Funktionsanalyser

Laboration 1

Ahmad Shammout, shmoot001@gmail.com
Johanna Olsson, johannamaria1@live.se

Innehållsförteckning

1. Introduktion

Laborationen gick ut på att skapa ett meny-drivet program som skulle beräkna de genomsnittliga, bästa, samt värsta fallen av fem olika funktioner. Tre funktioner var sorteringsfunktioner; *bubble sort*, *quick sort*, samt *insertion sort*, och två var sökfunktioner; *linear search* samt *binary search*. Programmet skulle delas upp i *front-end* och *back-end*, där *front-end* sköter presentationer av meny och resultat för användaren, och *back-end* sköter bakgrundsarbetet; hantering av algoritmerna, tidtagning, och annat som ska döljas för användaren. Resultatet visas i tabellform där exekveringstid, rättfärdigande av korrekt *Big-O* komplexitet, samt storlek av sekvenserna skulle inkluderas.

2. Bakgrund

Inför laborationen krävdes vissa förberedelser och förundersökningar. Detta för att få förståelse för de algoritmer som skulle implementeras i programmet och en uppfattning kring förväntade resultat. Här presenteras de olika algoritmerna samt hur de genomsnittliga, bästa, samt värsta fallen togs fram.

2.1 Tidskomplexitet

Inom datatekniken finns det ofta flera sätt att lösa programmeringsproblem på, vilket gör det viktigt att jämföra olika metoder och välja ut de som är mest optimala för varje problem *(GeeksForGeeks, 2022)*. Tidskomplexitet är ett sätt att bedöma hur användbara vissa algoritmer är beroende på hur lång tid de tar att köra *(GeeksForGeeks, 2022)*. För att beräkna tidskomplexitet används något som heter *Big-O* notationen *(Huang, 2020)*. *Big-O* är en matematisk notation som beskriver hur tidskomplexiteten för en funktion rör sig mot ett visst värde eller mot oändligheten. Ett exempel på detta är $O(n^2)$, där *n* representerar storleken av indata, och funktionen n^2 inom *O()* ger komplexiteten av funktionen beroende på indata *(Huang, 2020)*. Ett par olika tidskomplexiteter ordnat från bästa till värsta är; O(log(N)), O(N), O(N*log(N)) och $O(N^2)$. Rätt tidskomplexitet för en specifik algoritm kan hittas genom att leta efter konvergens när division sker med rätt förutsagda tidskomplexitet. Det vill säga att om tiderna det tar för algoritmen att köras klart vid olika tillfällen och med olika storlekar på inmatade värden divideras med rätt vald tidskomplexitet, så kommer alla dessa tider konvergera mot samma konstant.

2.2 Bubble Sort

Algoritmen *bubble sort* fungerar på så sätt att om två närliggande element är placerade i fel ordning, så byter de plats *(GeeksForGeeks, 2022)*. Sorteringen startar vid de första två elementen, och de jämförs för att kolla om de ska byta plats. Vid nästa tillfälle jämförs element två och tre, sedan tre och fyra, och så vidare. Det finns olika sätt att implementera *bubble sort* på, bland annat ett enklare sätt, och ett mer optimerat sätt. I den optimerade versionen stoppas algoritmen om det inte sker ett byte av platser i den inre *loopen*. För att ta fram det bästa fallet för *bubble sort* används en sorterad lista, detta då ingen tid behöver spenderas på att byta platser på element. I det bästa fallet blir *bubble sorts* tidskomplexitet O(N). Det värsta fallet för *bubble sort* tas fram genom att skicka en lista som är sorterad i omvänd ordning. Då måste algoritmen gå genom hela listan och sortera om alla element. Tidskomplexiteten för det värsta fallet blir $O(N^2)$. För det genomsnittliga fallet kommer *bubble sort* behöva sortera om ett antal av elementen i listan, detta ger samma tidskomplexitet som för det värsta fallet, det vill säga $O(N^2)$ *(GeeksForGeeks, 2022)*.

2.3 Quick Sort

Algoritmen *quick sort* är en så kallad *Divide and Conquer* algoritm, vilket betyder att den delar upp problemet i färre delproblem för att lösa dessa problem var för sig *(GeeksForGeeks, 2022)*. I *quick sort* väljs ett specifikt element ur listan ut, ett pivot-element, och sedan delas listan upp kring det valda elementet. Den viktigaste processen i algoritmen är något som kallas *partition*. Målet i *partition* är att med den givna listan sätta pivot-elementet på sin rätta position i en sorterad lista, och placera alla element med ett mindre värde före- samt elementen med ett större värde efter pivot-elementet. De två dellistorna sorteras sedan rekursivt. Det bästa fallet för *quick sort* är då man väljer ut det mittersta elementet i listan till pivot-elementet, detta då de två dellistorna som ska sorteras blir lika stora. Tidskomplexiteten för det bästa fallet är O(N*log(N)). Likt det bästa fallet för *quick sort* så får det genomsnittliga fallet samma tidskomplexitet. Det värsta fallet för algoritmen är då pivot-elementet väljs ut till det minsta eller största elementet i listan. I dessa situationer sker ingen bra uppdelning av listan och tidskomplexiteten blir $O(N^2)$ *(GeeksForGeeks, 2022)*.

2.4 Insertion sort

Algoritmen *insertion sort* är en sorteringsalgoritm som kan jämföras med hur man sorterar korten från en kortlek *(GeeksForGeeks, 2022)*. Listan som skickas till funktionen delas upp i en sorterad del och en osorterad del, där element från den osorterade listan sedan väljs ut för att läggas till på rätt plats i den sorterade listan. I det bästa fallet för *insertion sort* är listan redan sorterad, och då blir tidskomplexiteten O(N) då ingen tid spenderas till att byta plats på element. I det genomsnittliga- samt värsta fallet är listan antingen osorterad eller sorterad i omvänd ordning, och tidskomplexiteten går upp till $O(N^2)$ då mycket tid behöver spenderas till att sortera om elementen i listan *(GeeksForGeeks, 2022)*.

2.5 Linear search

Algoritmen *linear search* är en sekventiell sökalgoritm som börjar i en ände av en lista för att element för element leta efter ett önskat element *(GeeksForGeeks, 2022)*. Om det önskade elementet inte hittas så fortsätter sökningen till listans slut. Då varje element i listan endast träffas på en gång per sökning blir tidskomplexiteten densamma för både det värsta fallet och det genomsnittliga fallet, det vill säga O(N). Det värsta fallet för algoritmen är då det sista elementet i listan är det element som ska hittas, eller om elementet inte finns i listan. För det bästa fallet i *linear search* hittas det sökta elementet på första plats i listan och det ger en tidskomplexitet på O(1) *(GeeksForGeeks, 2022)*.

2.6 Binary Search

Algoritmen *binary search* är en sökalgoritm som söker efter ett specifikt element i en sorterad lista genom att dela upp sökintervallen i hälften*(GeeksForGeeks, 2022).* Det mittersta elementet i listan sätts som ett nyckelvärde, om nyckelvärdet är det sökta värdet så returneras det. Om värdet av det sökta elementet är mindre än nyckelvärdet minskas sökintervallet till den nedre halvan av listan, och om elementet har ett större värde än nyckelvärdet minskas sökintervallet till listans övre halva. Denna process fortsätter tills det sökta elementet är hittat eller tills sökintervallet är tomt *(GeeksForGeeks, 2022)*. Det bästa fallet för *binary search* är då nyckelvärdet är det sökta elementet, det vill säga det mittersta elementet i listan *(Sharma, 2022)*. Detta ger en tidskomplexitet på O(1). I det värsta fallet och det genomsnittliga fallet blir tidskomplexiteten densamma, O(log(N)). Det värsta

fallet för algoritmen är då det första eller sista elementet i listan ska hittas, eller om elementet inte finns i listan *(Sharma, 2022)*.

3. Design

Med laborationen tillkom en mall som innehöll både implementerade och deklarerade funktioner för *front-end* och *back-end*. Uppgiften blev där att implementera alla de algoritmer och funktioner som behövdes för laborationen samt implementera vissa av de funktioner som redan var deklarerade. Här presenteras de filer där förändringar och implementationer har utförts utefter mallen.

3.1 Algorithm.c

I filen algorithm.c fanns sorterings- och sökalgoritmerna för laborationen deklarerade. Det som utfördes här var implementation av alla algoritmerna samt tillägg av två funktioner för att hjälpa till med utförandet av *quick sort*. Det som kan vara värt att nämna här är att *quick sort* har implementerats rekursivt, samt att *bubble sort* har implementerats utifrån det optimala sättet med flaggor.

Algorithm.c är en *back-end* fil och inga visuella presentationer sker därifrån.

3.2 Analyze.c

I filen analyze.c fanns en funktion *benchmark* deklarerad från början. *Benchmark* har implementerats på ett sådant sätt att den tar emot vilken algoritm samt vilket fall som ska räknas på, och utifrån det anropas en utav de två funktionerna för tidtagning.

Det finns en tidtagningsfunktion för sorteringsfunktionerna, *timer_sort*, där används ett antal if-satser för att sätta upp olika situationer beroende på vilken algoritm och vilket fall som har blivit valt. Anrop sker till utvald funktion för att skapa rätt sorts listor, och tidtagningen startas innan anrop av vald sorteringsalgoritm. Tidtagningsfunktionen för sökalgoritmerna, *timer_search*, fungerar på samma sätt, if-satser för att sätta upp olika situationer beroende på vald algoritm och valt fall, samt anrop till rätt funktion för att skapa listor. Tidtagningen startas och därefter anropas den utvalda sökalgoritmen. För att beräkna tid har biblioteket *time.h* använts. Skapandet av listor och tidtagning sker ett antal gånger per list-storlek och algoritm för att ge säkrare resultat. Tidtagningen har också startats efter skapandet av listor för att inte få felaktiga värden på tider för algoritmerna.

Tre funktioner för skapandet av listor finns för att uppfylla alla olika fall. En funktion för sorterade listor, en funktion för slumpmässiga listor, och en funktion för omvänt sorterade listor.

Analyze.c är en *back-end* fil och inga visuella presentationer sker därifrån.

3.3 Ui.c

I filen ui.c fanns ett antal implementerade funktioner som hanterar det visuella vid exekvering av programmet, exempelvis felmeddelande om ett ogiltigt val görs från terminalen. Även menyn, *ui_menu* och hanteringen av vald algoritm och fall, *ui_run* var implementerade till viss del.

Menyn implementerades färdigt genom att skriva ut alla olika valmöjligheter och *ui_run* implementerades färdigt genom att lägga till anrop till *print*-funktionen, *benchmark*, och *calc* för alla möjliga val, med respektive parametrar.

Fem funktioner för att göra utskrivning av resultat snyggare lades till, *getAlgorithm, getCase,* och *getOutput* för att skicka rätt rubriker att skriva ut i tabellerna i terminalen, samt *print_list* och *print_data* för att skriva ut de fullständiga tabellerna. En funktion *calc* implementerades också för beräkning av tidskomplexiteter, och därifrån anropas *print_data* för att skriva ut tabellerna.

Ui.c är en *front-end* fil vilket innebär att visuella presentationer sker därifrån.

4. Resultat

För att bekräfta att algoritmer och andra funktioner i laborationen hade implementerats rätt krävdes flera provkörningar av programmet för att säkerställa rimliga resultat kopplat till tidigare förutsägelser. Här presenteras resultaten från varje algoritm och dess bästa, värsta, samt genomsnittliga fall.

4.1 Bubble Sort

Till algoritmen *bubble sort* skickades sex olika storlekar på listor. För varje storlek på lista utfördes tio iterationer, det vill säga att tio olika listor testades för *bubble sort* algoritmen för varje storlek på lista och under varje tidtagning för att få ett bra genomsnitt.

För det bästa fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är O(n). Tiden samt storleken på listan plottas på en graf, där visade resultaten en linjär ökning, se *Bubble sort best case* i Figur 2. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n), se *Bubble sort: best* i Figur 1.

För det värsta fallet skapades en lista som var sorterad i omvänd ordning. Tidskomplexitet för detta fall är $O(n^2)$. Tiden samt storleken på listan plottades på en graf, där visade resultaten en exponentiell ökning, se *Bubble sort worst case* i Figur 2. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för $O(n^2)$, se *Bubble sort: worst* i Figur 1.

För det genomsnittliga fallet skapades en lista med slumpmässiga tal mellan noll och listans storlek. Tidskomplexiteten för det genomsnittliga fallet är likt det värsta fallet $O(n^2)$. Tiden samt storleken på listan plottades på en graf, där visade resultaten en exponentiell ökning, se *Bubble sort average case* i Figur 2. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för $O(n^2)$, se *Bubble sort: average* i Figur 1.

Bubble sort : best

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000004	5.374670e-08	8.615566e-09	1.682728e-11
1024	0.000007	4.682893e-08	6.755986e-09	6.597643e-12
2048	0.000011	4.113445e-08	5.394951e-09	2.634254e-12
4096	0.000021	4.193633e-08	5.041777e-09	1.230903e-12
8192	0.000039	4.274718e-08	4.743934e-09	5.790935e-13
16384	0.000084	4.988784e-08	5.140925e-09	3.137771e-13

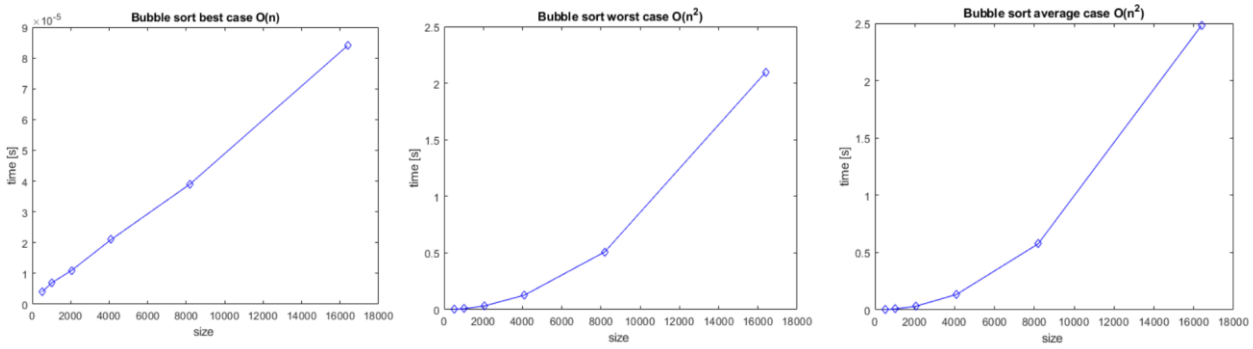
Bubble sort : worst

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.001904	2.320277e-05	3.719392e-06	7.264437e-09
1024	0.007892	5.342444e-05	7.707517e-06	7.526872e-09
2048	0.031668	1.178999e-04	1.546306e-05	7.550321e-09
4096	0.127245	2.583969e-04	3.106566e-05	7.584390e-09
8192	0.509713	5.606660e-04	6.222078e-05	7.595310e-09
16384	2.095132	1.240923e-03	1.278767e-04	7.804974e-09

Bubble sort : average

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.002093	2.550190e-05	4.087940e-06	7.984258e-09
1024	0.008059	5.455254e-05	7.870268e-06	7.685808e-09
2048	0.031420	1.169744e-04	1.534167e-05	7.491048e-09
4096	0.134388	2.729026e-04	3.280961e-05	8.010158e-09
8192	0.575990	6.335689e-04	7.031129e-05	8.582921e-09
16384	2.480248	1.469023e-03	1.513823e-04	9.239645e-09

Figur 1: Tid och tidskomplexitet för alla fallen av *bubble sort*.



Figur 2: Plottade grafer för bästa, värsta, och genomsnittliga fallen i *bubble sort*.

4.2 Quick Sort

Till algoritmen *quick sort* skickades sex olika storlekar på listor. För varje storlek på lista utfördes tio iterationer, det vill säga att tio olika listor testades för *quick sort* algoritmen för varje storlek på lista och under varje tidtagning för att få ett bra genomsnitt.

För det bästa fallet skapades en slumpmässig lista med tal mellan noll och listans storlek. Tidskomplexiteten för detta fall är O(n*log(n)). Tiden samt storleken på listan plottas på en graf, där visade resultaten en ökning enligt O(n*log(n)), se *Quick sort best case* i Figur 4. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n*log(n)), se *Quick sort: best* i Figur 3.

För det värsta fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är $O(n^2)$. Tiden samt storleken på listan plottas på en graf, där visade resultaten en exponentiell ökning, se *Quick sort worst case* i Figur 4. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för $O(n^2)$, se *Quick sort: worst* i Figur 3.

För det genomsnittliga fallet skapades en slumpmässig lista med tal mellan noll och listans storlek. Tidskomplexiteten för detta fall är O(n*log(n)). Tiden samt storleken på listan plottas på en graf, där visade resultaten en ökning enligt O(n*log(n)), se *Quick sort average case* i Figur 4. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n*log(n)), se *Quick sort: average* i Figur 3.

Quick sort : best

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000083	1.007289e-06	1.614679e-07	3.153670e-10
1024	0.000163	1.101694e-06	1.589408e-07	1.552156e-10
2048	0.000432	1.609499e-06	2.110923e-07	1.030724e-10
4096	0.000981	1.991576e-06	2.394364e-07	5.845614e-11
8192	0.002160	2.376291e-06	2.637126e-07	3.219148e-11
16384	0.004485	2.656242e-06	2.737248e-07	1.670684e-11

Quick sort : worst

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000588	7.163765e-06	1.148348e-06	2.242866e-09
1024	0.002716	1.838173e-05	2.651922e-06	2.589768e-09
2048	0.011944	4.446681e-05	5.832004e-06	2.847658e-09
4096	0.047083	9.561261e-05	1.149499e-05	2.806393e-09
8192	0.187576	2.063274e-04	2.289750e-05	2.795105e-09
16384	0.729407	4.320194e-04	4.451945e-05	2.717251e-09

Quick sort : average

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000088	1.075725e-06	1.724381e-07	3.367932e-10
1024	0.000166	1.121294e-06	1.617685e-07	1.579770e-10
2048	0.000500	1.861685e-06	2.441676e-07	1.192224e-10
4096	0.000916	1.860075e-06	2.236268e-07	5.459637e-11
8192	0.001785	1.963163e-06	2.178651e-07	2.659486e-11
16384	0.004078	2.415474e-06	2.489137e-07	1.519249e-11

Figur 3: Tid och tidskomplexitet för alla fallen av *quick sort*.

Figur 4: Plottade grafer för bästa, värsta, och genomsnittliga fallen i *quick sort*.

4.3 Insertion Sort

Till algoritmen *insertion sort* skickades sex olika storlekar på listor. För varje storlek på lista utfördes tio iterationer, det vill säga att tio olika listor testades för *insertion sort* algoritmen för varje storlek på lista och under varje tidtagning för att få ett bra genomsnitt.

För det bästa fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är O(n). Tiden samt storleken på listan plottas på en graf, där visade resultaten en linjär ökning, se *Insertion sort best case* i Figur 6. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n), se *Insertion sort: best* i Figur 5.

För det värsta fallet skapades en omvänt sorterad lista. Tidskomplexiteten för detta fall är $O(n^2)$. Tiden samt storleken på listan plottas på en graf, där visade resultaten en exponentiell ökning, se *Insertion sort worst case* i Figur 6. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för $O(n^2)$, se *Insertion sort: worst* i Figur 5.

För det genomsnittliga fallet skapades en slumpmässig lista med tal mellan noll och listans storlek. Tidskomplexiteten för detta fall är $O(n^2)$. Tiden samt storleken på listan plottas på en graf, där visade resultaten en exponentiell ökning, se *Insertion sort average case* i Figur 6. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för $O(n^2)$, se *Insertion sort: average* i Figur 5.

Insertion sort : best

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000008	9.474367e-08	1.518736e-08	2.966281e-11
1024	0.000012	8.112754e-08	1.170423e-08	1.142991e-11
2048	0.000021	7.666756e-08	1.005526e-08	4.909797e-12
4096	0.000055	1.116133e-07	1.341866e-08	3.276041e-12
8192	0.000095	1.041618e-07	1.155951e-08	1.411074e-12
16384	0.000170	1.007450e-07	1.038174e-08	6.336512e-13

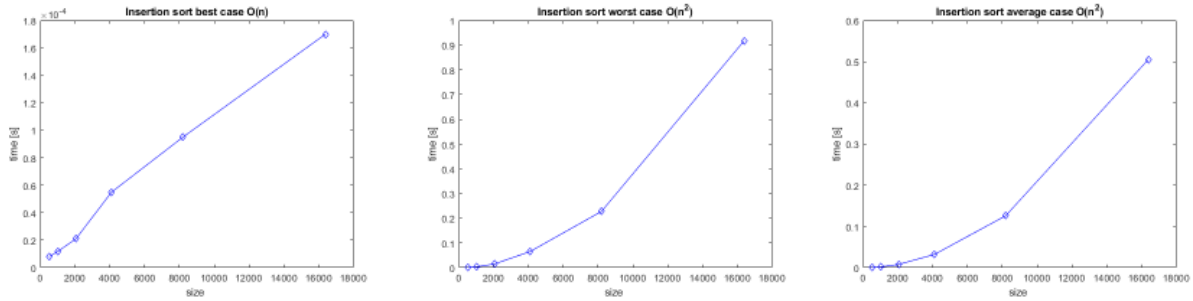
Insertion sort : worst

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000799	9.730812e-06	1.559844e-06	3.046570e-09
1024	0.003593	2.431814e-05	3.508366e-06	3.426139e-09
2048	0.015266	5.683445e-05	7.454071e-06	3.639683e-09
4096	0.065587	1.331882e-04	1.601249e-05	3.909299e-09
8192	0.227658	2.504157e-04	2.779027e-05	3.392367e-09
16384	0.917232	5.432664e-04	5.598341e-05	3.416956e-09

Insertion sort : average

size	time T(s)	T/nlogn	T/n	T/n^2
512	0.000432	5.261686e-06	8.434453e-07	1.647354e-09
1024	0.001983	1.342030e-05	1.936139e-06	1.890761e-09
2048	0.008102	3.016309e-05	3.956013e-06	1.931647e-09
4096	0.032166	6.532000e-05	7.853070e-06	1.917254e-09
8192	0.126135	1.387438e-04	1.539731e-05	1.879554e-09
16384	0.505147	2.991929e-04	3.083173e-05	1.881819e-09

Figur 5: Tid och tidskomplexitet för alla fallen av *insertion sort*.



Figur 6: Plottade grafer för bästa, värsta, och genomsnittliga fallen i *insertion sort*.

4.4 Linear Search

Till algoritmen *linear search* skickades sex olika storlekar på listor. För varje storlek på lista utfördes 10 000 iterationer, det vill säga att 10 000 olika listor testades för *linear search* algoritmen för varje storlek på lista och under varje tidtagning för att få ett bra genomsnitt.

För det bästa fallet skapas en sorterad lista. Tidskomplexiteten för detta fall är $O(1)$. Tiden samt storleken på listan plottas på en graf, där visade resultaten en ökning enligt $O(1)$, se *Linear search*

best case i Figur 8. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(1), se *Linear search: best* i Figur 7.

För det värsta fallet skapades en omvänt sorterad lista. Tidskomplexiteten för detta fall är O(n). Tiden samt storleken på listan plottas på en graf, där visade resultaten en linjär ökning, se *Linear search worst case* i Figur 8. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n), se *Linear search: worst* i Figur 7.

För det genomsnittliga fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är O(n). Tiden samt storleken på listan plottas på en graf, där visade resultaten en linjär ökning, se *Linear search average case* i Figur 8. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(n), se *Linear search: average* i Figur 7.

Linear searcch : best

size	time T(s)	T/1	T/n	T/(n/2)
512	0.000002	1.846688e-06	3.606813e-09	7.213625e-09
1024	0.000002	1.608890e-06	1.571181e-09	3.142363e-09
2048	0.000002	2.059669e-06	1.005698e-09	2.011396e-09
4096	0.000002	1.892947e-06	4.621453e-10	9.242906e-10
8192	0.000002	2.023151e-06	2.469666e-10	4.939333e-10
16384	0.000002	2.323373e-06	1.418074e-10	2.836149e-10

Linear searcch : worst

size	time T(s)	T/1	T/n	T/(n/2)
512	0.000004	4.132095e-06	8.070499e-09	1.614100e-08
1024	0.000007	6.656724e-06	6.500707e-09	1.300141e-08
2048	0.000012	1.234894e-05	6.029756e-09	1.205951e-08
4096	0.000025	2.536037e-05	6.191496e-09	1.238299e-08
8192	0.000045	4.466854e-05	5.452703e-09	1.090541e-08
16384	0.000086	8.554049e-05	5.220977e-09	1.044195e-08

Linear searcch : average

size	time T(s)	T/1	T/n	T/(n/2)
512	0.000003	2.728799e-06	5.329686e-09	1.065937e-08
1024	0.000008	7.833948e-06	7.650340e-09	1.530068e-08
2048	0.000014	1.361131e-05	6.646147e-09	1.329229e-08
4096	0.000024	2.368516e-05	5.782509e-09	1.156502e-08
8192	0.000048	4.846195e-05	5.915766e-09	1.183153e-08
16384	0.000082	8.200432e-05	5.005146e-09	1.001029e-08

Figur 7: Tid och tidskomplexitet för alla fallen av *linear search*.

Figur 8: Plottade grafer för bästa, värsta, och genomsnittliga fallen i *linear search*.

4.5 Binary Search

Till algoritmen *binary search* skickades sex olika storlekar på listor. För varje storlek på lista utfördes 10 000 iterationer, det vill säga att 10 000 olika listor testades för *binary search* algoritmen för varje storlek på lista och under varje tidtagning för att få ett bra genomsnitt.

För det bästa fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är O(1). Tiden samt storleken på listan plottas på en graf, där visade resultaten en ökning enligt O(1), se *Binary search best case* i Figur 10. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(1), se *Binary search: best* i Figur 9.

För det värsta fallet skapades en omvänt sorterad lista. Tidskomplexiteten för detta fall är O(log(n)). Tiden samt storleken på listan plottas på en graf, där visade resultaten ökning enligt O(log(n)), se *Binary search worst case* i Figur 10. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(log(n)), se *Binary search: worst* i Figur 9.

För det genomsnittliga fallet skapades en sorterad lista. Tidskomplexiteten för detta fall är O(log(n)). Tiden samt storleken på listan plottas på en graf, där visade resultaten en ökning enligt O(log(n)), se *Binary search average case* i Figur 10. Beräkningen av tidskomplexitet visade på konvergens mot en konstant för O(log(n)), se *Binary search: average* i Figur 9.

Binary search : best

size	time T(s)	T/1	T/n	T/logn
512	0.000002	1.776296e-06	3.469328e-09	2.847393e-07
1024	0.000002	2.245721e-06	2.193086e-09	3.239890e-07
2048	0.000002	2.069144e-06	1.010324e-09	2.713767e-07
4096	0.000002	2.174643e-06	5.309186e-10	2.614455e-07
8192	0.000002	2.451571e-06	2.992641e-10	2.720669e-07
16384	0.000002	2.395942e-06	1.462367e-10	2.469010e-07

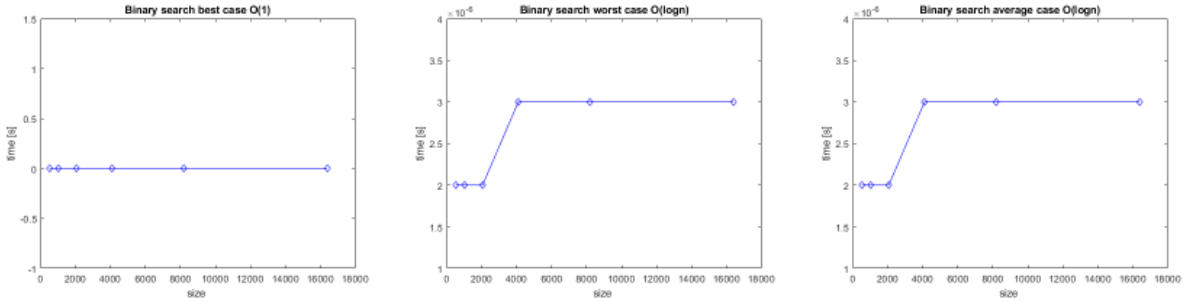
Binary search : worst

size	time T(s)	T/1	T/n	T/logn
512	0.000002	1.958856e-06	3.825890e-09	3.140035e-07
1024	0.000002	2.137683e-06	2.087581e-09	3.084025e-07
2048	0.000002	2.282653e-06	1.114577e-09	2.993793e-07
4096	0.000003	2.662360e-06	6.499903e-10	3.200811e-07
8192	0.000003	2.514345e-06	3.069269e-10	2.790334e-07
16384	0.000003	2.541515e-06	1.551218e-10	2.619022e-07

Binary search : average

size	time T(s)	T/1	T/n	T/logn
512	0.000002	2.427517e-06	4.741244e-09	3.891296e-07
1024	0.000002	2.255280e-06	2.202422e-09	3.253681e-07
2048	0.000002	2.446606e-06	1.194632e-09	3.208824e-07
4096	0.000003	2.964235e-06	7.236902e-10	3.563739e-07
8192	0.000003	2.541673e-06	3.102628e-10	2.820661e-07
16384	0.000003	2.721071e-06	1.660810e-10	2.804054e-07

Figur 9: Tid och tidskomplexitet för alla fallen av *binary search*.



Figur 10: Plottade grafer för bästa, värsta, och genomsnittliga fallen i *binary search*.

5. Analys

Under resultatdelen presenteras resultaten för de olika sorterings- och sökalgoritmerna. De olika algoritmerna betedde sig lite olika beroende på bästa, värsta, och genomsnittliga fall, och vissa var väldigt jämna i resultaten oberoende på vilka listor som skickades in. Här analyseras resultaten från de olika algoritmerna och vissa slutsatser dras.

De resultat som vi fick i *bubble sort* stämde med de teoretiska tidskomplexiteterna för de tre olika fallen, där tidskomplexiteten för bästa fallet är O(n), värsta fallet $O(n^2)$, samt genomsnittliga fallet $O(n^2)$. *Insertion sort* har samma teoretiska tidskomplexitet som *bubble sort*, vilket är O(n) för bästa fallet och $O(n^2)$ för både värsta och genomsnittliga fallet. När det kommer till resultaten som vi fick av *insertion sort* jämfört med de teoretiska tidskomplexiteterna så stämmer de överens med varandra. Där fick vi en linjär ökning i bästa fallet där tidskomplexiteten ska vara O(n), och exponentiell ökning i värsta och genomsnittliga fallet, vilket också stämmer med den teoretiska tidskomplexiteten $O(n^2)$. Algoritmen *quick sort* var svårast att implementera på grund av att när man ska testa de olika fallen så ska pivoten ha olika värde för varje fall. Till exempel för bästa fallet för *quick sort* så ska pivoten hamna i mitten, men i programmet hade vi svårt att ändra värde på pivot för varje fall. I stället genererade vi en lista med slumpmässiga tal, vilket betyder att bästa fallet för *quick sort* kan ge olika resultat varje gång man kör koden. Detta på grund av att man aldrig vet vilken siffra som genereras först i listan och blir vald som pivot. Bästa fallet för *quick sort* ger dock rimliga värden som stämmer med den teoretiska tidskomplexiteten som är O(nlogn), som även stämmer överens med de resultat vi fick, samt grafen som plottades; se Figur 4. För att undersöka värsta fallet för *quick sort* skapades en sorterad lista, där pivoten blir lika med ett. Den teoretiska tidskomplexiteten för värsta fallet är $O(n^2)$, vilket betyder att man kommer att förvänta sig en exponentiell ökning med tiden, och det stämmer överens med de tiderna samt grafen vi fick utifrån våra resultat; se Figur 3 och Figur 4. För det genomsnittliga fallet av *quick sort* skapades en slumpmässig lista, och pivoten kunde antingen hamna i början, i mitten eller i slutet av listan. Därav varierar resultaten beroende på vad pivoten är. I de flesta fallen stämmer resultaten för det genomsnittliga fallet med den teoretiska tidskomplexiteten vilket är O(nlogn), som visas i Figur 3 och Figur 4. Utifrån en analys av de tre sorteringsalgoritmerna kan vi dra slutsatsen att *quick sort* är den snabbaste sorteringsalgoritmen i de flesta fallen, men samtidigt svårast att implementera.

Sökalgoritmerna *binary search* och *linear search* skiljer sig ifrån sorteringslagoritmerna, de har i stället en uppgift att hitta ett specifikt värde i en lista. *Linear search* går igenom hela lista från första elementet till sista elementet för att försöka hitta det sökta värdet. Bästa fallet fick man genom att söka efter värdet som finns på första plats i listan, och då blir tiden konstant eftersom *linear search* hittar det sökta värdet direkt. Detta stämmer med den teoretiska tidskomplexiteteten O(1), se Figur 7 och Figur 8. Värsta fallet för *linear search* är när det sökta värdet ligger på sista platsen i listan, och då ökar tiden linjär. Detta på grund av att hela listan behöver sökas igenom, vilket ger en tidskomplexitet O(n) som stämmer med den teoretiska tidskomplexiteten. Det genomsnittliga fallet sker när det sökta värdet ligger i mitten av listan, vilket ger ett lite snabbare resultat än för det värsta fallet, men lite långsammare tid än det bästa fallet. Tidskomplexiteten för det genomsnittliga fallet är O(n/2), vilket stämmer med våra resultat samt grafer; se Figur 7 och Figur 8. *Binary search* har bättre resultat än *linear search* i de flesta fallen. I det bästa fallet uppnås samma tidskomplexitet som *linear search*'s bästa fall. I det bästa fallet söker man efter ett värde som ligger på första plats i listan, vilket ger ett konstant värde. Tidskomplexiteten för det här fallet är alltså precis som bästa fallet för *linear search* O(1), och det stämmer med resultaten och graferna; se Figur 9 och Figur 10. När det kommer till värsta och genomsnittliga fallet för *binary search* så har dem båda samma teoretiska

tidskomplexitet, vilket är O(log n). Detta stämmer också in på de värden som vi fick; se Figur 9 och Figur 10. Utifrån en analys av de två sökalgoritmerna kan vi dra slutsatsen att *binary search* är den snabbaste sökalgoritmen i de flesta fallen, men likt *quick sort* så är den även svårare att implementera.

6. Problem

Under laborationens gång stöttes det på ett antal olika problem. Genom samarbete och funderande kunde dessa problem lösas för att få fram det slutgiltiga programmet.

Det första problemet som stöttes på var beräkningen av tidskomplexiteter. Fel värden på listornas storlekar skickades till beräkningarna av tidskomplexiteterna vilket gav felaktiga värden ut. Felet låg i att storleken på listorna som skickades med i beräkningarna inte ändrades i varje varv av beräkningar. Första listornas storlekar var på 512 element, och sista listornas storlekar var på 16 384 element. Det värdet som skickades till beräkningarna till en början var enbart 512, vilket såklart gav felaktiga resultat för de listorna med fler element än 512. Trots att detta var ett relativt enkelt problem att lösa, tog det ganska mycket tid att hitta vart problemet låg. Anledningen till detta var att vi var allt för fokuserade på att felet låg på något mer komplicerat ställe i koden, exempelvis vid tidtagningen. Här lärde vi oss att kolla igenom alla funktioner steg för steg och inse att problemen kan uppstå vart som helst, så det är viktigt att kontrollera alla delar av programmet.

Ett mindre problem, som löstes relativt snabbt, var implementeringen av *quick sort*. Endast två parametrar fick skickas till den ursprungliga funktionen, och vi ville skicka med tre. Detta löstes på så sätt att den ursprungliga *quick sort* fick anropa en annan *quick sort* funktion som i sin tur kunde ta emot även den tredje parametern. Därifrån sköttes sedan *quick sort* algoritmen.

Det sista problemet vi stötte på var att vissa av sökalgoritmerna inte visade tider efter tidtagningen. Detta berodde på att vi i början hade använt samma antal iterationer för sökalgoritmerna som för sorteringsalgoritmerna. Sökalgoritmerna går mycket snabbare än sorteringsalgoritmerna, och på grund av det var det ingen bra idé att ha lika många iterationer för båda, då hade det antingen tagit orimligt lång tid att köra sorteringsalgoritmerna, eller väldigt kort tid att köra sorteringsalgoritmerna. Vi löste problemet genom att öka antalet iterationer för sökalgoritmerna från tio till 10 000. Detta gav bättre värden då tidtagningen hann köras lite längre. Vi ändrade också visningen av tid i terminalen så att fler decimaler för tiden blev med.

7. Sammanfattning

Laborationen för funktionsanalyser har gått ganska smidigt och bra, med enbart ett fåtal problem på vägen. Vi har fått en bättre förståelse för hur tidtagning kan användas och implementeras i C, samt hur några sorterings- och sökalgoritmer faktiskt fungerar och hur man kan ta reda på i vilka fall respektive algoritm passar bäst att implementeras. I denna laboration fick vi fram resultatet att quick sort är den sorteringsalgoritm utav de tre testade som är snabbast i de flesta fallen, samt att binary search i de flesta fallen genomför sökning snabbare än linear search. Tidskomplexitet och Big-O var även nytt för oss och laborationen gjorde det väldigt lätt och tydligt att förstå vad de två begreppen handlar om.

Referenslista

Geeks for geeks. (Juli, 2022). Time Complexity and Space Complexity. Hämtat från https://www.geeksforgeeks.org/time-complexity-and-space-complexity/

Huang S. (Januari, 2020). What is Big O Notation Explained: Space and Time Complexity. freeCodeCamp. Hämtat från https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/

Geeks for geeks. (November 2022). Bubble Sort Algorithm. Hämtat från https://www.geeksforgeeks.org/bubble-sort/

Geeks for geeks. (September 2022). QuickSort. Hämtat från https://www.geeksforgeeks.org/quick-sort/

Geeks for geeks. (Oktober 2022). Insertion Sort. Hämtat från https://www.geeksforgeeks.org/insertion-sort/

Geeks for geeks. (September 2022). Linear Search Algorithm. Hämtat från https://www.geeksforgeeks.org/linear-search/

Geeks for geeks. (Oktober 2022). Binary Search. Hämtat från https://www.geeksforgeeks.org/binary-search/

Sharma R. (Oktober 2022). Binary Search Algorithm: Function, Benefits, Time & Space Complexity. upGrad. Hämtat från https://www.upgrad.com/blog/binary-search-algorithm/