Institut for Matematik og Datalogi
Syddansk Universitet, Odense

24. april 2020

DM507 — Algoritmer og Datastrukturer

Eksaminatorie-timer uge 17, Forår 2020

Instruktorerne for DM507

Indhold

1 Opgave 1 - Opvarmning til projekt del III **2**

2 Opgave 2 - Eksamen juni 2010, opgave 1b **2**

3 Opgave 3 - Eksamen januar 2008, opgave 1a **4**

4 Opgave 4 - Cormen et al. opgave 16.1-4 **6**

5 Opgave 5 - Cormen et al. opgave 16.2-3 **9**

6 Opgave 6 - Cormen et al. opgave 16.1-3 **9**

7 Opgave 7 - Cormen et al. opgave 15.1-2 **11**

8 Opgave 8 - Cormen et al. opgave 16.3-3 **11**

9 Opgave 9 - Cormen et al. opgave 16.3-8 **16**

10 Opgave 10 - Cormen et al. opgave 16.1-4 **17**

1 Opgave 1 - Opvarmning til projekt del III

Tilføj en metode til DictBinTree som udskriver stierne til alle knuder i træet.

Der er lavet en løsning for Python og Java versionen af projektet. Algoritmen for Inorder-Tree-Walk [1, p. 288] er brugt med disse 3 ændringer:

- Algoritmen deles i to funktioner(Python)/metoder(Java): *in_order_walk_helper* som rekursivt kalder sig selv og printer, ligesom Inorder-Tree-Walk, og *in_order_walk_with_path* som kalder *in_order_walk_helper* med de rigtige parameter.
- Der er tilføjet en `path` parameter, som indeholder stien og bliver modificeret enten med en tilføjelse af L for left eller en tilføjelse af R for right.
- Print-statementen er ændret til at printe key'en og path'en ud.

Python:

```
1  def in_order_walk_with_path(T):
2      in_order_walk_helper("", T[0]) # Call the helper with an empty path and the root node
3
4  def in_order_walk_helper(path, node):
5      if(node is not None):
6          in_order_walk_helper(path + "L", node[1]) # Left subtree
7          print("Key "+str(node[0])+": "+path)    # Print the key and the path
8          in_order_walk_helper(path + "R", node[2]) # Right subtree
```

Java:

```
1  private BinNode root; // Root of the DictBinTree
2
3  public void in_order_walk_with_path(){
4      in_order_walk_helper("", root); // Call the helper with an empty path and the root node
5  }
6
7  private void in_order_walk_helper(String path, BinNode node){
8      if(node != null){
9          in_order_walk_helper(path + "L", node.left); // Left subtree
10         System.out.println("Key "+node.key+": "+path); // Print the key and the path
11         in_order_walk_helper(path + "R", node.right); // Right subtree
12     }
13 }
```

2 Opgave 2 - Eksamen juni 2010, opgave 1b

I følgende opgave skal vi angive et Huffman-træ for en streng med følgende tegn og tilhørende hyppigheder:

Tegn	a	b	c	d	e	f	g
Hyppighed	300	150	75	125	200	50	100

Tabel 1: De givne tegn med tilhørende hyppigheder

Vi konstruerer et muligt Huffman-træ ud fra Tabel 1 ved følgende skridt:

1. Skridt: Start med alle tegn som værende blade

```
[f: 50]  [c: 75]  [g: 100]  [d: 125]  [b: 150]  [e: 200]  [a: 300]
```

2. Skridt: Sammenlæg blade `[f: 50]` og `[c: 75]` til et nyt træ med rod `125`

```
[g: 100 ]      __125__      [d: 125]  [b: 150]  [e: 200]  [a: 300]
              /       \
           [f: 50]  [c: 75]
```

3. Skridt: Sammenlæg træet med rod `125` med blad `[g: 100]`, så vi får et træ med rod `225`

```
[d: 125]  [b: 150]  [e: 200]          __225__            [a: 300]
                                     /       \
                                  [g: 100] __125__
                                          /       \
                                       [f: 50]  [c: 75]
```

4. Skridt: Sammenlæg blade `[d: 125]` og `[b: 150]` til et nyt træ med rod `275`

```
[e: 200]            __225__            __275__            [a: 300]
                   /       \          /       \
                [g: 100] __125__   [d: 125]  [b: 150]
                        /       \
                     [f: 50]  [c: 75]
```

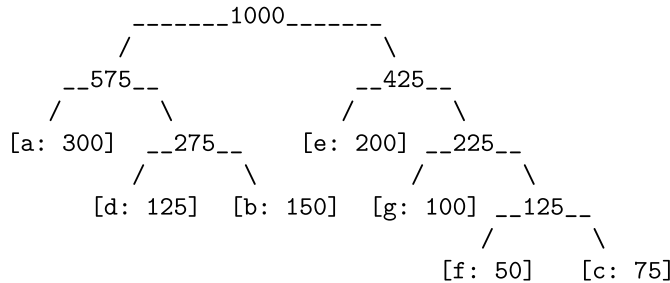
5. Skridt: Sammenlæg træet med rod `225` med blad `[e: 200]`, så vi får et træ med rod `425`

```
    __275__         [a: 300]         __425__
   /       \                        /       \
[d: 125]  [b: 150]               [e: 200] __225__
                                         /       \
                                      [g: 100] __125__
                                              /       \
                                           [f: 50]  [c: 75]
```

6. Skridt: Sammenlæg træet med rod `275` med blad `[a: 300]`, så vi får et træ med rod `575`

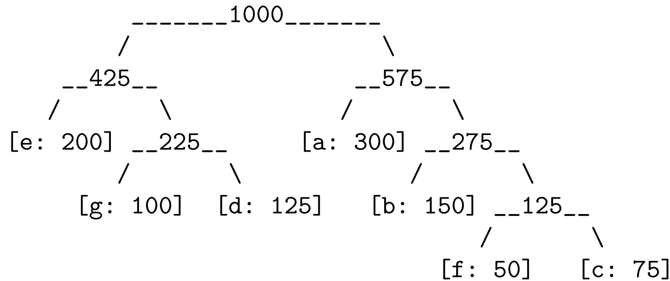
```
    __425__                           __575__
   /       \                         /       \
[e: 200]  __225__                 [a: 300] __275__
         /       \                        /       \
      [g: 100]  __125__                [d: 125]  [b: 150]
               /       \
            [f: 50]  [c: 75]
```

7. Skridt: Sammenlæg træerne med rod `425` og `575`, så vi får et træ med rod `1000`



```
# Huffman-træ T1:
         _______1000_______
        /                  \
    __575__              __425__
   /       \            /       \
[a: 300]  __275__    [e: 200] __225__
         /       \           /       \
      [d: 125]  [b: 150]  [g: 100] __125__
                                  /       \
                               [f: 50]   [c: 75]
```

Det kan bemærkes at i **2. Skridt** ville det også have været muligt at sammenlægge blade `[g: 100]` og `[d: 125]` til et nyt træ med rod `100 + 125 = 225`. I dette tilfælde, vil et andet muligt Huffman-træ blive konstrueret, som endeligt vil se ud som følger:



```
# Huffman-træ T2:
         _______1000_______
        /                  \
    __425__              __575__
   /       \            /       \
[e: 200] __225__     [a: 300] __275__
        /       \            /       \
     [g: 100]  [d: 125]   [b: 150] __125__
                                  /       \
                               [f: 50]   [c: 75]
```

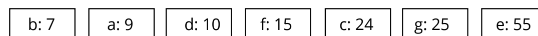
Til sidst kan det endvidere nævnes at andre mulige Huffman-træer kan konstrueres ved at bytte rundt på højre og venstre barn i een eller flere knuder i ovenstående Huffman-træer `T1` og `T2`.

3 Opgave 3 - Eksamen januar 2008, opgave 1a

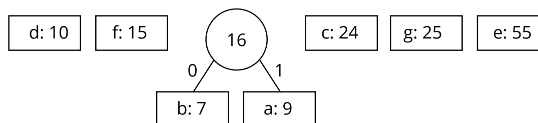
Vi følger algoritmen som beskrevet på Rolfs slides: "slå de to deltræer med de to mindste samlede frekvenser sammen".

Vi holder træerne sorteret efter samlet frekvens for tegnene i et træ (som er skrevet i roden).

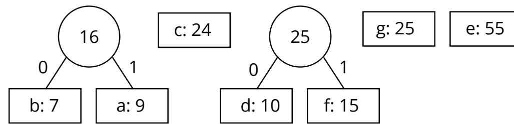
Først er alle tegn et deltræ:



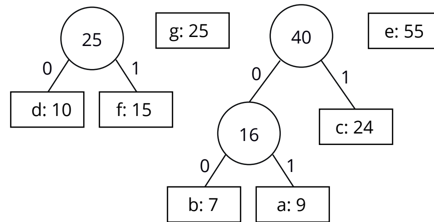
Træet med samlet frekvens 7 og træet med samlet frekvens 9 bliver slået sammen:



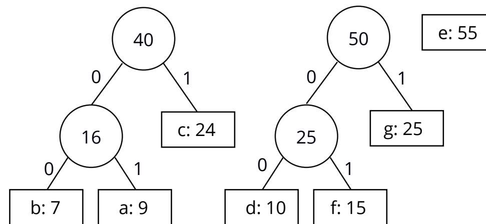
Træet med samlet frekvens 10 og træet med samlet frekvens 15 bliver slået sammen:



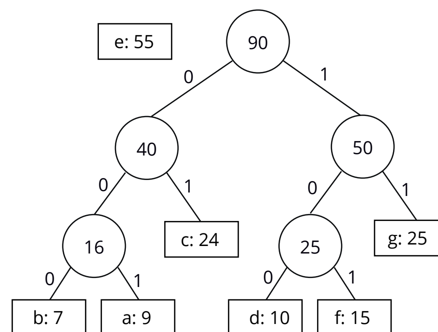
Træet med samlet frekvens 16 og træet med samlet frekvens 24 bliver slået sammen:



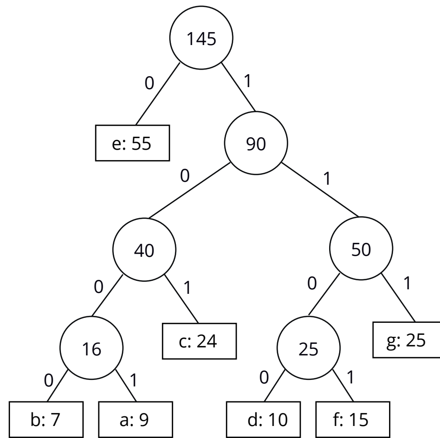
Træet med samlet frekvens 25 og træet med samlet frekvens 25 bliver slået sammen:



Træet med samlet frekvens 40 og træet med samlet frekvens 50 bliver slået sammen:



Til sidst bliver træet med samlet frekvens 55 og træet med samlet frekvens 90 slået sammen, og vi har ét træ tilbage, som er et Huffman-træ:



Da Huffmans algoritme som beskrevet på Rolfs slides ikke siger noget om, hvilket af de to træer vi sætter sammen der skal være venstre/højre barn, vil alle træer som man kan skabe ved at bytte om på venstre og højre barn for en eller flere knuder også være gyldige Huffman træer, og vil alle give koder af samme længde for tegnene.

4 Opgave 4 - Cormen et al. opgave 16.1-4

I denne opgave får vi givet en samling aktiviteter med en start- og sluttid, som skal skemalægges til forelæsningslokaler. Vores opgave er at finde en grådig algoritme som skal skemalægge alle aktiviteterne ved brug af færrest mulige lokaler. Vi antager at startiden $<$ slutiden og at kun én aktivitet kan foregå i et lokale af gangen (for ellers kunne vi bare skemalægge alle aktiviteter til det samme lokale).

På ugesedlen får vi et hint om at lade a_t være antal aktiviteter som er i gang ved tid t og lade t' være det tidspunkt, hvor a_t er maksimal. Vi skal så først argumentere for at $a_{t'}$ er en nedre grænse for antal lokaler der skal bruges. Det kan vi gøre på følgende måde.

Ved tid t skal vi mindst bruge a_t lokaler, for ellers skal 2 aktiviteter foregå i samme lokale, hvilket vi har antaget at de ikke kan[1]. Heraf følger det at $a_{t'}$ må være en nedre grænse for antal lokaler som skal bruges, , eftersom at for alle t er a_t en nedre grænse - herunder for tiden t'.

Vi skal nu finde en simpel grådig algoritme som fordeler aktiviteterne ud på $a_{t'}$ forskellige lokaler, sådan at der ikke er nogen aktiviteter som foregår samtidigt i det samme lokale. Begyndelsen på en strategi ville være at finde en måde at vise at et lokale er optaget i den periode en aktivitet er blevet tildelt det. Det kan vi vise ved at have en liste `freeRooms` til frie lokaler som vi kan tildele en aktivitet, og når aktiviteten er færdig kan vi tilføje dens lokale til listen igen.

Så ville det være naturligt at stille spørgsmålet: Hvordan kan vi vide hvornår en aktivitet er færdig i vores ene gennemløb af aktiviteterne? Vi ved at hver aktivitet både har en starttid og en sluttid. Så det vi gerne vil opnå er at ved starttiden tildeler vi aktiviteten et lokale fra

[1] (Ikke pensum) Den her måde at argumentere på er også mere formelt formuleret som 'the pigeonhole principle' som man blandt andet kan læse om i kapitel 6.2 i diskret matematik bogen[2]. 'The pigeonhole principle' er utrolig simpelt, men har rigtig mange interessante og ikke trivielle anvendelser!

`freeRooms` og ved sluttiden returnerer vi lokalet til `freeRooms`, sådan at andre aktiviteter ind imellem kan få tildelt et frit lokale. Det kan vi gøre ved at "splitte"aktiviteterne i en start del og en slut del, sammen med et ID så vi ved hvilken aktivitet hver tid startede med at høre til.

Mere præcist, en aktivitet kan ses som tuplen $(s, f, \texttt{ID})$, hvor s er starttiden, f er sluttiden og `ID` er et unikt heltal mellem $0, \ldots, n-1$ (hvor n er antal aktiviteter). Når aktiviteten bliver splittet laver vi 2 tupler en til starttiden $(s, 0, \texttt{ID})$ og en til sluttiden $(f, 1, \texttt{ID})$, hvor det midterste 1 indikere at tuplen indeholder en starttid og 0 indikere at tuplen indeholder en sluttid (dvs. typen). I en efterfølgende algoritme referer vi til samlingen af splittede aktiveter som `T`.

Når vi har aktiviteterne i den splittede form vil det være naturligt at sortere dem. Mere præcist, lade os sortere dem efter tid og type sådan at slut tiderne kommer først, hvis en start tid og en slut tid er ens. Ellers kan vi komme til at bruge et helt nyt lokale selvom en aktivitet som i samme øjeblik lige har frigivet et. Med de sorteret tider kan vi nu tildele og tage lokaler tilbage i kronologisk rækkefølge ved at gennemløbe `T` fra mindste tid til største tid.

Til sidst er spørgsmålet om hvordan vi helt præcist skal tildele lokaler og tilføje dem tilbage til `freeRooms`, da vi sagde at hvert aktivitet havde et unikt id mellem $0, \ldots, n-1$, vil det være oplagt at bruge dem som indeks til et array `assignment` sådan at `assignment[i]` er lokalet tildelt til aktiviteten med det `ID = i`. På den her måde kan vi tildele et lokale med `assignment[ID] = room` og finde lokalet igen ved at bruge `ID` når vi med en slut tuple skal tilføje lokalet til `freeRooms` igen.

Hvis vi antager at vi har splittet aktiviteterne op og sorteret dem, får vi følgende pseudokode når vi sætter resten sammen:

```
1  maxRoomNumber = 0
2  freeRooms = en tom liste af ints (lokaler)
3  assignment = et int array med n pladser (1 for hver aktivitet)
4
5  for (tid, type, ID) in T:
6      if (type == 1): // start time
7          if (|freeRooms| == 0): // alle rum optaget
8              maxRoomNumber = maxRoomNumber + 1
9              tilføj maxRoomNumber til freeRooms
10         room = fjern et element fra freeRooms
11         assignment[ID] = room
12     else: // finish time
13         tilføj assignment[ID] til freeRooms
14
15 return assignment
```

Køretiden for algoritmen er tiden det tager at sortere T, lagt sammen med køretiden for at splitte aktiviteterne og for-løkken i algoritmen ovenover. At splitte aktiviteterne må tage $O(n)$, da vi skal lave konstant arbejde for hver aktivitet, og det samme gælder for for-løkken (her skal vi dog lave konstant arbejde for $2n$ elementer, men det er asymptotisk det samme). Det vil sige at den samlede køretid er sortering + $O(n)$ eller $O(n \log n)$, hvis vi bruger en optimal sammenligningsbaseret sorteringsalgoritme som f.eks. mergesort.

For at vise **korrektheden** af algoritmen skal vi vise 2 ting:

1. At `assignment` er en korrekt fordeling af aktiviteterne ud i lokalerne, sådan at der ikke

er overlap mellem dem

2. At algoritmen er optimal, dvs. at algoritmen højst bruger $a_{t'}$ forskellige lokaler (vi har vist at $a_{t'}$ er en nedre grænse)

Vi kan se at **(1)** må holde, da lokale med indeks gående fra 1..`maxRoomNumber` enten er optaget af en aktivitet eller frit. Herudover indeholder listen `freeRooms` præcis de frie lokaler og det er kun herfra en aktivitet bliver tildelt et lokale. Derfor må det være en korrekt fordeling af aktiviteterne ud på lokalerne.

(2) kan vises med følgende invariant over for-løkken:

$$\texttt{maxRoomNumber} = a_t + |\texttt{freeRooms}|$$

Hvor t er den tid, det t'te element i T repræsenterer.

Initalization: Inden for-løkken er `maxRoomNumber` $= 0$, her må både a_t og $|$`freeRooms`$|$ være 0 da ingen aktiviteter er i gang og `freeRooms` er en tom liste. Invarianten holder derfor inden første iteration.

Maintanence: Lad os antage at invarianten holdte ved tid t_1 inden den nuværende iterationen med tid t_2. I for-løkken er der 3 cases: Enten behandles en slut tid, en start tid med frie lokaler i `freeRooms` eller en start tid uden ledige lokaler (`freeRooms` er tom).

Behandler vi en slut tid vil $a_{t_2} = a_{t_1} - 1$, da en aktivitet stopper. Samtidig tilføjer algoritmen den stoppede aktivitets lokale til `freeRooms` som derfor bliver en større. Og da `maxRoomNumber` forbliver det samme kan vi konkludere at ved en slut tid må invarianten holde til næste iteration.

Behandler vi en start tid, hvor $|$`freeRooms`$| \neq 0$ bliver aktiviteten tildelt et lokale fra listen. Hvilket vil sige, $|$`freeRooms`$|$ bliver en mindre, mens $a_{t_2} = a_{t_1} + 1$ (da en ny aktivitet starter). `maxRoomNumber` forbliver stadig det samme og derfor må invarianten også holde til næste iteration i det her case.

Til sidst kan vi have en starttid, hvor $|$`freeRooms`$| = 0$. Her bliver `maxRoomNumber` en større og der tilføjes et nyt lokale til `freeRooms` som lige efter bliver givet til aktiviteten. Derfor forbliver $|$`freeRooms`$|$ stadig 0, mens $a_{t_2} = a_{t_1} + 1$ er blevet en større (da en ny aktivitet starter). Da både `maxRoomNumber` og a_t er blevet en større må det også holde til næste iteration af for-løkken.

Vi kan derfor konkludere at invarianten holder til næste iteration i alle cases.

Termination: Hver gang `maxRoomNumber` bliver større i en iteration t, er $|$`freeRooms`$| = 0$ efter slutningen af den iteration, så pga. invarianten gælder det på det tidspunkt at `maxRoomNumber =` a_t. Hvilket kombineret med at vi kan se at variablen `maxRoomNumber` kun stiger lader os sige følgende. Lad `maxRoomNumber'` være dens endelige værdi, og lad t' være den iteration, hvor denne værdi opnås. Vi har da at `maxRoomNumber'` $= a_{t'}$, hvilket vil sige at algoritmen bruger højst $a_{t'}$ forskellige lokaler. Da vi har vist at $a_{t'}$ er en nedre grænse (dvs. at algoritmen mindst skal bruge dette antal lokaler) må antallet af brugte lokaler være optimale.

Vi kan derfor konkludere at algoritmen både er optimal og korrekt.

5 Opgave 5 - Cormen et al. opgave 16.2-3

I denne opgave kigger vi på et særligt tilfælde af *0-1 knapsack problemet*, hvor vi er givet en kapacitet af rygsækken W, n elementer og det følgende gælder: Når elementerne sorteres i stigende orden efter deres vægt, så er rækkefølgen den samme som når elementerne sorteres i faldende orden efter deres værdi. Vi kan kigge på et konkret eksempel hvor $n = 4$ og dette er tilfældet:

```
Element: E1, E2, E3, E4  <-- Elementernes rækkefølge
Værdi  : 40  30  20  10  <-- Faldende sorteret efter korresponderende værdi
Vægt   : 1   2   3   4   <-- Stigende sorteret efter korresponderende vægt
```

Vi foreslår en *grådig algoritme* til at løse problemet med og vi argumenterer herefter for at den er korrekt, dvs. at den returnerer en optimal løsning.

En oplagt grådig algoritme sorterer først elementerne efter faldende værdi (som i eksemplet ovenfor), således at elementerne forekommer i rækkefølgen:

$$E_1, E_2, E_3, \ldots, E_n. \tag{1}$$

Algoritmen tager herefter elementerne i den rækkefølge de forekommer, så længe at der er plads i rygsækken. Vi kalder resultatet $E_1, E_2, E_3, \ldots, E_k$ for L, hvor E_k er det sidste element der er plads til i rygsækken med kapacitet W.

Vi argumenterer nu for at denne grådige algoritme er korrekt. For at gøre dette ser vi på en optimal løsning OPT og opstiller elementerne i L og OPT efter faldende værdi (L er allerede opstillet på denne måde). I denne forbindelse ser vi at:

a) Det i'te element i L kan ikke have en *større vægt* end det i'te element fra OPT.

b) Det i'te element i L kan ikke have en *mindre værdi* end det i'te element fra OPT.

Grunden til dette, er at den grådige algoritme altid tager det næste element, som forekommer i rækkefølgen af elementer der er opstillet efter faldende værdi og efter stigende vægt. Med andre ord, så vil den grådige algoritme altid vælge elementer med den største værdi, hvilket betyder at det i'te element som forekommer i L ikke kan have en større vægt eller en mindre værdi, end de resterende ikke-valgte elementer som forekommer i rækken af opstillede elementer.

Vi har desuden, at på grund af **a)** kan der ikke være færre elementer i L end i OPT (ellers kunne algoritmen have taget et element mere). På grund af det forrige og **b)** kan værdien af L derfor ikke være mindre end værdien af OPT. Vi har derfor at L er optimal og algoritmen er korrekt.

6 Opgave 6 - Cormen et al. opgave 16.1-3

I denne opgave skal vi vise at det ikke er alle grådige strategier som giver et optimalt resultat når man forsøger at løse aktivitets-selektions problemet. Definitionen for aktivitets-selektions problemet kan findes i [1, side 415]. Overordnet er målet at vælge så mange aktiviteter som muligt uden at vælge to eller flere aktiviteter der har overlap.

Mere specifikt skal vi vise at disse tre strategier ikke virker optimalt.

1. Vælg aktivitet fra kompatible aktiviteter med kortest varighed.
2. Vælg aktivitet fra kompatible aktiviteter med færreste overlap i forhold tilbageværende aktiviteter.
3. Vælg aktivitet fra kompatible aktiviteter med tidligste starttid.

Ved at give mod-eksempler til hver strategi kan vi vise at de ikke giver den optimale løsning i alle tilfælde.

Strategi nr. 1:

Givet aktiviteterne i tabel 2

tid	1	2	3	4	5	6	7	8	9	10
				[4,		6]				
	[1,				5][5,					10]

Tabel 2: modbevis til strategi nr. 1

ville strategi nr. 1 vælge (start: 4, slut: 6) fordi det er den aktivitet der har kortest varighed. I starten er alle tre aktiviteter kompatible med det der allerede er valgt, fordi der ikke er valgt noget endnu. Når (start: 4, slut: 6) er valgt vil det derefter ikke være muligt at vælge en af de to andre aktiviteter fordi de overlapper med den valgte aktivitet.

Her er der så valgt én aktivitet, hvor den optimale løsning ville være at vælge (start: 1, slut: 5) og (start: 5, slut: 10) da disse to ikke overlapper og man så ville have valgt to aktiviteter i stedet for en.

Strategi nr. 2:

Givet aktiviteterne i tabel 3

tid	1	2	3	4	5	6	7	8	9
	[1,			4]		[6,			9]
	[1,			4]		[6,			9]
		[2,		4][4,		6][6,		8]	
	[1,		3][3,		5][5,		7][7,		9]

Tabel 3: modbevis til strategi nr. 2

ville strategi nr. 2 starte med at vælge (start: 4, slut: 6) da den overlapper med færrest andre aktiviteter. Herefter ville strategien så kunne vælge mellem alle andre aktiviteter med undtagelse af (start: 3, slut: 5) og (start: 5, slut: 7) da disse to overlapper med den valgte. Alle andre aktiviteter overlapper nu med tre andre aktiviteter så alle sammensætninger vil give et resultat der indeholder 3 aktiviteter.

Den optimale løsning ville være hele nederste række af de viste aktiviteter. Det at vælge hele denne række ville give et resultat med 4 aktiviteter i stedet for 3.

Strategi nr. 3:

Givet aktiviteterne i tabel 4

tid	1	2	3	4	5	6	7	8	9	10
	[1,									10]
		[2,	3][3,	4][4,	5][5,	6][6,	7][7,	8][8,	9][9,	10]

Tabel 4: modbevis til strategi nr. 3

ville strategi nr. 3 vælge (start: 1, slut: 10) da det er aktiviteten med tidligst starttidspunkt. Dette giver et resultat med kun en aktivitet, hvor det at vælge alle de andre aktiviteter giver et resultat der indeholder 8 aktiviteter.

7 Opgave 7 - Cormen et al. opgave 15.1-2

Vis med modeksempel, at den følgende grådige algoritme ikke altid vælger den optimale løsning til *Rod Cutting*-problemet. Densiteten af en stang (Engelsk: rod) af længde i er p_i/i, hvor p_i er prisen. Den grådige algoritme, for en stang af længde n, skærer et stykke af længde i af, hvor $1 \leq i \leq n$, med maksimal densitet for i. Dette fortsættes nu med den tilbageværende stykke med længde $n - i$.

Siden algoritmen kun optimerer stykket, som skæres af med længde i, så bliver det resterende stykke $n - i$ ikke betragtet eller optimeret. Derfor skal vi opstille et eksempel, hvor den optimale værdi for stykket i "ødelægger"den optimale løsning ved at "ødelægge"værdien for resten af stangen $n - i$.

Hvis vi kigger på en stang af længde 3, og følgende pristabel:

længde i	1	2	3
pris p_i	0	3	4
densitet p_i/i	0	1.5	1.333..

Algoritmen kigger på stangen af længde 3 og vælger et i, for $1 \leq i \leq 3$ således at densiteten er størst. Her er densiteten størst ved $i = 2$. Derfor vil stangen blive opdelt i en stang af længde 2, og en stang af længde 1. Da længde 2 har prisen 3, og længde 1 har prisen 0, så vil denne algoritme skabe en total pris på 3. Men hvis vi beholdte stangen som en stang af længde 3, så vil prisen være 4, som er den optimale løsning.

Altså har vi vist, med et modeksempel, at den grådige algoritme ikke altid vælger den optimale løsning.

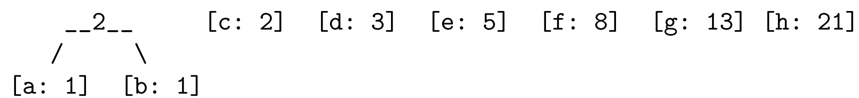
8 Opgave 8 - Cormen et al. opgave 16.3-3

I denne opgave skal vi bestemme en optimal Huffman kode, når frekvenserne er givet ved de første 8 Fibonacci tal. Benyttes Huffmans algoritme opnås følgende Huffman-træ:

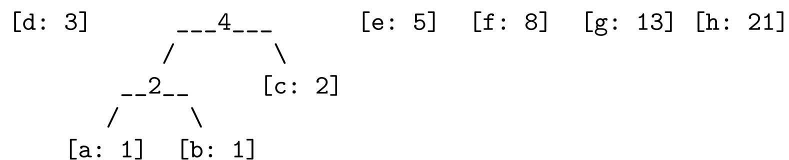
0. Gennemløb:

```
[a: 1]  [b: 1]  [c: 2]  [d: 3]  [e: 5]  [f: 8]  [g: 13] [h: 21]
```

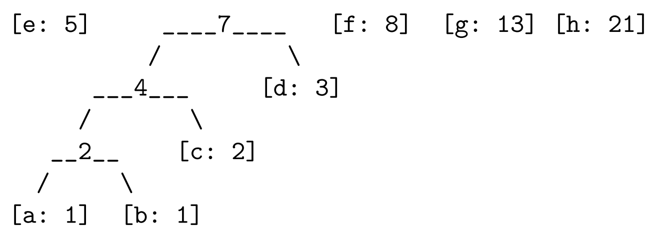
1. Gennemløb:



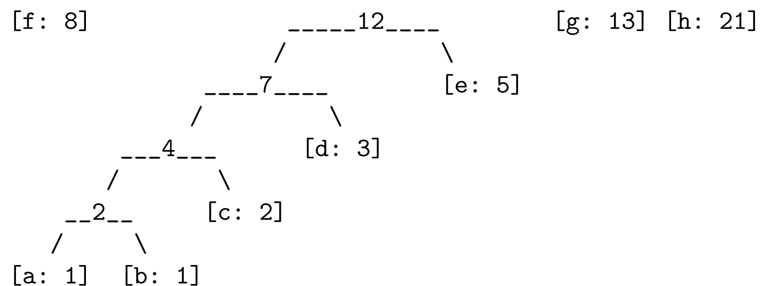
2. Gennemløb:



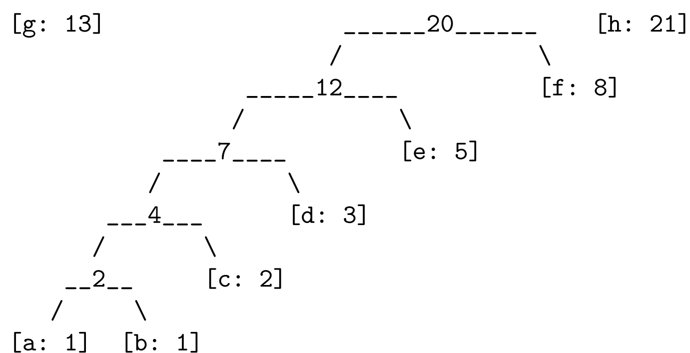
3. Gennemløb:



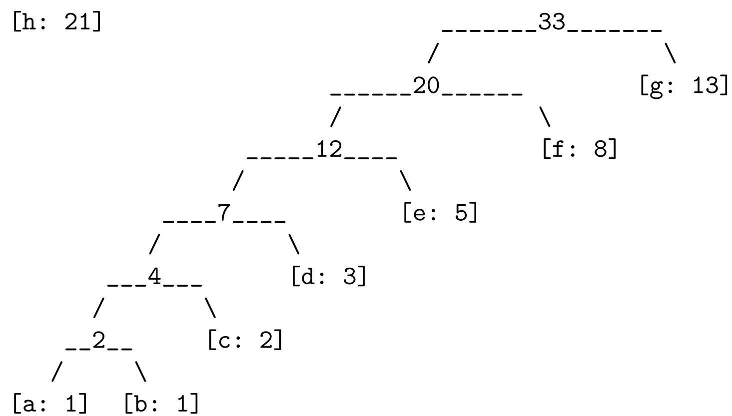
4. Gennemløb:



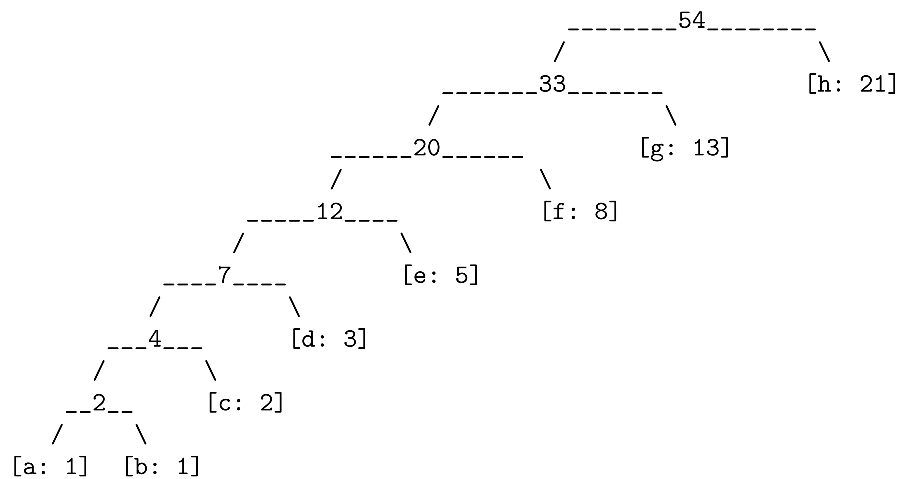
5. Gennemløb:



6. Gennemløb:



7. Gennemløb:



Ud fra Huffman-træet kan følgende Huffman-koder udledes:

- h : 1
- g : 01
- f : 001
- e : 0001
- d : 00001
- c : 000001
- b : 0000001
- a : 0000000

Fortsætter mønstret er den i'te kode givet ved

$$\text{kode}(i) = \begin{cases} \text{n-1 gange "0"} & \text{hvis } i = 1 \\ \text{"1"} & \text{hvis } i = n \\ \text{"0"} + \text{kode}(i+1) & \text{hvis } 1 < i < n \end{cases}$$

Mønstret for Huffman-koderne opstår, da der eksisterer en maksimal rod-blad sti, hvor alle indre knuder er inkluderet[2]. Hvis n frekvenser gives som input til Huffmans algoritme, så vil der være n blade i det skabte Huffman træ. Da træet der skabes er et fuldt binært træ (følger af virkemåden af Huffmans algoritme), så vil der være $n-1$ indre knuder i træet (kan bevises ved strukturel induktion). Hvis den maksimale rod-blad sti eksisterer består den af $n-1$ kanter ($n-2$ kanter mellem de indre knuder og én kant mellem den dybeste indre knude og et af dets børn, som er et blad); altså, højden af træet vil være $n-1$. Vi skal altså vise, at det skabte Huffman træ har højden $n-1$, når input er de første n Fibonacci tal. Følgende invariant ønskes bevist, hvor F er samlingen af træer Huffmans algoritme vedligeholder:

Blandt træerne i F er der et træ, der indeholder de første k Fibonacci tal, har højden $k-1$ og alle andre træer består af én knude

Initialization: Inden første gennemløb i Huffmans algoritme består F af n træer (et for hvert af de første n Fibonacci tal). Et af disse indeholder f_0, som er det første Fibonacci tal. Træet indeholdende f_0 består kun af roden, så $k = 1$ og dets højde er 0 ($= k-1$). Alle andre træer består også kun af roden og har dermed én knude.

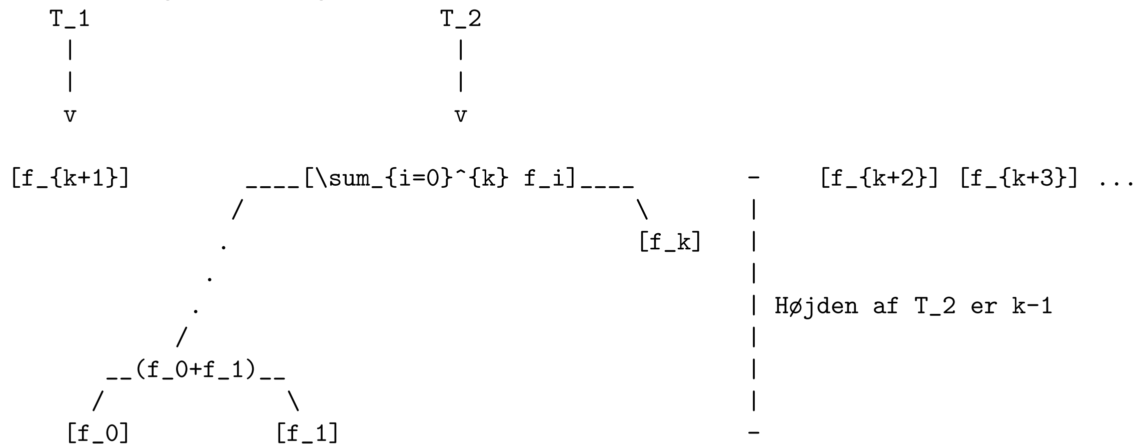
Maintenance: Lad T_1 og T_2 være de træer fra F, som Huffmans algoritme udvælger i næste iteration. Lad T' være træet, der skabes ud fra T_1 og T_2 i iterationen, og lad F' være samlingen af træer efter iterationen.

Det følger af invarianten, at der eksisterer et træ i F, som indeholder de første k Fibonacci tal - kald dette T. Frekvensen af T er summen af de første k Fibonacci tal. Jf. **Lemma 8.1** (side 15) er denne frekvens mindre end f_{k+2} (og alle Fibonacci tal efter f_{k+2}). For alle andre træer i F gælder, at antallet af knuder er én og frekvensen af disse må være et Fibonacci tal mindst lig f_{k+1} (de forrige Fibonacci er indeholdt i T). Da frekvensen af T er skarpt mindre end f_{k+2}, så må T_1 være træet med frekvensen f_{k+1} og $T_2 = T$, hvor $\textit{frekvens}(T_1) \leq \textit{frekvens}(T_2)$ (relationen følger af **Lemma 8.2** på side 15).

Da T' skabes ud fra T_1 og T_2, så vil T's højde vil være én større end højden af T_2; altså, vil dets højde være k (det følger af invarianten T_2 har højden $k-1$ og ingen af de andre træer kan være højere end det). Da T_2 indeholder de første k Fibonacci tal og T_1 indeholder det $k+1$'te Fibonacci tal, så består T' af de første $k+1$ Fibonacci tal. Derudover har Huffmans algoritme i iterationen ikke ændret på træer, som ikke er T_1 eller T_2, og dermed vil disse fortsat består af én knude.

Altså, så gælder invarianten også efter en iteration i Huffmans algoritme. I F' er T' som indeholder de første $k+1$ Fibonacci tal, har højden k og alle andre træer består af én knude.

[2]Der er flere mulige Huffman træer. Disse Huffman træer har ikke ovenstående mønster, men de kan transformeres om til et Huffman træ med samme form og dermed mønster. Dette gøres ved at bytte rundt på højre og venstre undertræ i en eller flere knuder.



Figur 1: Illustration af situationen beskrevet under Maintenance

Termination: Algoritmen slutter når der i F er et Huffman træ med n blade tilbage. Det følger af invarianten, at der i F er et træ med bestående af de første k Fibonacci tal og som har højde $k-1$. Da der kun er et træ tilbage i F, så må det betyde $k = n$ og træets højde er $n-1$.

Vi har altså bevist det skabte Huffman træ har højde $n-1$, og dermed vil vi få samme mønster for Huffman-koderne som beskrevet tidligere (evt. efter nogle ombytninger af højre og venstre undertræ i en eller flere knuder).

Lemma 8.1. *Hvis f_n er det n'te Fibonacci tal, så er $\sum_{i=0}^{n} f_i < f_{n+2}$.*

Lad $P(n)$ være

$$\sum_{i=0}^{n} f_i < f_{n+2}.$$

Da $f_0 < f_2$, så gælder P(0). For $k \geq 1$, antag $P(k-1)$. Da

$$\sum_{i=0}^{k} f_i = \sum_{i=0}^{k-1} f_i + f_k < f_{k+1} + f_k = f_{k+2}$$

følger det $P(k)$ er sandt. Dermed følger det ved simple induktion over n, at $P(n)$ er sandt for alle $n \in \mathbb{N}$.

Lemma 8.2. *Hvis f_n er det n'te Fibonacci tal, så er $f_{n+1} \leq \sum_{i=0}^{n} f_i$.*

Lad $P(n)$ være

$$f_{n+1} \leq \sum_{i=0}^{n} f_i.$$

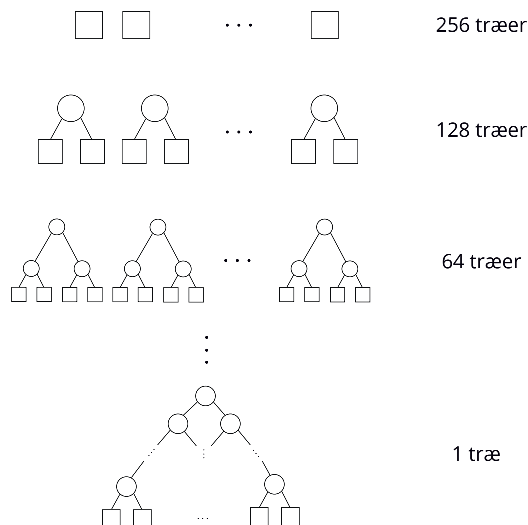
Da $f_1 \leq f_0$, så gælder P(0). For $k \geq 0$, antag $P(k)$. Da

$$\begin{aligned} f_{k+1} &\leq \sum_{i=0}^{k} f_i \Leftrightarrow \\ f_{k+1} + f_k &\leq \sum_{i=0}^{k} f_i + f_{k+1} \Leftrightarrow \qquad \text{udnytter } f_k \leq f_{k+1} \\ f_{k+2} &\leq \sum_{i=0}^{k+1} f_i \end{aligned}$$

følger det $P(k+1)$ er sandt. Dermed følger det ved simple induktion over n, at $P(n)$ er sandt for alle $n \in \mathbb{N}$.

9 Opgave 9 - Cormen et al. opgave 16.3-8

Når vi udfører Huffman's algoritme, har vi til at starte med et træ for hvert tegn, dvs. vi har 256 træer. Vi kender ikke frekvenserne af hvert tegn, men vi ved at den højeste frekvens er mindre end 2 gange den mindste frekvens. Lad os kalde den mindste frekvens k. Siden den højeste har frekvens mindre end 2 gange den mindste, så må den højeste frekvens være $< 2k$. Dvs. alle frekvenserne ligger i intervallet $[k, 2k)$. Huffman's algoritme tager nu de to træer med mindst samlede frekvens og lægger sammen til et nyt træ. Dette nye træs samlede frekvens må ligge i intervallet $[2k, 4k)$ (da begge frekvenser i dette træ er i intervallet $[k, 2k)$). Vi observerer nu, at næste gang vi skal tage de 2 træer med mindst samlet frekvens, så kan dette nye træ ikke være en af dem, da alle de andre træer har samlet frekvens $< 2k$, og det nye træ har samlet frekvens $\geq 2k$. Dette fortsætter indtil alle 256 træer er blevet slået sammen til 128 træer, som alle har samlet frekvens i intervallet $[2k, 4k)$. Med 128 træer tilbage, vil de to første træer der bliver samlet til et nyt hver have samlet frekvens i intervallet $[2k, 4k)$, og det nye træ vil have samlet frekvens i intervallet $[4k, 8k)$. Det samme argument som før fortæller os, at alle træerne med frekvenser i intervallet $[2k, 4k)$ vil blive slået sammen først. Dette fortsætter indtil vi er nede på kun at have ét træ tilbage:



I dette træ vil alle blade have samme dybde, dvs. alle tegn vil have samme længde kode. Længden af denne kode er dybden af et blad (dvs. antal kanter fra bladet til roden). Hver gang vi sætter 2 træer sammen stiger dybden af bladene i træet med én. Vi havde 256 træer til at starte med, og efter at alle disse blev sat sammen havde vi 128, så blev de sat sammen igen, og vi havde 64, osv. Dvs. hver gang vi halverer antallet af træer, stiger dybden af bladene med én. Vi halverer antallet af træer indtil vi har ét træ tilbage, dvs. hvis h er antal halveringer, så finder vi h ved:

$$\frac{256}{2^h} = 1 \Leftrightarrow 256 = 2^h \Leftrightarrow h = \lg(256) = 8$$

Dvs. antallet af halveringer vi laver er 8, og det er dermed også dybden af alle blade, og dermed antallet af bits Huffman kodningen skal bruge for at repræsentere hvert tegn. Det er altså ikke mere effektivt at bruge Huffman kodningen fremfor den almindelige fixed-length, da de bruger samme antal bits på at repræsentere hvert tegn, og den endelig fil vil derfor have den samme størrelse.

10 Opgave 10 - Cormen et al. opgave 16.1-4

I den her opgave får vi givet en mængde af punkter på den reelle tallinje $\{x_1, x_2, \ldots, x_n\}$, ud fra disse tal skal vi finde det mindste antal intervaller som er "unit-length closed intervals", dvs. intervallerne skal have følgende form

$$[x, 1 + x] = \{k \in \mathbb{R} \mid x \leq k \leq 1 + x\}$$

Sådan at alle punkterne $x_1, x_2, \ldots, x_n$ er inde for et af de fundne intervaller.

En simpel grådig ide kunne være at sortere punkterne og så punkt efter punkt tjekke om vi har et interval som dækker hver især. Finder vi et punkt x_i som ikke er dækket tilføjer vi bare et nyt interval som starter fra x_i. Det kan mere præcist skrives med følgende pseudokode:

```
1  Sorter input (og lad {x_1, x_2,...x_n} være input i sorteret orden)
2  V = ∅
3  for i=1 to n:
4      if x_i ikke er dækket af intervaller i V:
5          tilføj intervallet [x_i, 1 + x_i] til V
6  return V
```

Den lidt svære del er nu vise at dette er en korrekt algoritme som returnerer det optimale antal intervaller. Til at hjælpe os får vi et hint på ugesedlen til en invariant: De hidtil valgte intervaller er en delmængde af en optimal dækning. Herudover bør vi også vise at V dækker alle punkterne.

Lad V_i være V efter i iterationer af for-løkken, så ville en lidt mere præcis formulering af invarianten være følgende:

1. V_i dækker punkterne $x_1, ..., x_i$
2. Der eksisterer en optimal løsning OPT_i som indeholder V_i

Den første del af invarianten følger af hver gang algoritmen møder et punkt som ikke er blevet dækket så tilføjes et nyt interval til V_i som dækker det. Siden algoritmen kigger på alle punkter fra x_1 til x_i må alle i punkter være dækket af V_i.

Vi kan bevise anden del af vores invariant med et induktionsbevis:

Basis: Inden for-løkken er $i = 0$ og $V = \emptyset$. Invarianten må derfor være opfyldt, da den tomme mængde er en delmængde af alle mængder.

Induktionsskridt: Vi antager at invarianten holder for $i - 1$, hvor $i \geq 1$ og viser at den holder for i:

Der er to cases alt efter om algoritmen går ind i if-sætningen.

Hvis if-sætningen *ikke* bliver udført var x_i allerede dækket af et interval, det vil sige at $V_i = V_{i-1}$ og ifølge induktionsantagelsen må en optimal løsning allerede indeholde V_{i-1}.

Hvis if-sætningen udføres er $V_i = V_{i-1}$ med intervallet $[x_i, x_i + 1]$ tilføjet. Der må eksistere en optimal løsning OPT_{i-1} som indeholder V_{i-1} (pga. induktionsantagelsen) og dækker x_i med mindst ét interval I (da OPT_{i-1} er en dækning). Lad OPT_i være OPT_{i-1} med I erstattet af intervallet $[x_i, x_i + 1]$. Intervallet I kan ikke være en del af V_{i-1}, da vi ved at V_{i-1} ikke dækker x_i, for ellers ville if-sætningen ikke være blevet udført. Derfor må OPT_i stadig indeholde hele V_{i-1} og dermed også V_i. Dette følger af at OPT_i må indeholde V_{i-1} og $[x_i, x_i + 1]$, hvilket er definitionen af V_i.

Vi har nu vist at OPT_i indeholder V_i, men vi bliver også nødt til at vise at der ikke blev fjernet dækning af punkter ved at erstatte I med intervallet $[x_i, x_i + 1]$.

Det kan ses da $x_1, ..., x_{i-1}$ er dækket af V_{i-1} (som er indeholdt af OPT_i) og x_i er dækket af det tilføjet interval $[x_i, x_i + 1]$. Derfor kan udskiftningen af I med $[x_i, x_i + 1]$ kun fjerne dækning i området imellem punkterne x_{i-1} og x_i. Med andre ord intervallet (x_{i-1}, x_i)[3], men der kan ikke være nogle tal mellem punkterne x_{i-1} og x_i, fordi $x_1, x_2, ...$ er punkterne i sorteret orden. Desuden er OPT_i stadig optimal, da $|OPT_i| = |OPT_{i-1}|$.

Når for-løkken slutter, viser invarianten at output $V = V_n$ er en optimal dækning: Den ligger inden i en optimal dækning pga. anden del af invarianten, og kan ikke være ægte mindre end denne, da den selv er en dækning pga. første del af invarianten.

[3] Intervallet (x_{i-1}, x_i) kan også skrives som $\{k \in \mathbb{R} \mid x_{i-1} < k < x_i\}$

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] K.H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, eighth edition, 2018.