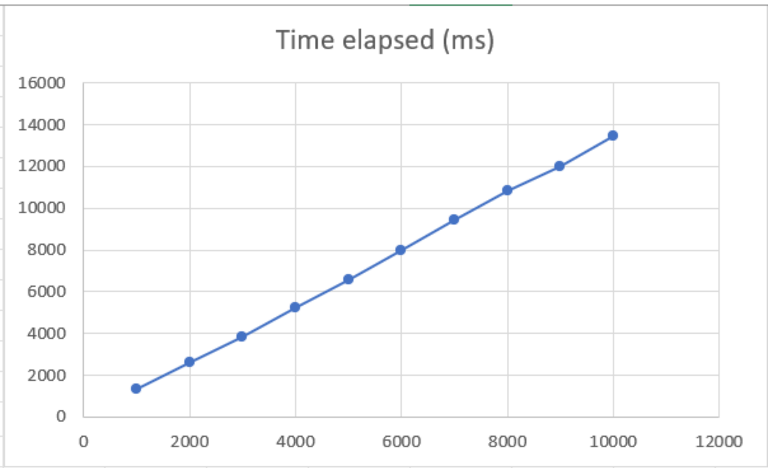
Øving 2, rekursiv programmering

Metode 1

Metode 1 bruker rekursjon for å gå gjennom vært eneste heltall fra n til 1. Siden metoden bare kaller seg selv 1 gang per lag, er rekursjonen lineær. Hvert lag i rekursjonen gjør et konstant arbeid på O(1) Basert på dette får vi en algoritme med tidskompleksitet på O(n)

Dette kan videre bekreftes ved å skissere en graf, her basert på 10^6 kjøringer med n fra 1.000 til 10.000.

n- value	Time elapsed (ms)
1000	1303
2000	2636
3000	3842
4000	5205
5000	6576
6000	8002
7000	9408
8000	10867
9000	11970
10000	13482
Iterations:	1000000, (10^6)



Figur 1 Kjøring av metode 1

Vi ser at grafen som skisseres blir tilnærmet lineær.

Metode 2

Metoden bruker også en lineær rekursjon for å gå gjennom tall fra n til 1, men har økt effektivitet i forhold til metode 1 ved å redusere n-verdi på neste metodekall til halvparten av den forrige n-verdien. Resultatet av dette gjør at algoritmen utfører $\log_2(n)$ metodekall, som tilsvarer en tidskompleksitet på log(n)

Dette kan vises ved å beskrive stegene ved hjelp av uttrykket:

$$T(n) = aT\left(\frac{n}{b}\right) + c * n^k$$

For metode 2 har vi følgende verdier: a= 1, b=2 og k = 0. Setter vi dette inn i formelen får vi:

$$b^k = a \rightarrow 2^0 = 1 \rightarrow 1 = 1$$

Dette betyr at kompleksiteten er på formen:

$$T(n) \in \Theta(n^k * \log(\mathrm{n}))$$

Siden k er 0, ender vi med resultatet:

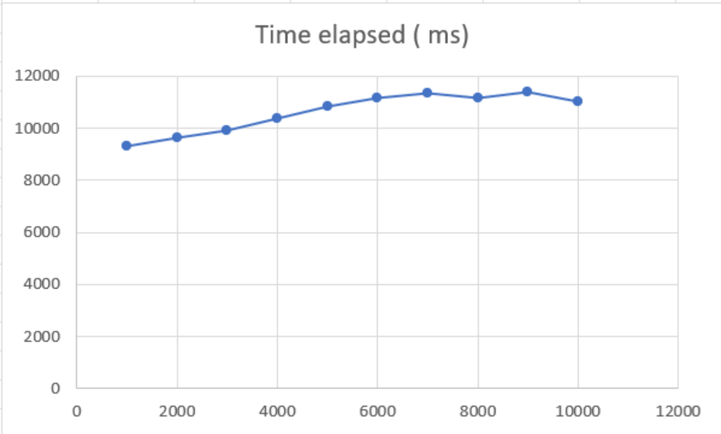
$$T(n) \in \Theta(\log(\mathrm{n}))$$

Denne metoden var vanskeligere å teste, da verdien for n ikke kunne gå særlig høyere enn 10.000 før man fikk StackOverflowException. For å kompensere for dette ble metoden kjørt mange ganger per test (opp til 10^9) men også her er man begrenset av minnekapasitet. Når iterasjonstallet nærmet seg maksverdien for heltall ble det kastet OutOfMemoryError på grunn av en array som ble for stor, noe som hindret ytterligere økning av tallet.

En kjørt test lignende den for forrige metode (bare med høyere antall iterasjoner), endte med et resultat som så slik ut:

n - value	Time elapsed (ms)
1000	9310
2000	9622
3000	9910
4000	10389
5000	10813
6000	11138
7000	11367
8000	11144
9000	11382
10000	11001

Iterations: 1000000000 (10^9)



Figur 2, Kjøring av metode 2

Vi ser at grafen kan hentyde til at tiden er logaritmisk proporsjonal med tiden, ser vi på forholdet mellom n=1000 og n=10000 ser vi at det er på 1.18, det forventede forholdet er log2(10000)/log2(1000) = 1.33. Altså ser det ut som om svaret kan stemme overens med hypotesen.

Sammenligning

Method 1:

n	amount of iterations	Time (ms)	Relative difference
1000	100000 (10^5.0)	131	1.0
2000	100000 (10^5.0)	248	1.8931297709923665
5000	100000 (10^5.0)	584	4.458015267175573
10000	100000 (10^5.0)	1141	8.709923664122137

Method 2:

n	amount of iterations	Time (ms)	Relative difference
1000	100000 (10^5.0)	4	1.0
2000	100000 (10^5.0)	1	0.25
5000	100000 (10^5.0)	1	0.25
10000	100000 (10^5.0)	1	0.25

Figur 3, Div kjøringer med 10^5 iterasjoner

Om vi tester metodene med like mange iterasjoner, ser vi at den første metoden er mye tregere enn den andre, noe som kan forklares med forskjellen i tidskompleksitet når vi har såpass stor n.

Utskrift av kjøring

Test data from the given task:

Expected	Method 1	Method 2
32.5	32.5	32.5
141.4	141.39999999999998	141.39999999999998

Method 1:

n	amount of iterations	Time (ms)	Relative difference
1000	100000 (10^5.0)	119	1.0
2000	100000 (10^5.0)	237	1.9915966386554622
5000	100000 (10^5.0)	591	4.966386554621849
10000	100000 (10^5.0)	1148	9.647058823529411

Method 2:

n	amount of iterations	Time (ms)	Relative difference
1000	100000000 (10^8.0)	837	1.0
2000	100000000 (10^8.0)	883	1.054958183990442
5000	100000000 (10^8.0)	1095	1.3082437275985663
10000	100000000 (10^8.0)	1103	1.3178016726403823

Figur 4, Utskrift av programmet slik det er levert