av exekvering

Resultatet lades in Excel för att kunna visualiseras och få fram ett passande polynom.



Figur 1: Excel-graf med polynom, tid angiven i ns.

Med hjälp av Excel kunde vi visa en linjär relation, där vi erhöll ett linjärt polynom. Funktionen ligger nära trendlinjen, som representerar en perfekt linjär linje. Med fler datapunkter hade förhållandet blivit ännu närmare 1:1. Som illustreras i figuren nedan.

Duplicates

Nu till den sista uppgiften: att hitta dubbletter mellan två arrayer av storlek n, det vill säga ett element som finns i den första arrayen och även i den andra. Denna uppgift liknar mycket sökövningen och vi kommer att använda samma strategi: för varje nyckel i den första arrayen, försök att hitta den i den andra arrayen. Skillnaden här är att båda arrayerna växer i storlek; en liten förändring kan tyckas, men det gör en enorm skillnad. Om arraystorleken är tillräckligt stor ($n > 100$) kan vi hoppa över loopparametern; tiden för att hitta dubbletter kommer att vara tillräckligt lång för att dölja osäkerheter i klockan.

```
int main() {
    int sizes[] = {100, 200, 400, 800, 1600, 3200};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    printf("Array size\tTime (ns)\n");

    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        long time = duplicates(n);
        printf("%d\t\t%ld\n", n, time);
    }
```

```
    }
```

Program implementerades en algoritm som söker efter gemensamma element mellan två arrayer av storlek n. Programmet genererar två arrayer med slumpmässiga heltal, där varje element från den första arrayen jämförs med alla element i den andra. Tiden som krävs för att hitta alla dubbletter beräknas med hjälp av funktionen `clock_gettime`, vilket möjliggör mätning av exekveringstid med hög precision.

Tabell 2: Exekveringstid för dubblettsökning

Array Size (n)	**Time (ns)**
100	7280
200	25501
400	99592
800	370625
1600	1181814
3200	4834832

Exekveringstiden för *Duplicates Search* presenteras i Tabell 2. Tidsåtgången ökar snabbt med växande arraystorlekar, vilket reflekterar den algoritmiska komplexiteten. För små arrayer, som $n = 100$, är tiden relativt kort, medan för större storlekar, som $n = 3200$, ökar tiden exponentiellt. Detta resultat bekräftar att algoritmens tidskomplexitet är $O(n^2)$, eftersom varje element i den första arrayen jämförs med alla element i den andra.

Figur 2: Excel graf med polynom, tid angiven i ns

Resultaten visualiserades i en graf (Figur 2), där exekveringstiden plottas mot arraystorleken. Den resulterande kurvan följer en kvadratisk trend, vilket överensstämmer

med den förväntade tidskomplexiteten. En polynomtrendlinje av andra graden applicerades på data och visade en hög anpassning till de observerade värdena. Detta stödjer slutsatsen att algoritmens prestanda påverkas signifikant av arraystorlekens kvadratiska tillväxt.