Oppgave 1c)

I denne oppgaven har det blitt implementert følgende sorteringsalgoritmer:

- Quicksort med worstcase kjøretid: $O(n^2)$
- Insertion sort med worstcase kjøretid: $O(n^2)$
- Selection sort med worstcase kjøretid: $O(n^2)$
- Heapsort med worstcase kjøretid: $O(n\ logn)$

Alle algoritmene ble kjørt på filene med størrelse 10, 100 og 1000. De større random-filene ble kjørt med quick og heap. De andre nearly_sorted-filene ble kjørt med insertion, quick og heap.

n:	Insertion:	Quicksort:	Selection:	Heapsort:
10	2	11	4	10
100	56	24	74	20
1000	409	78	1009	75

Tabell 1: viser kjøretiden fra eksperimentene utført på inputfilene med navn: random_10, random_100 og random_1000.

.

n:	Insertion:	Quicksort:	Selection:	Heapsort:
10	1	11	3	10
100	8	12	41	10
1000	6	42	606	69

Tabell 2: viser kjøretiden fra eksperimentene utført på inputfilene: nearly_sorted_10, nearly_sorted_100 og nearly_sorted_1000.

Tabellene 1) og 2) viser kjøretiden fra eksperimentene som ble utført på noen av inputfilene. Her er det lett å se at kjøretiden stemmer godt overens med kjøretidssanalysen for inputfilene med navn random_xx, men ikke helt med de nesten sorterte filene. I tabell 1) kan vi se at for n=10 så har Heapsort den «tregeste» kjøretiden mens andre algoritmer som insertion og quicksort har «raskere» kjøretid selv de har dårligere worstcase per teori. For n= 100 og n=1000 så ser vi at både insertion og selection bruker mer og mer tid, mens Heapsort bruker mindre tid sammenlignet med de andre algoritmene. Ett veldig interessant funn i denne oppgaven var kjøretiden til quicksort. Kjøretiden til denne algoritmen er avhengig av det pivot-elementet. Dersom dette elementet i midten av listen så får den en kjøretid på $O(n\ logn)$, og dersom den er på i starten av listen eller på slutten av listen så vil kjøretiden har kvadratisk tid. Vi valgte å **Math.random** til å bestemme ett tilfeldig verdi for pivot-elementet og på denne måten så sikret vi oss at denne algoritmen har sin beste kjøretid. Dette kan vi observere på tabell 1), nemlig kjøretiden til quicksort er omtrent likt som Heapsort.

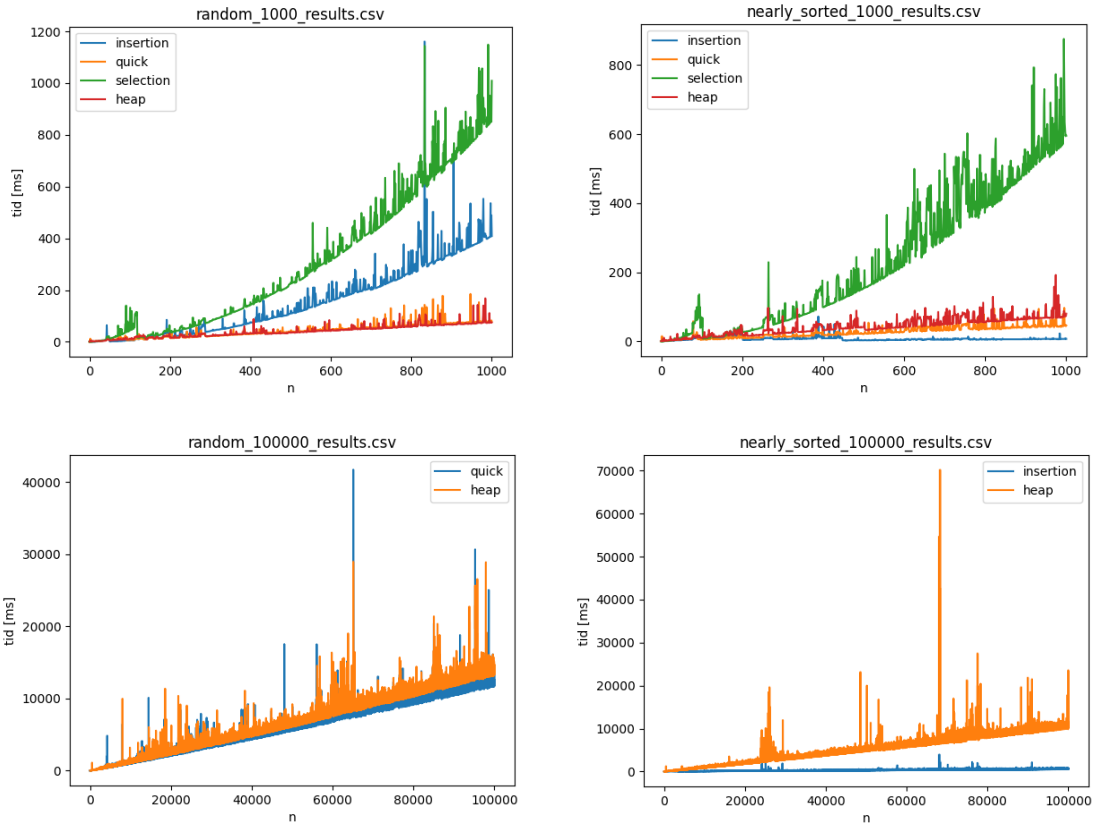
Ved å bruke programmet på de omtrent sorterte filene så det oppdaget noen interessante funn som gjorde at kjøretiden ikke stemte veldig godt med de worstcase tilfellene. Det første vi observerer er at Heapsort og selection oppfører seg som i tabell 1) og det er ikke noe overraskende at de to algoritmene er litt raskere på grunn av listen er omtrent ordnet. Men for insertion og quicksort så ser vi at kjøretiden er mye lavere. Quicksort har nesten halvert sine kjøretider fra tabell 1. Dette er naturlig siden denne algoritmen er gjør mye færre operasjoner enn ovenfor (se tabell 3). Det mest imponerende resultatet er nemlig insertion sin kjøretid. Det vakre med insertion algoritmen er at den har to loops, der den indre er en while loop med en gitt betingelse. Så som oftest så bryter den ut av den indre while-loopen og gjør færre operasjoner. Dette gjør at den kjører mye raskere. Som vi ser i tabell 2) så er den veldig rask på «nesten sorterte» lister.

Filnavn:	Antall sammenligninger:	Antall bytter:

random_1000	486775	241892
nearly_sorted_1000	3931	467

Tabell 3: viser antall sammenligner og antall bytter for insertion-algoritmen med n =1000.

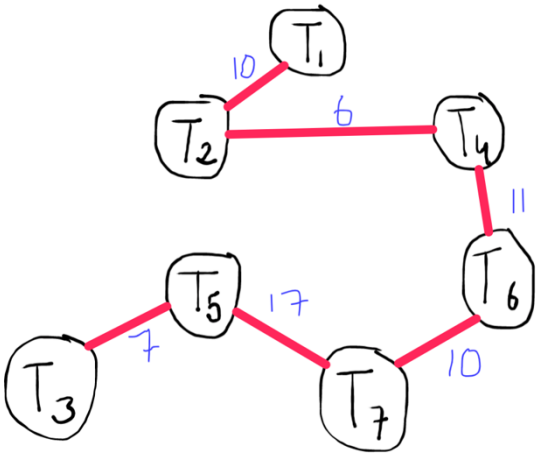
For å konkludere så kan vi si at sorteringsalgoritmer som quicksort og heapsort egner seg veldig godt for å sortere store datamengder med helt tilfeldig orden (se på plottene under). Men dersom datamengden er nesten sortert i en ordnet rekkefølge, så lønner det seg absolutt å bruke insertion. Og for små datamengden så er aller enklest å bruke insertion eller selection siden de er veldig enkle å implementere og raske for mindre n.



Oppgave 2: Mobilnettverk

1.Din første oppgave er å beregne en øvre grense for hvor mye kabel selskapet trenger. Hva er kostnaden av den dyreste måten å koble sammen alle signaltårn på? Oppgi svaret ditt som et tall.

Øvre grense for kabel som trengs, vil være 61 km. Den dyreste måten å koble sammen alle signaltårn på er å koble signaltårnene på følgende måte:

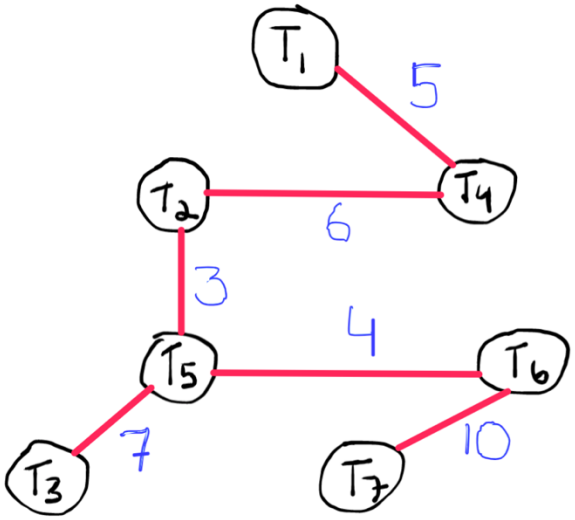


Her er noder Signaltårn, og kanten mellom dem er kabelen som forbinder dem, og kantvekten er lengden på den nødvendige kabelen.

Her vil den totale kostnaden være summen av alle kantene som forbinder to tårn, dvs. 10 + 6 + 11 + 10 + 17 + 7 = 61 km. Ovenstående bilde viser den sammenhengende grafen over signaltårnene. Dette kan enkelt beregnes ved hjelp av Maksimalt spenntre.

2.Hva er den billigste måten å koble sammen alle signaltårn på? Oppgi dette som et tall. Videre, oppgi hvilken algoritme du brukte, samt hvordan du formaliserte problemet. Til slutt, oppgi hvilke andre algoritmer som er blitt dekket i pensum, som man kunne ha brukt i stedet.

Den billigste måten å koble sammen alle signaltårn på er nedenfor:



Total kostnad for å koble på denne måten vil være garantert minimum, dvs. 5 + 6 + 3 + 4 + 10 + 7 = 35 km. Jeg brukte minimal spenntre - Grådig Algoritme for å beregne minimumskostnaden for å koble sammen alle signaltårnene. Problemet kan tenkes som en ikke-rettet graf der disse signaltårnene er noder og kantene mellom dem er kabelen som forbinder to signaltårn (som allerede er gitt i problemstillingen). Nå er problemet delt inn i et enklere problem der vi trenger å koble alle noder slik at den totale kostnaden for å koble dem er minimum / maksimum. Dette er et veldig kjent problem som enkelt kan løses ved hjelp av Minimum / Maksimalt spenntre (MST). Dette kan implementeres ved hjelp av Prims eller Kruskals algoritme.

Oppgave 3: O Notasjon

I denne oppgaven skal du approksimere kjøretiden til forskjellige algoritmer. Oppgi en kjøretidsanalyse ved bruk av O-notasjon for hver av de følgende algoritmene. Du trenger ikke å telle alle primitive steg, men begrunn svarene dine.

Algorithm 1

Input: Et tall n

```
1 Procedure Algo1(n)
2 |  return n + 1
```

Kjøretiden til denne algoritmen er konstant, dvs. O(1). Siden det tar 'n' som input og returnerer 'n + 1' (legger til +1 i input) som output som tar konstant tid til å gjøre.

Algorithm 2

Input: To positive tall m og n

```
1 Procedure Algo2(m, n)
2 |  output = 0
3 |  for i = 1 to m do          // Outer loop – kjøres m ganger
4 |  |  for j = 1 to n do       // Inner loop - kjøres n ganger
5 |  |  |  output = Algo1(output)   // tar konstant tid - O(1)
6 |  |  end
7 |  end
8 |  return output
```

Kjøretiden til denne algoritmen er O(m*n). Siden outer loop kjøres 'm' ganger og inner loop kjøres 'n' ganger. Dermed vil den totale tidskompleksiteten være - m*n + konstant. Endelig verdi av output variabelen vil være m*n. La oss si n = 5 og m = 5, så vil total kjøretid være:

```
for i=1 to 5:
        for j=1 to 5:
                output=Algo1(output)
        end
end
```

Etter hver iterasjon av outer loop, økes output verdien med n (her n=5).
Dermed etter 5 iterasjon dvs. i = 1,2,3,4,5 .. vil output være 1+2+3+4+5+ ....+25= 25 (n*m).

Algorithm 3

Input: Et positivt tall n

```
1 Procedure Algo3(n)
2 |  output = 0
3 |  j = n
4 |  while j > 1 do     // si at den er utført k ganger
5 |  |  Algo1(output)   // tar O(1) tid
6 |  |  j = ⌊j/2⌋       //gulveres verdien av j/2 dvs. gulv (2.5) = 2.
7 |  end
8 |  return output
```

Kjøretiden til denne algoritmen er $O(\log_2(n))$. Siden etter hver iterasjon, verdi av j er redusert til j/2, vil dermed maksimale totale iterasjoner være $\log_2(n)$ dvs. **$k = \log_2(n)$.**

$$T(n) = T(n/2) + C$$

Etter K-iterasjon:

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n$$

$$\Rightarrow \log_2(2^k) = \log_2(n)$$

$$\Rightarrow K \cdot \log_2 2 = \log_2(n)$$

$$\Rightarrow K = \log_2(n).$$

I beregningen ovenfor er C konstant og k er det totale antallet iterasjoner, dvs. total tidskompleksitet.