

Verslag AD 3: Lukas Barragan Torres

Inhoudsopgave

- [Inleiding](#)
- [Ontwerpbeslissingen en implementatie](#)
 - [Tree](#)
 - [Huffman](#)
 - [Hu-Tucker](#)
 - [Knuth](#)
 - [Comprimeren](#)
 - [Gecodeerde lijnen](#)
 - [Lijnlengtes](#)
 - [Header \(footer\)](#)
 - [Extraheren](#)
 - [Sorteren](#)
- [Benchmarks](#)
 - [Tree](#)
 - [Compress](#)
 - [Extract](#)
 - [Sort](#)
 - [Sort vergelijking met Bash sort](#)
- [Opmerkingen](#)
 - [Opmerking 1](#)
 - [Opmerking 2](#)
 - [Opmerking 3](#)
- [Conclusie](#)

Inleiding

Dit verslag is gemaakt voor het vak algoritmen en datastructuren. De bedoeling is om een programma te schrijven dat kan comprimeren met een orde-bewarende prefix code. Het programma moet ook kunnen sorteren wanneer het gecomprimeerd is en deze terug extraheren.

Ontwerpbeslissingen en implementatie

Tree

Er zijn meerdere mogelijkheden om een orde-bewarende prefix boom op te stellen. In deze sectie wordt uitgelegd waarom een standaard Huffman implementatie niet zomaar volstaat en geeft vervolgens twee algoritmes die dit wel correct doen, namelijk het algoritme van Hu-Tucker en het algoritme van Knuth. Beide zijn geïmplementeerd in het project, maar enkel Hu-Tucker wordt gebruikt.

Huffman

Wanneer we van een frequentietabel naar een orde bewarende prefix boom willen gaan kunnen we niet zomaar het standaard Huffman algoritme gebruiken. Dit is makkelijk te zien met een voorbeeld. Stel dat de volgende tekst gecompriemd moet worden:

```
ABBC
```

Dan zal volgens het Huffman algoritme de karakters met de laagste frequentie eerst samengenomen worden, namelijk `A` en `C`. Daarna wordt deze net aangemaakte deelboom samengenomen met het karakter `B`. De prefix boom die hieruit ontstaat kan onmogelijk orde-bewarend zijn. Moest dit wel zo zijn zou $\text{Code}(A) < \text{Code}(B) < \text{Code}(C)$, maar aangezien `A` en `C` zusterknopen zijn verschillen hun codes met exact 1. Hier kan $\text{Code}(B)$ dus onmogelijk tussen.

Men moet wel opletten, want er zijn gevallen waar de Huffman-code orde-bewarend is. Neem de volgende tekst:

```
ABCC
```

Hier zal het Huffman algoritme wel een orde-bewarende code genereren (mits het kleinste karakter in de linker tak geplaatst wordt bij het samennemen). Dit voorbeeld zal volgende codes genereren:

```
A: 00
B: 01
C: 1
```

Hu-Tucker

Het Hu-Tucker algoritme^[1] werkt als volgt. Men begint met de frequenties - gesorteerd volgens karakter - op een rij. Al deze frequenties worden initieel gelabeld als `TERMINAL` (dit komt omdat al deze knopen bladeren voorstellen). Daarna neem je het paar met de kleinste som van frequenties, zonder dat er een `TERMINAL` knoop tussen deze knopen zit. Je vervangt de linkerknoop uit het paar met de nieuwe samengevoegde knoop (identiek aan het Huffman algoritme, hier wordt dus een klein boompje van gemaakt met opgetelde frequentie) en verwijdert de rechter knoop uit de lijst. Daarna label je deze nieuwe knoop

met `INTERNAL`. Doe dit tot er maar één knoop meer overblijft. deze knoop stelt een boom voor. Deze boom is niet noodzakelijk een orde-bewarende boom. Bereken van elk blad in de boom de lengtes. Begin daarna bij de lengte l_0 van het meest linkse blad. de eerste code bestaat dan uit l_0 nullen. volgende codes worden op volgende manier gevonden:

1. Tel 1 op bij de vorige code^[2]
2. Vul de code aan met nullen of neem er weg tot deze even lang is als de lengte

Illustratie ter voorbeeld. De lengtes `3, 3, 2, 4, 4, 4, 4` geven de codes `000, 001, 01, 1000, 1001, 1010, 1011, 11`

Het opstellen van de initiële boom waaruit de lengtes worden gehaald kan technisch gezien in $\theta(n * \log(n))$ met n het aantal karakters. Aangezien wij over maximaal 128 karakters lopen koos ik voor een simpele $\theta(n^2)$ implementatie, waarbij heel de lijst overlopen wordt tijdens het zoeken naar het kleinste paar. De lengtes van van de bladeren vinden en hieruit de codes berekenen is $\theta(n)$. Het algoritme is dus $\theta(n^2)$.

Knuth

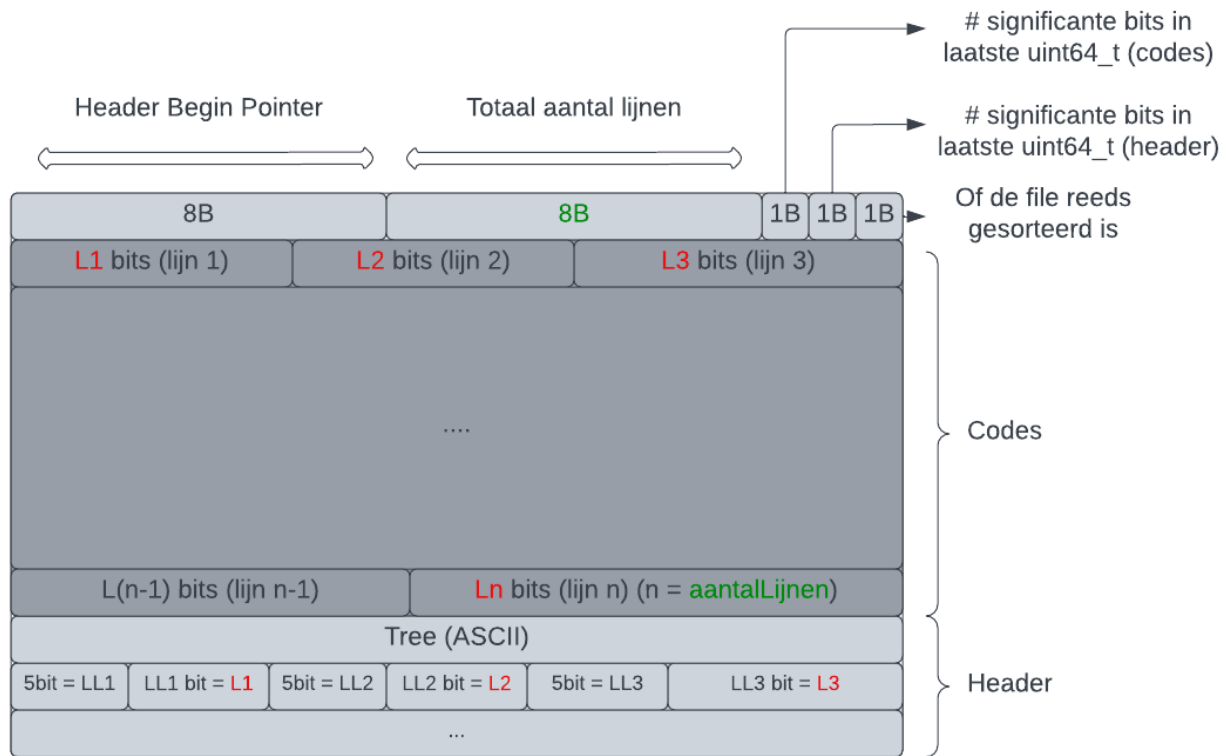
Naast het Hu-Tucker algoritme implementeer heb ik ook nog het algoritme van Knuth^[3] voor het opstellen van een optimale orde-bewarende boom geïmplementeerd. Deze heb ik uiteindelijk niet gebruikt maar is wel nog te vinden in de code (`make_OPC.c`).

Het algoritme voor het bouwen van een optimale alfabetische binaire boom berust op het principe dat in een optimale OPC boom, zowel de linker- als de rechter kinderboom ook optimaal zijn. Dit principe wordt toegepast in een dynamisch programmeringskader met de matrix `M`. In `M` staat elke cel `M[i][j]` in de rechter bovenhelft voor de kosten van de optimale boom met karakters van `i` tot `j`, berekend door alle mogelijke splitsingen van de karakterreeks te overwegen en de kosten van de optimale linker- en rechter sub-bomen op te tellen. De linker onderhelft van `M` bevat de som van de frequenties van de karakters in elke sub-reeks. Door dit principe te volgen, bouwt het algoritme voort op de berekende kosten van kleinere, optimale sub-bomen om stapsgewijs de kosten van grotere bomen te vinden, wat leidt tot de optimale boomstructuur voor de gehele reeks karakters. Het algoritme van Knuth is $\theta(n^3)$

Comprimeren

Voor het comprimeren van bestanden heb ik een aantal belangrijke keuzes gemaakt.

- Gecodeerde lijnen zijn **niet** gealigneerd tot de volgende byte, uint64, etc.
- De lengte van elke lijn wordt ook ongealigneerd bijgehouden in het bestand.
- Lijnlengtes bevinden zich in de header (technisch gezien footer)
- `\n` wordt **wel** gecodeerd.
- Er is een byte die aangeeft of het bestand reeds gesorteerd werd.



Gecodeerde lijnen

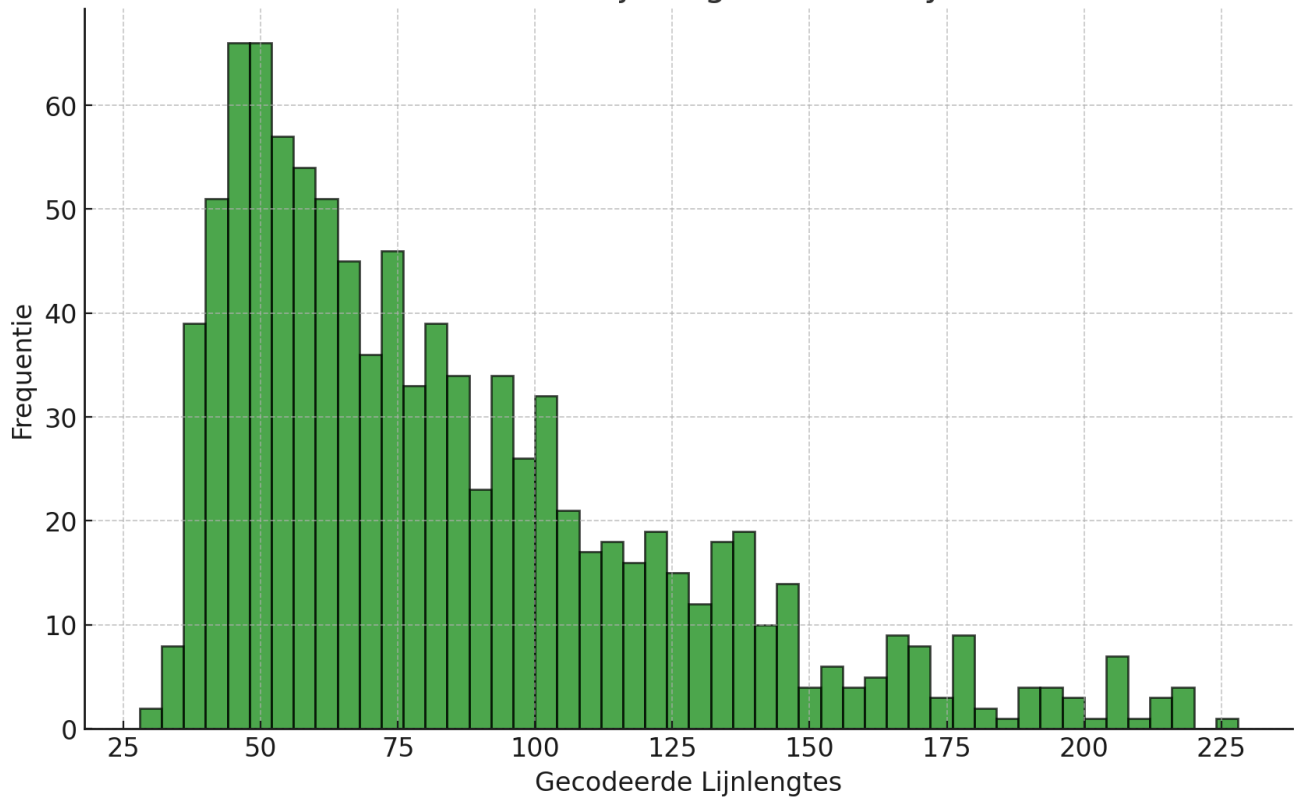
Een gecodeerde lijn is vrij simpel. De karakters worden gecodeerd en vervolgens uitgeschreven als bit-sequentie. Dit wil zeggen dat deze kan beginnen en eindigen middenin een byte.

Lijnlengtes

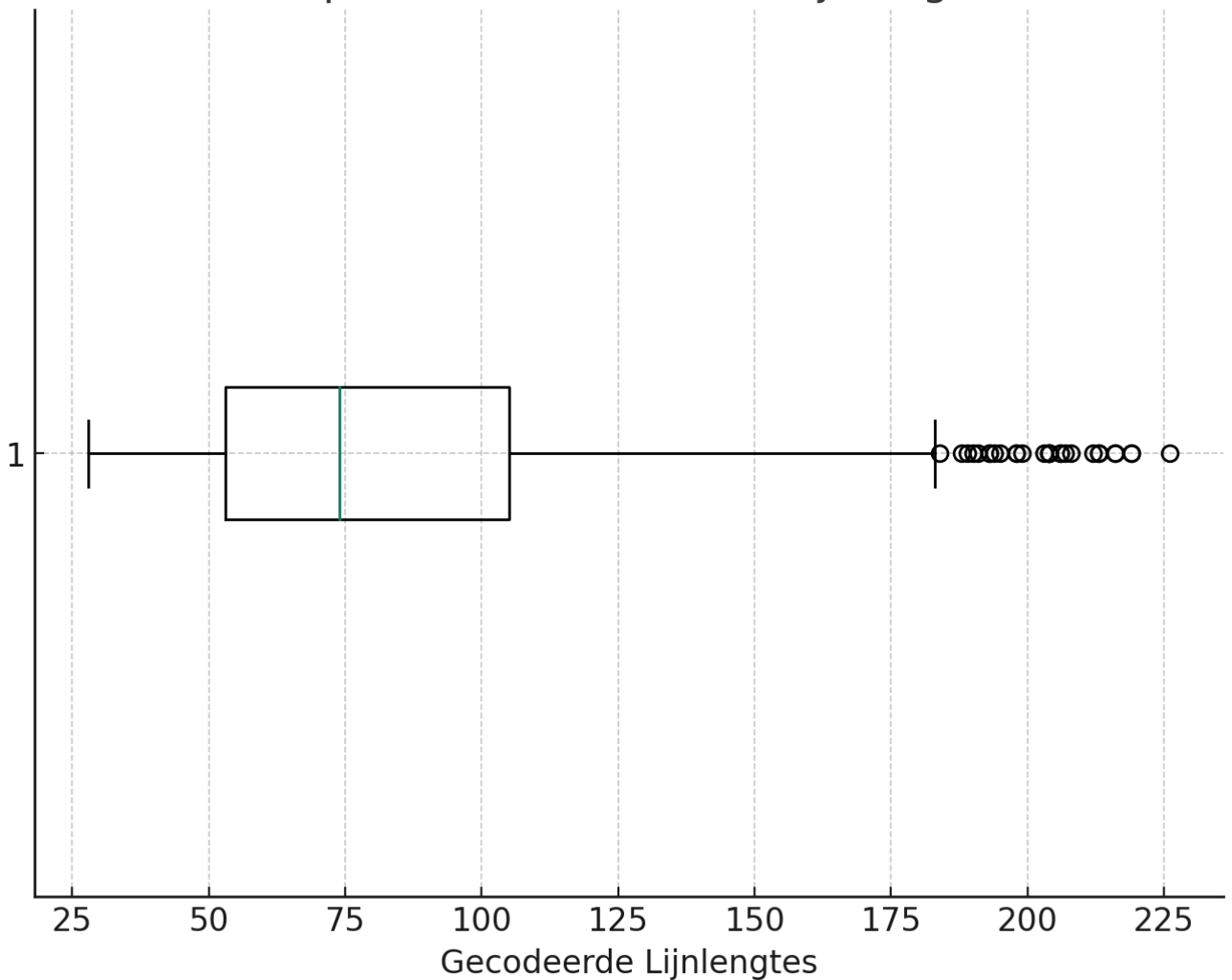
Het project specificeert dat het programma een gecomprimeerd bestand kan sorteren aan de hand van orde-bewarende prefix codes. Je zou tijdens het sorteren karakters één per één kunnen extraheren om zo de newlines te kunnen identificeren, maar dit gaat helemaal in tegen het principe van de orde-bewarende prefix code. Daarom bevat de header de lengte van elke lijn. Dit heeft verschillende voordelen die uitgediept worden in de [sorteren-sectie](#) van het verslag.

Het is wel duidelijk dat deze lijnlengtes met een hevige overhead gepaard gaan. Bij een naïeve implementatie zou voor elke lijnlengte 22 bits nodig zijn. Dit komt omdat de maximale lijnlengte 45354 is en de lengte van een code maximaal 64 is. Om de lengte van deze gecodeerde lijn voor te stellen heb je dus $\text{ceil}(\log_2(45354 \cdot 64)) = 22$ bits nodig. Echter, wanneer we naar de distributie kijken van gecodeerde lijnlengtes voor bijvoorbeeld `tiny.txt`,

Gecodeerde lijnlengtes voor tiny.txt



Boxplot van Gecodeerde Lijnlengtes



dan zien we dat de meeste lijnen helemaal niet zo lang zijn. Een oplossing hiervoor is om een lijnlengte op te splitsen in twee delen. Het eerste deel bestaat uit $\text{ceil}(\log_2(22)) = 5$ bits die de lengte van de lengte van de lijn aangeeft (op de diagram aangegeven met `LL`). Het tweede deel is dan `LL` bits die de werkelijke lengte bevatten. Met deze aanpassing gebruiken we op `tiny.txt` ongeveer 47% minder bits voor het opslaan van de lijnlengtes.

Header (footer)

De header met alle lijnlengtes bevindt zich op het einde van het bestand. Deze ontwerpbeslissing heb ik genomen voor een aantal redenen:

- Lijnlengtes kunnen snel sequentieel overlopen worden zonder te springen in het bestand.
- Tijdens het comprimeren is geen tussentijdse buffer nodig om de lijn in op te slaan zodat de lengte kan uitgeschreven worden naar het bestand vóór de lijn.
- Deze header kan gemakkelijk weggelaten worden na het sorteren.

Dit laatste is ook de reden dat er in het begin van het bestand een byte is dat aangeeft of het bestand ooit al gesorteerd werd. De lijnlengtes werden tijdens compressie bijgehouden om efficiënt en zonder te extraheren te kunnen sorteren, maar wanneer een bestand al gesorteerd is geweest zullen deze waarden nooit meer gebruikt worden^[4]. Neem als voorbeeld `large.txt`. Deze is `945 MiB`. Na het comprimeren wordt dit `592 MiB`. Wanneer uiteindelijk gesorteerd wordt zal de grootte nog dalen naar `523 MiB`.

Deze keuze verklaart ook waarom newlines gecodeerd worden. Moest dit niet gebeuren zou extraheren zonder lijnlengtes niet meer mogelijk zijn omdat niet duidelijk is waar een lijn eindigt.

Extraheren

Bij het extraheren wordt de boom in de header verwerkt. Daarna worden de codes bit per bit ingelezen. Elke bit stemt overeen met een afdaling in de boom. Wanneer men een blad tegenkomt weten we het karakter dat uitgeschreven zal moeten worden.

Sorteren

Doordat lijnlengtes sequentieel worden bijgehouden in de header kunnen we deze snel overlopen en berekenen hoeveel lijnen er in elk blok passen. Op basis van deze lijnlengtes construeren we voor elke lijn een `struct` met de lengte van de lijn en waar deze begint (in de ingeladen buffer). Deze `struct`'s worden nadien gesorteerd via Heap-sort en uitgeschreven naar een tijdelijk bestand. De keuze voor Heap-sort was hoofdzakelijk omdat dit een "in-place" sorteer algoritme is dat dus - buiten de lijst waarin gesorteerd wordt - geen extra geheugen gebruikt wordt.

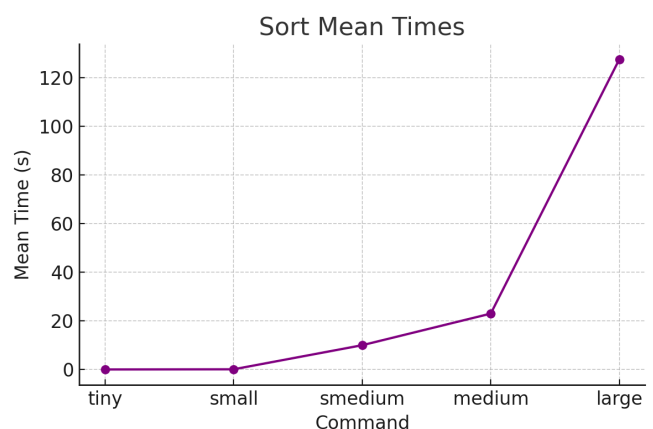
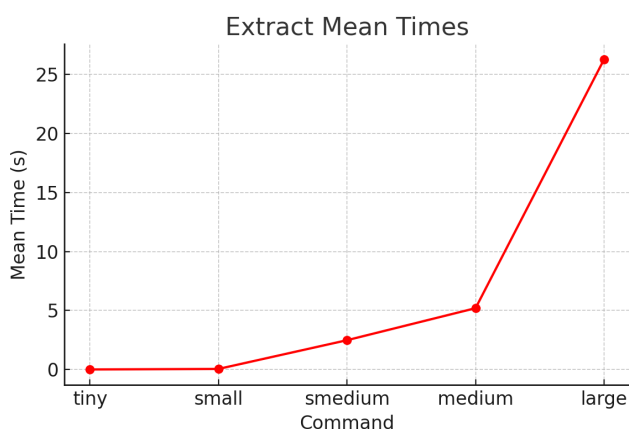
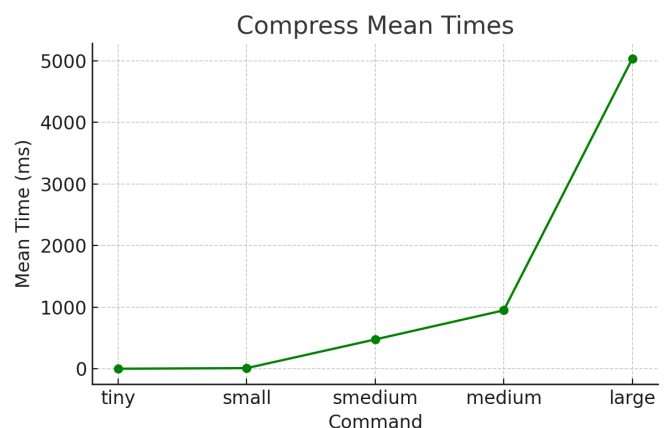
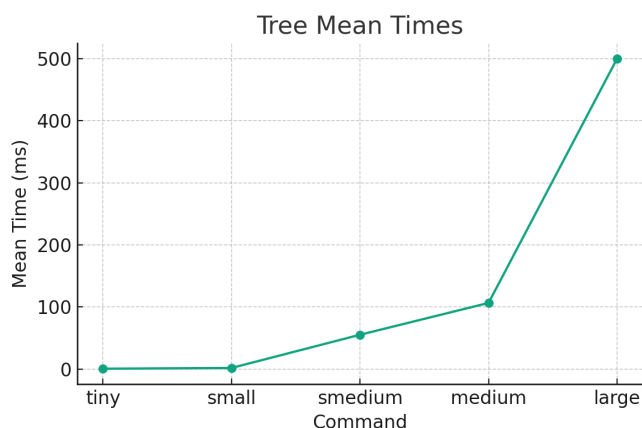
Wijzers naar het begin van de blokken in dat bestand wordt ook bijgehouden. Daarna worden `m` (aantal blokken) inputbuffers^[5] aangemaakt en één outputbuffer^[5-1]. Nu zal volgens het extern sorteren algoritme telkens uit de eerste lijnen van het blok de kleinste gekozen en uitgeschreven worden tot de blokken leeg zijn.

Een performantie-analyse en vergelijking met Unix sort is te vinden in de [Benchmarks sectie](#)

Benchmarks

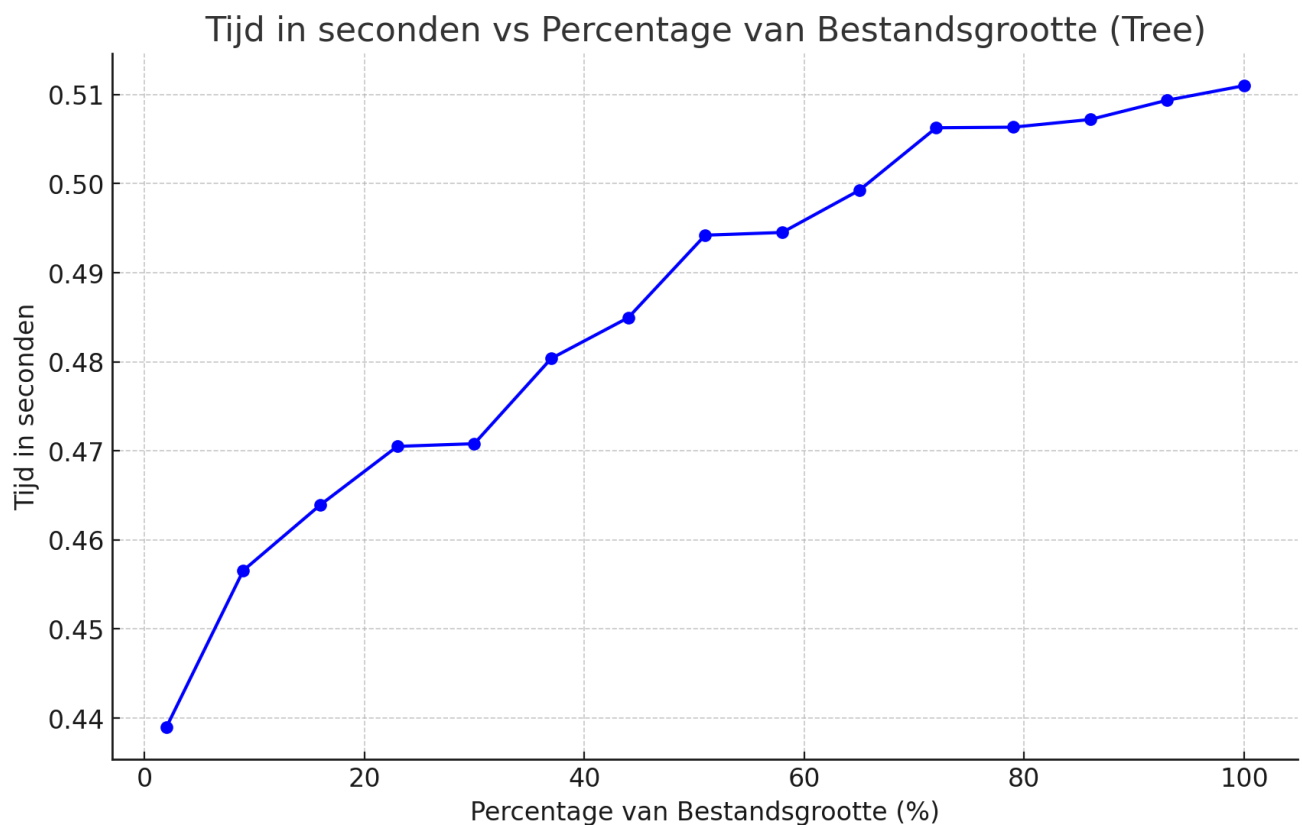
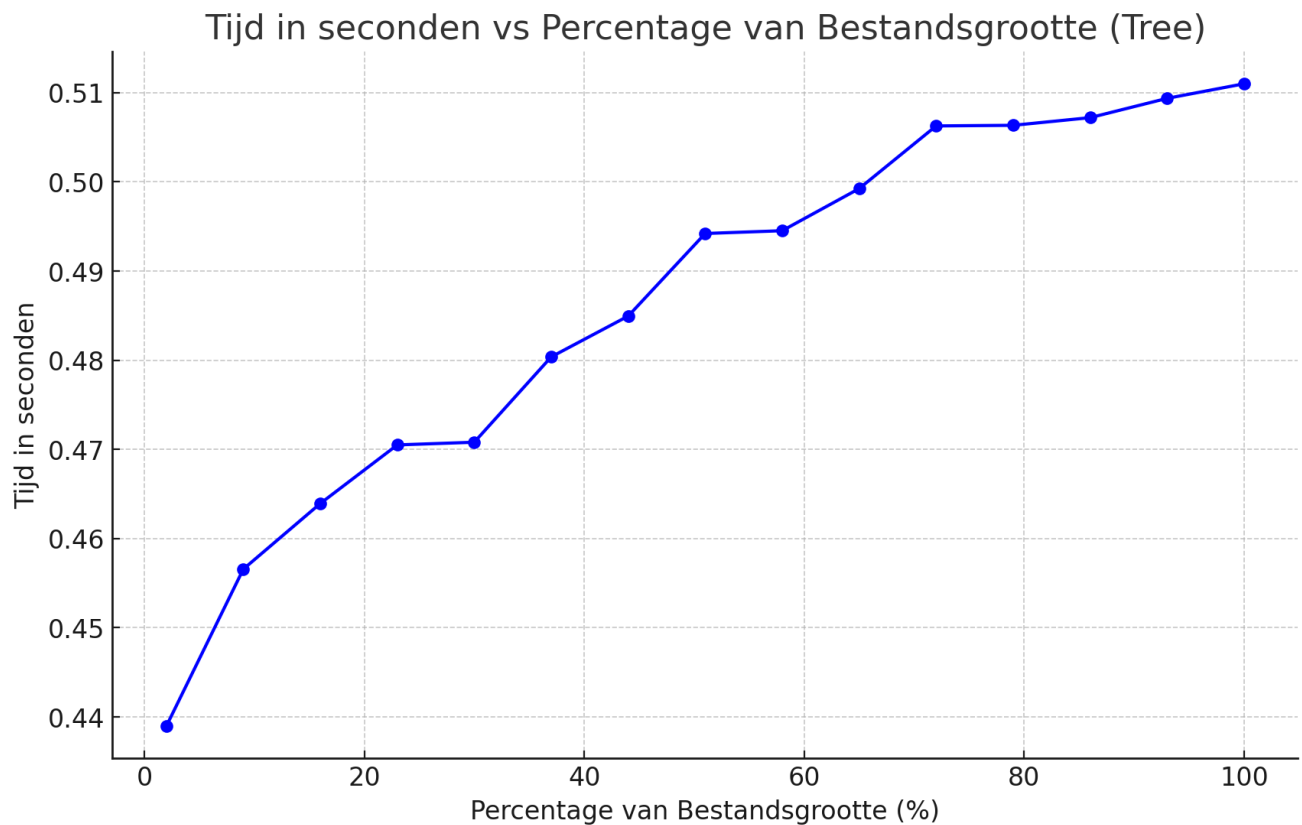
Elk van de commando's (`tree`, `compress`, `extract`, `sort`) zal toegepast worden op elk test-bestand (`tiny.txt`, `small.txt`, `smedium.txt`, `medium.txt` en `large.txt`) met een vaste geheugengrootte `m = 1GiB`. We verkrijgen de volgende tabel en grafiek:

Command	Tree_Mean	Compress_Mean	Extract_Mean	Sort_Mean
tiny	0.45 ms	0.52 ms	0.80 ms	0.80 ms
small	1.65 ms	9.94 ms	45.37 ms	69.63 ms
smedium	55.15 ms	476.00 ms	2.48 s	9.98 s
medium	106.43 ms	949.21 ms	5.20 s	22.99 s
large	500.09 ms	5.04 s	26.30 s	127.64 s



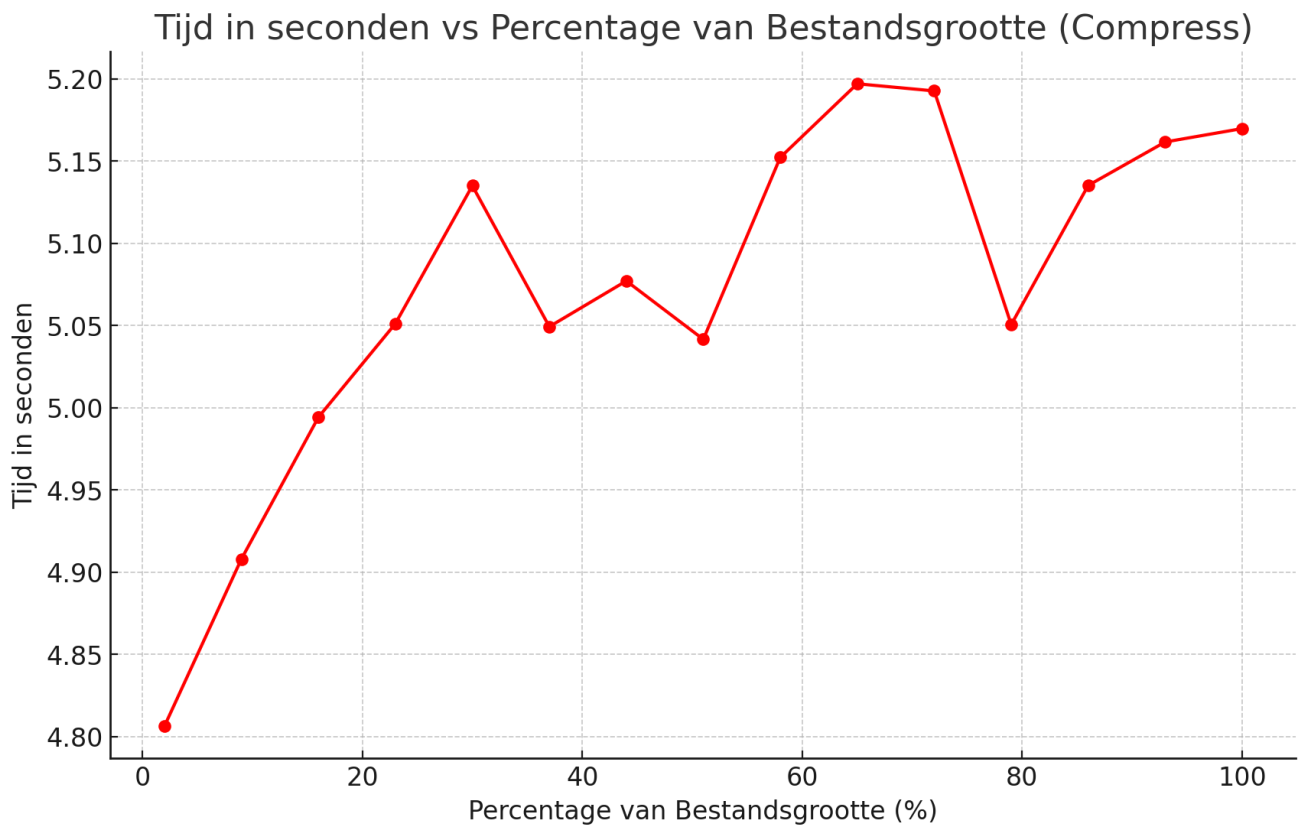
Daarna passen we elk commando toe op `large.txt` met variërende `m` om de invloed hiervan te bekijken laten we `m` lopen van `2% = 19MiB` naar `100% = 945MiB` van de filegrootte in stappen van `7%` (`2`, `9`, `16`, `23`, `30`, `37`, `44`, `51`, `58`, `65`, `72`, `79`, `86`, `93`, `100`).

Tree



We zien dat het programma sneller is wanneer minder geheugen wordt meegegeven. Dit is contra-intuïtief en heeft misschien te maken met het feit dat het langer duurt om een groot blok geheugen te vinden dan vele kleine blokken.

Compress

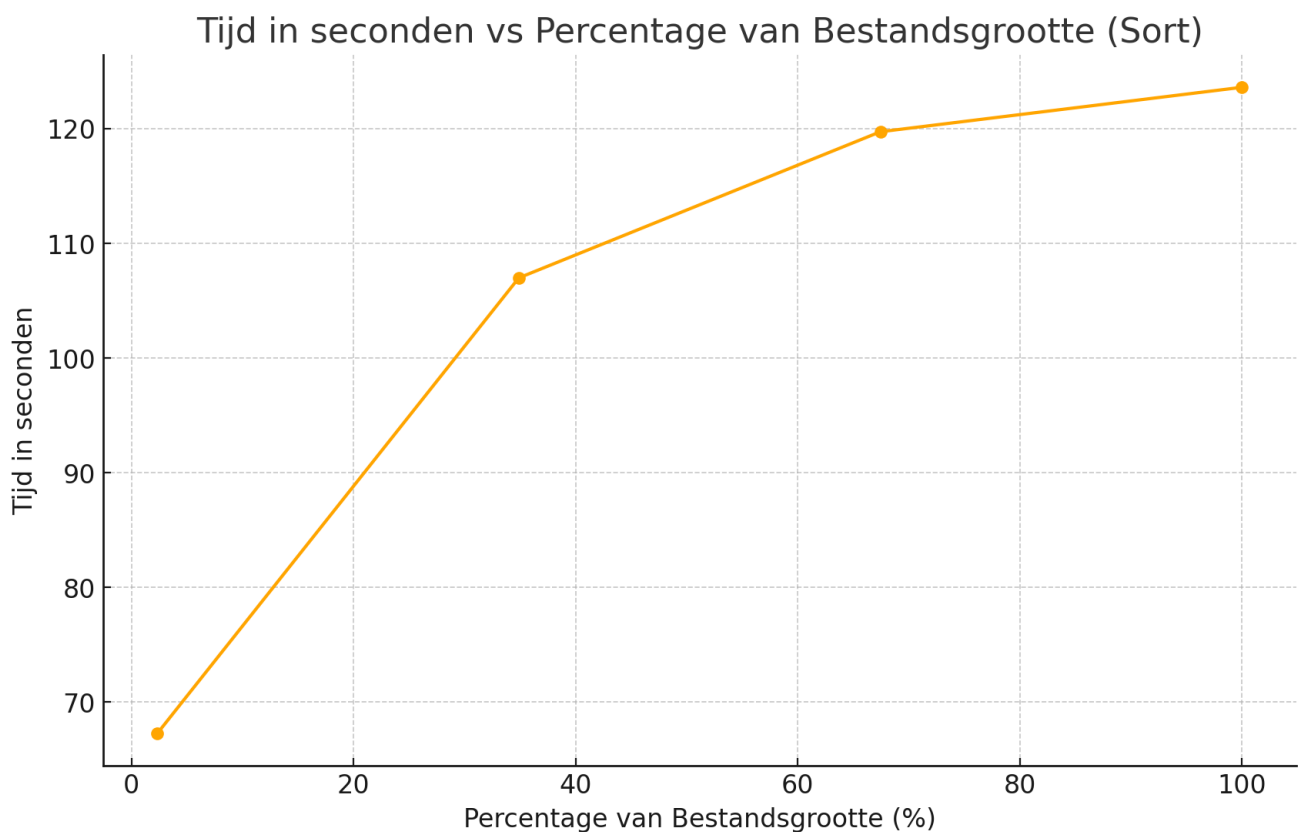


We zien hetzelfde fenomeen bij de `compress` functie

Extract

Extract volgt ook dezelfde trend als compress en tree.

Sort



Sort vergelijking met Bash sort

Bij het vergelijken met bash sort op `large.txt` zullen we `m = 32MiB` nemen. We bekomen de volgende data. sort kan dit sorteren in gemiddeld `176 seconden`. ons programma doet het in `62 seconden` (zoals te zien op de grafiek). Het lijkt er dus op dat eerst compressie toepassen een positieve invloed heeft op de snelheid van sorteren.

Opmerkingen

Opmerking 1

Doorheen het project zijn een aantal belangrijke structuren gedefinieerd. De twee belangrijkste zijn `BitInputHandler` en `BitOutputHandler`. Deze twee `struct`'s implementeren het uitschrijven en inlezen van een arbitrair aantal bits. Omdat ik niet met `uint64` of `byte` alignment werk was deze abstractie nodig om dit project proper en haalbaar te houden. Bij het creëren van deze structuren moet een het aantal bytes meegegeven worden die de buffer achterliggend zal gebruiken. Dit is handig bij bijvoorbeeld het mergen van blokken wanneer er bijvoorbeeld `k BitInputHandlers` aangemaakt kunnen worden met elk `m / k` bytes als buffergrootte.

Opmerking 2

In het begin van dit project schreef ik uit per byte. Dit heb ik later veranderd naar uitschrijven per `uint64_t` om performantie redenen. Door tijdsnood heb ik niet meer alle variabelenamen kunnen veranderen van `byte` naar `uint64`. zo moet `significantBitsInLastByte` bijvoorbeeld `significantBitsInLastUint64` zijn

Opmerking 3

mijn programma slaagde niet voor de extra docker-test omdat deze te veel geheugen gebruikte. Wanneer ik dit met de `massif-visualiser` bekeek werd er echter nooit meer dan de meegegeven `m` gealloceerd. Ik vermoed dat er misschien allocaties zijn die op de achtergrond door `glibc` gebeuren die deze testen doen falen.

Conclusie

De keuzes die ik in het project heb gemaakt zijn eerder gericht om de compressiefactor zo goed mogelijk te maken, bijvoorbeeld het ongealigneerd laten van de codes, het logaritme nemen van de lijnlengte en het weglaten van deze lengtes na het sorteren.

Ik heb ingezet om code proper en leesbaar te maken zonder hiervoor de performantie te laten inboeten. Ik hoop dat dit gelukt is. Bedankt.

Lukas Barragan Torres

1. Hu, T. C., & Tucker, A. C. (1971). Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM Journal on Applied Mathematics*, 21(4), 514–532. <http://www.jstor.org/stable/2099603>↵
2. Dit heb ik zelf gevonden na het lezen van https://math.mit.edu/~djkh/18.310/18.310F04/huffman_algorithms.html↵
3. Knuth, D. E. (1997). *The art of computer programming* (3rd ed.). Addison Wesley.↵
4. Men zou natuurlijk wel het argument kunnen maken dat dit gesorteerd bestand nog doorzocht zou moeten worden. In dat geval zou het misschien beter zijn dit niet weg te laten.↵
5. Deze structuren wordt in meer detail uitgelegd in de [Opmerkingen sectie](#)↵↵