DM507 Algoritmer og datastrukturer

Forår 2020

Projekt, del III

Python version

Institut for matematik og datalogi
Syddansk Universitet

16. april, 2020

Dette projekt udleveres i tre dele. Hver del har sin deadline, således at arbejdet strækkes over hele semesteret. Deadline for del III er fredag den 15. maj. De tre dele I/II/III er ikke lige store, men har omfang omtrent fordelt i forholdet 15/30/55. 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 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) repræsenteres bytes af 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.

> En byte er derfor otte bits i filer på disk, men er i dit Python-program 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]`.

Sammenhængen mellem heltal og bytes er som for det binære talsystem, dvs.

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

4. Skriv de 256 hyppigheder (dvs. 256 heltal) til outputfilen.

5. Scan inputfilen igen. Undervejs find for hver byte dets kodeord (ved opslag i tabellen 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 et array til videre brug, eftersom RAM-forbruget derved stiger fra $O(1)$ til inputfilens størrelse (som kan være meget stor).

I punkt 3 skal man lave et rekursivt gennemløb (svarende til 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, og når man når et blad, kan dette kodeord gemmes i tabellen. Kodeord (som jo ikke er lige lange) skal i tabellen 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. Cormen et al. 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 `key` være et tal mens `data` skal repræsentere (roden i) et træ, på en måde som vi beskriver nedenfor. 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 elements, hvordan man tilgår felterne i et element `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.

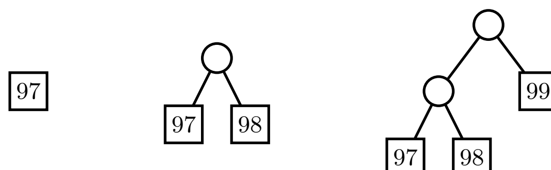
Vi beskriver nu, hvordan vi i feltet `data` for et `Element` skal repræsenterer (roden i) et træ. Vi har i del II set et eksempel på, hvordan vi kan repræsentere knuder i træer vhj. lister. Vi bruger en tilsvarende repræsentation her, men ikke helt den samme (da vi ikke arbejder med søgetræer i del III).

Træerne, som skal bruges i Huffmans algoritme, er binære træer med en byte (heltal med værdi mellem 0 og 255) gemt i blade. Se figuren i bogen side 432. Hyppigheden vist som tal i knuderne i illustrationerne i bogen skal dog ikke implementeres, da kun rodens tal bruges i Huffmans algoritme. Dette tal bliver her i stedet gemt `key` i et `Element`. Vi lader blade være lister af længde én, som indeholder det heltal, som udgør bladets byte. Vi lader indre knuder være lister af længde to, der indeholder venstre og højre barn af knuden. Man kan altså kende blade fra indre knuder på deres listelængde (længde én vs. længde to). For et blad `v` kan dens nøgle tilgås som `v[0]`, for en indre knude kan dens venstre barn og højre barn tilgås som henholdsvis `v[0]` og `v[1]`.

Med udgangspunkt i følgende hyppighedstabel

Byte	...	97	98	99	...
Hyppighed	...	1000	1500	4000	...

vises her som illustration tre eksempler på træer:



Disse tre træer og tilhørende hyppigheder skal altså gemmes i `Element`'s med følgende `key` og `data`:

`key`:	1000	2500	6500
`data`:	`[97]`	`[[97],[98]]`	`[[[97],[98]],[99]]`

Den præcise specifikation af output 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. Man kan 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 kan gøre dette. I dette er der også mulighed for at læse og skrive heltal som 32 bits. Dette skal bruges, når man laver den første del af output.

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 jeres Huffman-træ, og dem kender andre programmer ikke.[3] Kun jeres `Decode` fra opgave 2 vil kunne læse den del af output. Samme problematik gør sig gældende for første del af 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).

> Recap: 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 programmet fra opgave 1, og som skriver en fil med dens 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.

[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.

2. Genererer 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 til 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 skrive en byte repræsenteret af heltallet `b`. Det pågældende file object skal være åbnet i binary writing mode (med argumentet `'wb'` i `open`).

> Recap: 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 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

Du skal kun aflevere dine Python filer. Disse skal indeholde grundige kommentarer. De skal også indeholde navnene og SDU-logins på gruppens medlemmer. Dine programmer vil blive testet med mange typer filer (`txt`, `.doc`,

`.jpg`,...), og du bør selv gøre dette inden aflevering, men du skal ikke dokumentere disse test.

Filerne skal afleveres elektronisk i Blackboard med værktøjet "SDU Assignment", som findes i menuen til venstre på kursussiden i Blackboard. Du skal aflevere alle filer, som kræves for at køre `Encode.py` og `Decode.py`, også f.eks. `PQHeap.py` fra del I. De skal enten afleveres som individuelle filer eller som ét `zip`-arkiv. Du skal ikke aflevere noget på papir.

Aflever materialet senest:

Fredag den 15. maj, 2020, kl. 12:00.

Bemærk at aflevering af andres kode eller tekst, hvad enten kopieret fra medstuderende, fra nettet, eller på andre måder, er eksamenssnyd, og vil blive behandlet særdeles alvorligt efter gældende SDU regler. Man lærer desuden heller ikke noget. Kort sagt: kun personer, hvis navne er nævnt i den afleverede fil, må have bidraget til koden.