DM507/DS814 Algoritmer og datastrukturer

Forår 2026

Projekt, del III

Institut for matematik og datalogi
Syddansk Universitet

27. april, 2026

Dette projekt udleveres i tre dele. Hver del har sin deadline, således at arbejdet strækkes over hele semesteret. Deadline for del III er tirsdag den 26. maj kl. 23:59. De tre dele I/II/III er ikke lige store, men har omfang omtrent fordelt i forholdet 15/25/60. Projektet skal besvares i grupper af størrelse to eller tre.

Mål

Målet for del III af projektet er at lave dit eget værktøj til at komprimere filer. Komprimeringen skal ske via Huffman-kodning. Der skal laves to programmer: et til at kode/komprimere en fil, og et til dekode den igen.

Vær sikker på at du forstår Huffmans algoritme (Cormen et al. afsnit 15.3 indtil side 43 midten [3. udgave: afsnit 16.3 indtil side 433]) før du læser resten af denne opgavetekst.

Opgaver

Opgave 1

Du skal i Python implementere et program, der læser en fil og laver en Huffman-kodet version af den. Den præcise definition af input og output følger nedenfor. Programmet skal hedde `Encode.py`, og skal kunne køres sådan:

```
python Encode.py nameOfOriginalFile nameOfCompressedFile
```

Input-filen skal ses som en sekvens af bytes (8 bits). Hver byte udgør et tegn. Der er derfor $2^8 = 256$ mulige forskellige tegn i input, og vores alfabet har således størrelse 256. Tegn kaldes i resten af opgaven for bytes. I Python vil

kaldet `read(1)` fra file objects læse én bytes fra en fil, hvis filen er åbnet i binary reading mode (med argumentet `'rb'` i `open`).

Python (Python 3) har en type ved navn "byte objects", som man kan tænke på som (immutable) lister af heltal med værdier mellem 0 og 255.[1] Kaldet `read(1)` fra file objects returnerer et byte object af længde én. Når man har nået enden af filen, returnerer `read(1)` et byte object af længde nul (i stedet for længde én).

> En byte er derfor otte bits i filer på disk, men ankommer til dit Python-program som et byte object af længde én. Hvis man tilgår første (og eneste) element i dette byte objekt, får man et heltal mellem 0 og 255. Hvis byte objektet hedder `b`, tilgås første element som `b[0]`. *Dette heltal skal du bruge til at repræsentere byten undervejs i dit Python-program.*

Sammenhængen mellem `int`'s og bytes er som for det binære talsystem, dvs. som vist nedenfor. Denne sammenhæng kender `read()` og `write()`, når man åbner filer i binary mode, og man skal ikke selv beskæftige sig med den i Python-programmet. *Man skal altså blot lade bytes være de heltal mellem 0 og 255, der er blevet indlæst som beskrevet ovenfor.*

Heltal	Byte
0	00000000
1	00000001
2	00000010
3	00000011
⋮	⋮
254	11111110
255	11111111

Programmet `Encode` skal virke således:

1. Scan inputfilen og lav en tabel (en liste med 256 entries) over hyppigheden af de enkelte bytes (husk at bytes er heltal mellem 0 og 255, og kan bruges som indekser i lister).

2. Kør Huffmans algoritme med tabellen fra punkt 1 som input (alle 256 entries, *også* dem med hyppighed nul[2]).

[1] Lidt forvirrende vises et helt byte object i Python ved hjælp af ASCII-tegn, hvis det er muligt (f.eks. repræsenteres et byte object `s` med indhold `[120,121,122]` som `b'xyz'` (med `b` for "binary"), jvf. at tegnet `x` i ASCII-tabellen har nummer 120), men enkeltelementer i byte objekter *er* heltal (f.eks. er `s[0]` lig `120`).

[2] Huffmans algoritme virker kun for alfabeter med mindst to tegn. Hvis man udelader tegn med hyppighed nul, vil filer med indhold af typen `aaaaaaa` give et alfabet af størrelse ét. Derfor dette krav.

3. Konvertér Huffmans træ til en tabel (en liste med 256 entries) over kodeord for hver af de mulige bytes (husk at bytes er repræsenteret som heltal mellem 0 og 255, og kan bruges som indekser i lister).

4. Skriv de 256 hyppigheder (dvs. 256 de heltal i tabellen fra punkt 1) til outputfilen.

5. Scan inputfilen igen. Undervejs find for hver byte dets kodeord (ved opslag i tabellen fra punkt 3 over kodeord), og skriv dette kodeords bits til outputfilen.

I ovenstående scannes inputfilen to gange. Dette er at foretrække frem for at scanne den én gang og derefter gemme dens indhold i programmet til videre brug, eftersom programmets RAM-forbrug derved stiger fra $O(1)$ til inputfilens størrelse (som kan være meget stor).

Punkt 3 skal udføres ved at lave et rekursivt gennemløb (i stil med et inorder-gennemløb af et søgetræ) af Huffman-træet og derved generere alle kodeordene. Under gennemløbet vedligeholder man hele tiden et kodeord svarende til stien fra roden til den nuværende knude. Når man går nedad i træet skal kodeordet udvides med enten 0 eller 1. Når man når et blad, skal dette kodeord gemmes i tabellen. Kodeord (som jo ikke er lige lange) skal repræsenteres af en streng af `'0'` og `'1'` tegn. En sådan streng kan så efter et opslag i tabellen gennemløbes tegn for tegn og konverteres til bits, som skrives til outputfilen.

Huffman-kodning skal implementeres via en prioritetskø (jvf. pseudokoden Cormen et al. side 434 [3. udgave: side 431]). *Hertil skal genbruges gruppens implementation* `PQHeap.py` *fra del I.*

I del I var prioritetskøen en liste af tal. Her i del III skal prioritetskøen være en liste af `Element`'s, hvor koden for `Element` udleveres sammen med denne projektbeskrivelse. Et `Element` har to felter `key` og `data`. Her skal `data` repræsentere et træ, på en måde som vi beskriver nedenfor, og `key` skal være træets samlede hyppighed (dvs. et tal). Når sammenligningsoperatorer (`==`, `<`, `=<`, etc.) bruges mellem `Element`'s, vil koden for `Element` gøre, at sammenligningen automatisk bliver mellem `key`'s for de to `Element`'s.

> *Derfor kan koden fra* `PQHeap.py` *genbruges direkte.* Det eneste, som ændres, er at i `insert(A,e)` er `e` et `Element`, og i `extractMin(A)` returneres et `Element` (nemlig det med mindste `key` blandt alle `Element`'s i prioritetskøen).

Med denne projektbeskrivelse er udleveret et program `PQSortElements.py`, som er en variant af `PQSort.py` fra del I, men justeret til at sortere `Element`'s i stedet for heltal. I `PQSortElements.py` kan ses, hvordan man opretter

`Element`'s, hvordan man tilgår felterne i et `Element` med navn `e` som `e.key` og `e.data`, og hvordan `Element`'s indsættes med `PQHeap.insert(pq,e)` og udtages med `PQHeap.extractMin(pq)`. Du skal bruge din egen `PQHeap.py` fra del I for at køre programmet.

Træerne, som skal bruges i Huffmans algoritme, er binære træer med en byte (dvs. et heltal med værdi mellem 0 og 255) gemt i blade. Se figuren i bogen side 435 [3. udgave: side 432]). Hyppigheden skal dog *ikke* gemmes i indre knuder, selv om dette sker i illustrationerne i bogen, da kun hyppigheden for træets rod bruges i Huffmans algoritme, og rodens hyppighed er allerede gemt som `key` i det `Element`, der indeholder træet. Indre knuder er derfor tomme. Man kan enten lave to forskellige Python-klasser til henholdsvis indre knuder og til blade. Eller man kan bruge samme Python-klasse til alle knuder, og så lade feltet for knudens byte indeholde -1 for indre knuder, hvorved disse kan skelnes fra blade.

Den præcise specifikation af output af programmet `Encode` er:

> Først 256 heltal hver skrevet som 32 bits (hvilket fylder $256 \cdot 32$ bits i alt), som angiver hyppighederne af de 256 mulige bytes i input, derefter de bits, som Huffman-kodningen af input giver.

Bemærk, at for korte filer (eller lange filer, som ikke kan komprimeres væsentligt med Huffmans metode) kan output af `Encode` være lidt længere end den oprindelige fil, eftersom vi bruger lidt plads på at gemme hyppighedstabellen. Man kunne tænke sig mange måder at undgå eller begrænse denne situation på, men dette er ikke en del af projektet.

Når man skriver et kodeord i output, har man brug for at skrive bits én ad gangen. Der er i Python ikke metoder til at læse og skrive enkelte bits til disk (mindste enhed er en byte), men underviseren har udleveret et bibliotek `bitIO.py`, som indeholder klasserne `BitReader` og `BitWriter`, der har metoder, som kan gøre dette. Der er også metoder til at læse og skrive heltal som 32 bits, hvilket skal bruges, når man skriver hyppighedstabellen som den første del af output. Se det udleverede program `text.py` for eksempler på metodernes anvendelse. Man behøver i dette projekt *ikke* forstå metodernes interne virkemåde, man skal blot kunne bruge dem.

Bemærk, at output fra `Encode` *ikke* kan læses med normale editors eller tekstbehandlingsprogrammer. De gemte Huffman-kodeord giver jo kun mening, når de fortolkes som Huffman-koderne fra præcis jeres Huffman-træ, og dem kender andre programmer ikke.[3] Kun jeres `Decode` fra opgave 2 kan læse den del af output. Samme problematik gælder for første del af output,

[3] Andre programmer vil forsøge at fortolke bits som de plejer, f.eks. som utf8-, latin1- eller ASCII-encoded tekst. Det vil resultere i meningsløst output.

som har gemt hyppighedstabellen med `writeint32bits(intvalue)` fra den udleverede klasse `BitWriter`. Denne del kan kun læses af `readint32bits()` fra den udleverede klasse `BitReader`, sådan som `Decode` skal starte med at gøre (se næste afsnit).

Til hjælp til debugging kan man evt. bruge programmet fra opgaven fra uge 19, som læser de enkelte bits i en fil og undervejs udskriver disse bits som `0` og `1` tegn. Med projektet er også udleveret en samling test-filer, med angivelse af, hvor store de komprimerede filer er, hvis `Encode` er lavet korrekt. Endelig kan man checke, at man får den oprindelige fil tilbage efter brug af `Encode` og derefter `Decode`.

> OPSUMMERING: I opgave 1 skal man bruge kaldet `read(1)` fra file objects til at *læse bytes fra inputfilen* (den originale fil). Man skal bruge metoderne `writeint32bits(intvalue)` og `writebit(bit)` fra klassen `BitWriter` i biblioteket `bitIO.py` til at *skrive heltal* (for hyppighedstabel) *og bits* (for Huffmans-koderne) *til outputfilen* (den komprimerede fil). Begge filer skal åbnes i "binary mode". Når en `BitWriter` instantieres, skal den have et file object som argument.

Opgave 2

Du skal i Python implementere et program, som læser en fil med data genereret af dit program fra opgave 1, og som skriver en fil med det originale (ukomprimerede) indhold. Programmet skal hedde `Decode.py`, og skal kunne køres sådan:

```
python Decode.py nameOfCompressedFile nameOfDecodedFile
```

Programmet `Decode` skal virke således:

1. Indlæs fra inputfilen tabellen over hyppighederne for de 256 bytes.
2. Generer samme Huffman-træ som programmet fra opgave 1 (dvs. *samme* implementation skal bruges begge steder, så der i situationer, hvor Huffman-algoritmen har flere valgmuligheder, vælges det samme).
3. Brug dette Huffman-træ til at dekode resten af bits i inputfilen, og imens som output skrive den originale version af filen. Dette gøres ved at bruge de læste bits til at navigere ned gennem træet (mens de læses). Når et blad nås, udskrives dets byte, og der fortsættes fra roden.

Alt dette kan gøres i ét scan af input.

Bemærk, at det samlede antal bits fra udskrivningen af Huffman-koderne af det udleverede bibliotek om nødvendigt bliver rundet op til et multiplum af otte (dvs. til et helt antal bytes), ved at nul-bits tilføjes tilf sidst, når filen lukkes. Dette skyldes, at man på computere kun kan gemme filer, som indeholder et helt antal bytes. Disse muligt tilføjede bits må ikke blive forsøgt dekodet, da ekstra bytes så kan opstå i output. Derfor må man under dekodning finde det samlede antal bytes i originalfilen ved at summere hyppighederne, og under rekonstruktionen af den ukomprimerede fil holde styr på hvor mange bytes, man har skrevet.

For at skrive bytes til en fil, kan man i Python bruge kaldet `write(bytes([b]))` fra file objects (samt built-in funktionen `bytes()`) til at skrive en byte repræsenteret af heltallet `b`. Det pågældende file object skal være åbnet i binary writing mode (med argumentet `'wb'` i `open`).

> OPSUMMERING: I opgave 2 skal man bruge metoderne `readint32bits()` og `readbit()` fra klassen `BitReader` fra det udleverede bibliotek `bitIO.py` til at *læse heltal* (for hyppighedstabel) *og bits* (for Huffmans-koderne) *fra inputfilen* (den komprimerede fil). Man skal bruge kaldet `write(bytes([b]))` (hvor `write()` er fra file objects og `bytes()` er en built-in funktion) til at *skrive bytes til outputfilen* (den genskabte originale fil). Her er `b` et heltal som repræsenterer den byte, som skal skrives. Begge filer skal åbnes i "binary mode". Når en `BitReader` instantieres, skal den have et file object som argument.

Formalia

I skal kun aflevere jeres Python filer. Disse skal indeholde grundige kommentarer. De skal også indeholde navnene og SDU-logins på gruppens medlemmer. Jeres programmer vil blive testet med mange typer filer (`txt`, `.doc`, `.jpg`,...), og I bør selv gøre dette inden aflevering, men I skal ikke dokumentere disse test.

I skal aflevere `Encode.py` og `Decode.py` samt alle andre Python filer, som kræves for at køre dem, f.eks. `PQHeap.py` fra del I. Filerne skal enten afleveres som individuelle filer eller som ét `zip`-arkiv.

Hvis nogen dele af koden er skrevet af generativ AI, eller på anden måde via kopiering fra andet værk, skal dette erklæres i source-filerne. Koden vil så *ikke* blive læst og vil ikke få feedback, men vil kun blive udsat for tests. Dette er fordi, at formålet med projektet er, at de studerende opnår kompetencer til *selv at udføre opgaven*. Hvis en gruppe sætter læringsambitionen lavere

end dette, ønsker jeg også at sænke brugen af ressourcer under retning.

Filerne skal afleveres elektronisk i itslearning i mappen Afleveringsopgaver under faneblad Ressourcer. Afleveringsmodulet er også sat ind i en itslearning plan i kurset.

Under afleveringen skal man erklære gruppen ved at angive alle medlemmernes navne. Man skal kun aflevere én gang per gruppe. Bemærk at man under aflevering kan oprette midlertidige "drafts", men man kan kun aflevere *én gang*.

Aflever materialet senest:

Tirsdag den 26. maj kl. 23:59.