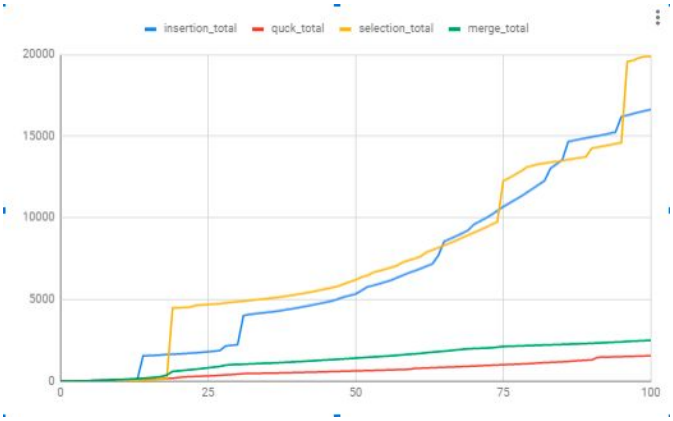
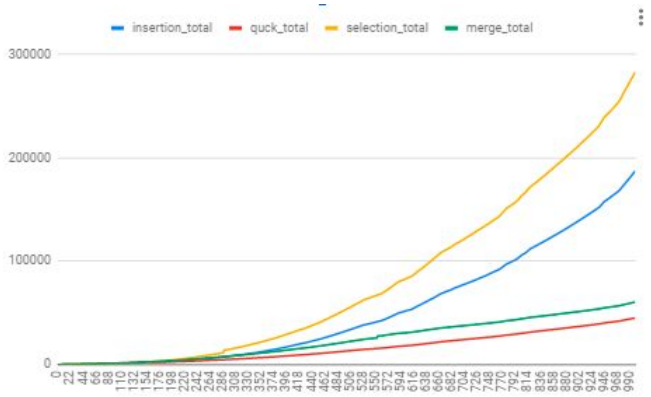
Oblig 3 IN2010 Jørgen Lunder og William Resvoll Skaug

Oppgave 1)

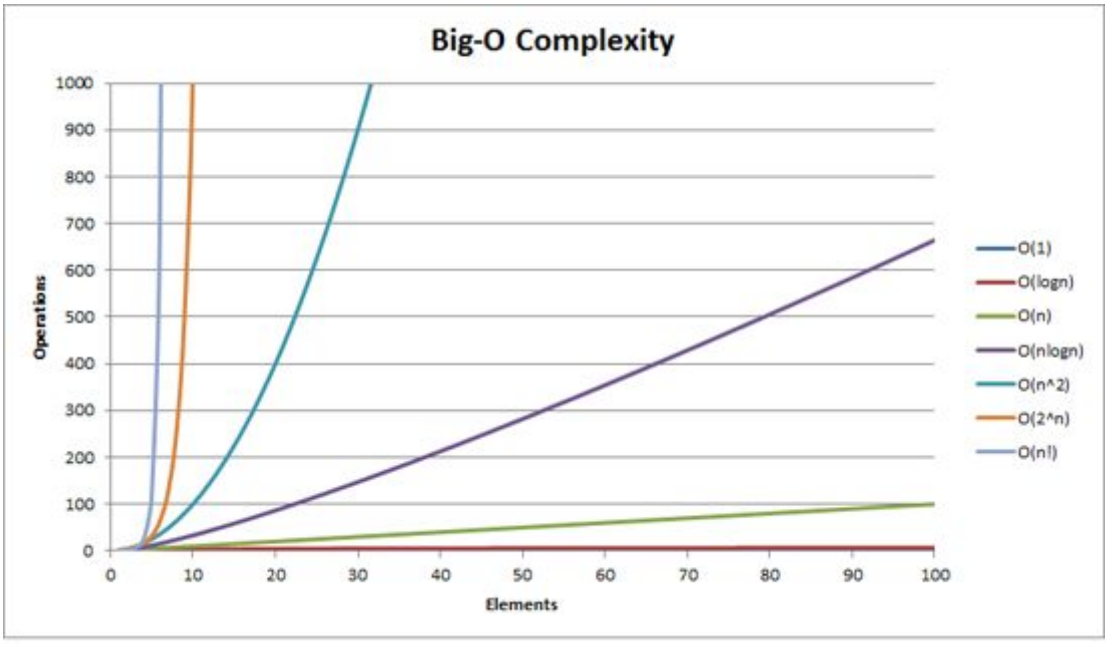
I hvilken grad stemmer kjøretiden overens med kjøretidsanalysene (store O) for de ulike algoritmene?

kjøretid ved n=100 og n=1000





Her kan en se de diverse kjøretidene ved n= 100 og 1000. En kan se en tydelig kurve hvor quick og merge er veldig like i kjøretid, og at selection er litt trege enn insertion.



sammenlignet med grafene over kan en se at

selection og insertion tilsvarer veldig likt O(n^2)

quick og merge tilsvarer veldig likt O(log(n))

Hvilke sorteringsalgoritmer utmerker seg positivt når n er veldig liten? Og når n er veldig stor?

Når n er veldig stor vil de raskeste algoritmene være best, i vårt tilfelle quick og merge. De sparer veldig mye tid ved en stor n.

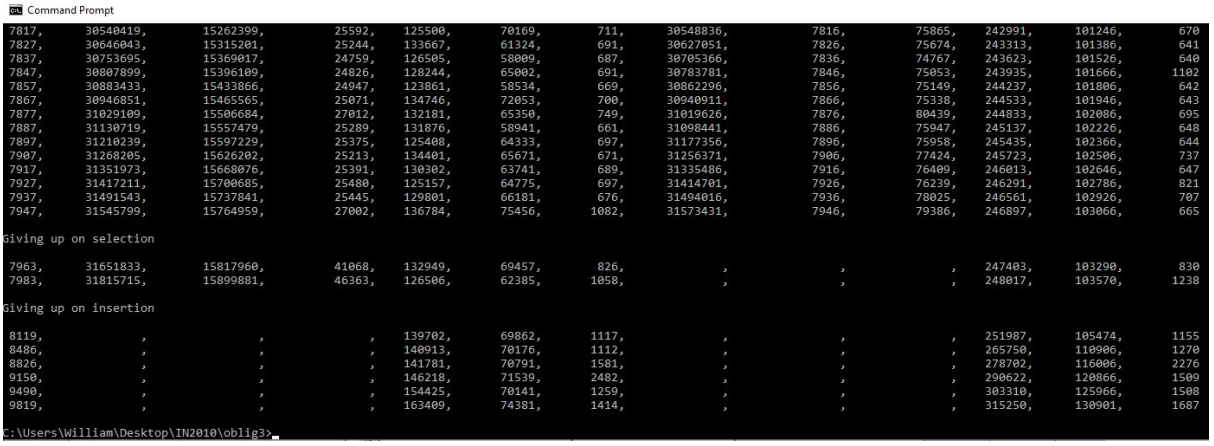
Ved en liten n vil alle algoritmene fungere fint men ved feks n = 100 er det fortsatt stor forskjell fra quick og merge sitt antall i swaps og comparison (veldig få), til selection og insertion sitt antall i swaps og comparisons som er mye flere en de to førstnevnte sine.

Ved n=100

Ins_cmp	ins_swp	qui_cmp	qui_swp	sel_cmp	sel_swp	mer_cmp	mer_swp
187466	88950	34498	17500	166650	4950	87214	30267

Hvilke sorteringsalgoritmer utmerker seg positivt for de ulike inputfilene?

Ved n > 1000 fungerer quick og merge mye bedre enn de 2 trege algoritmene.
Ved rundt n = 8000 stoppet programmet for de 2 tregeste, og ved n > 1000 så var de veldig trege i forhold.



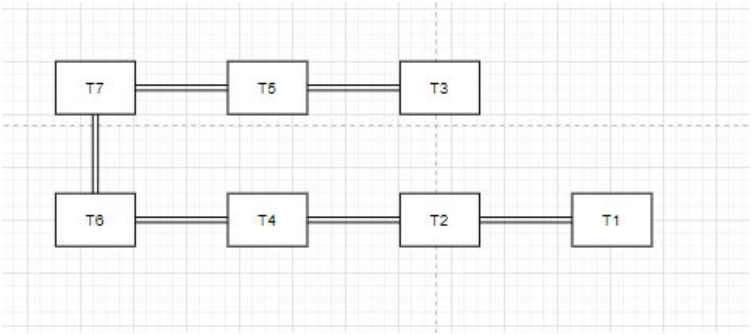
Har du noen overraskende funn å rapportere?

Enkelte steg i selection og insertion algoritmene ble det plutselig et stort “hopp”i tiden de brukte. I bilde under kan en se eksempel ved n=100, så går selection fra å bruke 29 ns i ett steg, til å bruke 4349 i det neste.

C	D	E	F	G	H	I	J	K
insertion_swaps	insertion_time	quick_cmp	quick_swaps	quick_time	selection_cmp	selection_swaps	selection_time	merge_cmp
0	2	1	0	4	0	0	1	1
0	1	1	0	1	0	0	1	1
1	20	4	1	3	1	1	2	10
1	2	8	4	3	3	2	2	22
2	2	9	4	4	6	3	3	32
2	3	15	9	6	10	4	3	47
6	3	17	5	10	15	5	3	60
8	3	20	10	5	21	6	5	73
15	7	33	11	7	28	7	3	84
20	3	31	13	9	36	8	5	102
27	6	33	14	6	45	9	5	118
32	7	39	15	6	55	10	6	134
35	8	42	23	7	66	11	6	148
45	10	55	28	9	78	12	8	164
48	1477	58	31	16	91	13	8	178
55	12	57	29	9	105	14	10	192
57	14	72	41	10	120	15	14	204
69	26	84	51	24	136	16	[illegible]	225
77	28	78	37	25	153	17	29	244
79	17	82	48	12	171	18	4349	263
98	16	110	46	46	190	19	[illegible]	280
116	23	122	52	39	210	20	16	299
136	24	139	67	16	231	21	18	316
149	25	112	46	13	253	22	115	333

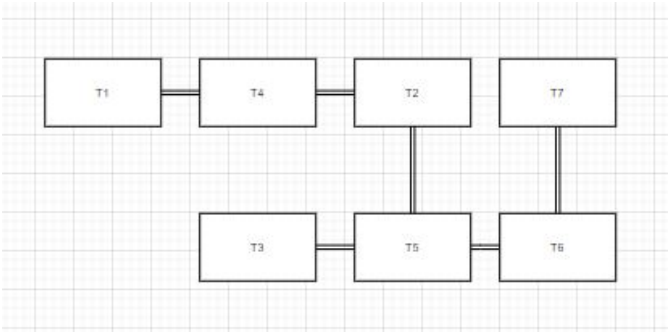
Oppgave 2)

1. For å finne den øvre grensen for hvor mye kabel selskapet trenger sorterte vi tabellen etter lengder med den lengste først. Så koblet vi sammen hver og en nedover, utenom de som ble koblet sammen to veier. Med denne fremgangsmåten fant vi den lengste veien å koble sammen tårnene, uten at noen tårn er koblet sammen ekstra unødvendige ganger, som i en sykel. Med denne metoden fikk vi at øvre grense for kabel er 61 km.



17
10
11
7
6
10

2. For å finne den nedre grensen for hvor mye kabel som trengs sorterte vi etter korteste først og brukte ellers samme fremgangsmåte som i (1). Dette ga en total på 35 km kabel. Dette er Dijkstra's algoritme. Vi kunne også brukt Floyd-Warshal's algoritme, dersom vi skulle funnet raskeste veien mellom alle nodene.



Oppgave 3)

1.

```
Algorithm 1
Input: Et tall n
1 Procedure Algo1(n)
2 |  return n + 1
```

O-Notasjon → O(1)
Denne algoritmen gjør steget 1 gang med tallet n.

2.

```
Algorithm 2
Input: To positive tall m og n
1 Procedure Algo2(m, n)
2 |  output = 0
3 |  for i = 1 to m do
4 |  |  for j = 1 to n do
5 |  |  |  output = Algo1(output)
6 |  |  end
7 |  end
8 |  return output
```

O-Notasjon → O(m*n)
Denne algoritmen går gjennom alle tallene for i, og for hver i går gjennom j. Kaller på algol 1 som gjør 1 operasjon (O(1)).
Dermed blir det en dobbel for løkke med m*n operasjoner

3.

```
Algorithm 3
Input: Et positivt tall n
1 Procedure Algo3(n)
2 |  output = 0
3 |  j = n
4 |  while j > 1 do
5 |  |  Algo1(output)
6 |  |  j = ⌊j/2⌋
7 |  end
8 |  return output
```

O-notasjon → O(log(n))
setter j til å være n. Går gjennom j større enn 1. Kaller på algol 1 som gjør 1 operasjon (O(1)). j halveres. Dermed logaritmisk tid pga halveringen.