Huffman Coding

Data Compression

In data compression is reducing the space required to storing a piece of data.

- We can replace recurrent elements in the data with a unique representation for each distinct element, which will require lesser space.
- Based on the unique representations we used to denote each element, the compressed data can be decompressed to get the original message

Encoding

Given a code (corresponding to some alphabet C) and a message the message is encoded by replacing the characters with the codewords

How can we represent a-f, using fixed length codes?

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Encoding

Using fixed length codes:

6 characters → 3 bits are required for each code

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code						

Encoding

File size to store 100,000 characters?

= 300,000

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code	000	001	010	011	100	101

Encoding

Variable length codewords

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code	000	001	010	011	100	101
Variable length codes	0	101	100	111	1101	1100

Encoding

File size

= (45.1 + 13.3 + 12.3 + 16.3 + 9.4 + 5.4).1000 = 224 000

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code	000	001	010	011	100	101
Variable length codes	0	101	100	111	1101	1100

Decoding

Given an encoded message, decoding is the process of turning it back into the original message.

- A message is uniquely decodable if it can only be decoded in one way.
- The unique decipherability property is needed in order for a code to be useful.

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code	000	001	010	011	100	101
Variable length codes	0	101	100	111	1101	1100

Decode the sequence: 001011101

Prefix Codes

- No codeword is also a prefix of some other codeword
- Prefix codes are desirable because they simplify decoding.
    - In encoding; we just concatenate the codewords representing each character of the file
    - Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

Huffman Codes

Huffman codes are a optimal code of variable length codewords which can be used for lossless data compression.

Here, we consider the data to be a sequence of characters;

Huffman's greedy algorithm uses a table giving how often each character occurs **(frequency) to build up an optimal** way of representing each character as a unique binary string (binary character code) which is called a **codeword**.

These can be used to compress data very effectively

- savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

Prefix Codes

Given a tree corresponding to the prefix codes, we can easily decode a character sequence

Let C be the alphabet, a set of n characters and that each character $c \in C$ is an object with attribute c.freq

The tree (T) for an optimal prefix code has

- exactly n leaves and n - 1 internal nodes

For each character c

- $d_T(c)$ denote the depth of the c's node → length of the codeword for c

Prefix Codes

For each character c in the alphabet C,

- the attribute c.freq denote the frequency of c in the file
- $d_T(c)$ denote the depth of the c's node → length of the codeword for c

The number of bits required to encode a file is,

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

This is also called the cost of the tree T.

Huffman Codes - Algorithm

```
HUFFMAN(C)
n ← |C|
Q ← C                              // Q is a minimum priority queue, for frequency attribute
for  i ← 1 to n - 1
       do  ALLOCATE-NODE(z)
       left[z] ← x ← EXTRACT-MIN(Q)
              right[z] ← y ← EXTRACT-MIN(Q)
              f[z] ← f[x] + f[y]                    // f[c]is frequency of character c
              INSERT(Q, z)
return EXTRACT-MIN(Q)
```

Huffman Codes - Algorithm

	1	2	3	4	5	6
Character	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

HUFFMAN(C)

1 $n \leftarrow |C|$

2 $Q \leftarrow C$

3 for $i \leftarrow 1$ to n - 1

4 ALLOCATE-NODE(z)

5 $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)

6 $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)

7 $f[z] \leftarrow f[x] + f[y]$

8 INSERT(Q, z)

9 return EXTRACT-MIN(Q)

Huffman Codes Algorithm - Runtime

Assumes that Q is implemented as a binary min-heap.

For a set C of n characters, the initialization of Q can be performed in O(n) time using the BUILD-MIN-HEAP procedure.

The for loop is executed exactly |n| - 1 times. Each heap operation requires time O(log n).

The loop contributes

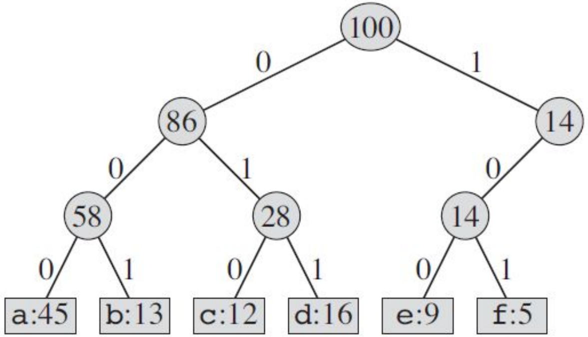
$$= (|n|-1)*O(\log n) = O(n \log n)$$

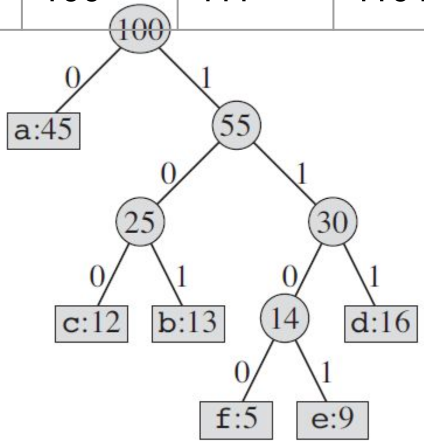
Thus, the total running time of HUFFMAN on a set of n characters

$$= O(n) + O(n\log n) = O(n \log n)$$

Huffman Codes

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code	000	001	010	011	100	101
Variable length codes	0	101	100	111	1101	1100

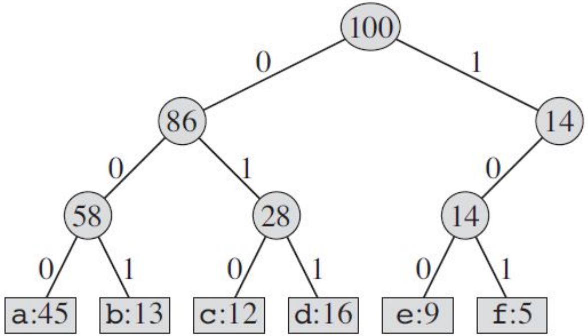


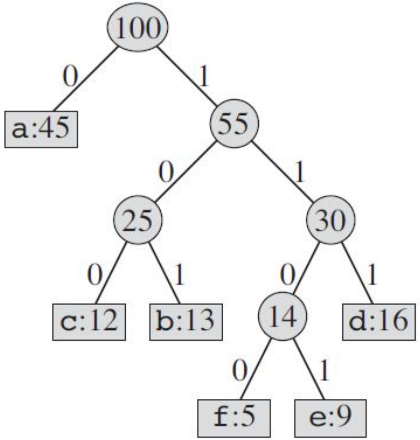


Huffman Codes

An optimal code for a file is always represented by a ***full* binary tree**, in which every non-leaf node has two children.

The fixed-length code in our example is not optimal since its tree, is not a full binary tree.





Correctness of the Algorithm

Greedy choice property:

x and y be two characters in C having the lowest frequencies

- Then the codewords for x and y have the same length and differ only in the last bit.

Correctness of the Algorithm

Optimal substructure property:

If x and y two characters in C having the lowest frequencies

- Then let C’ = C - {x,y} ∪ {z} where f(z) = f(x) + f(y)
- Let T’ be the tree representing the optimal code for C’

We can obtain the T

- by removing leaf node z, and replace with internal node having x, y as leaf nodes
- which represent optimal code for C

0/1 Knapsack Problem

A thief robbing a store finds n items, each item, i worth \$ i and weighs w_i kg
The thief wants to take as valuable a load as possible, in his knapsack which can carry **at most W**

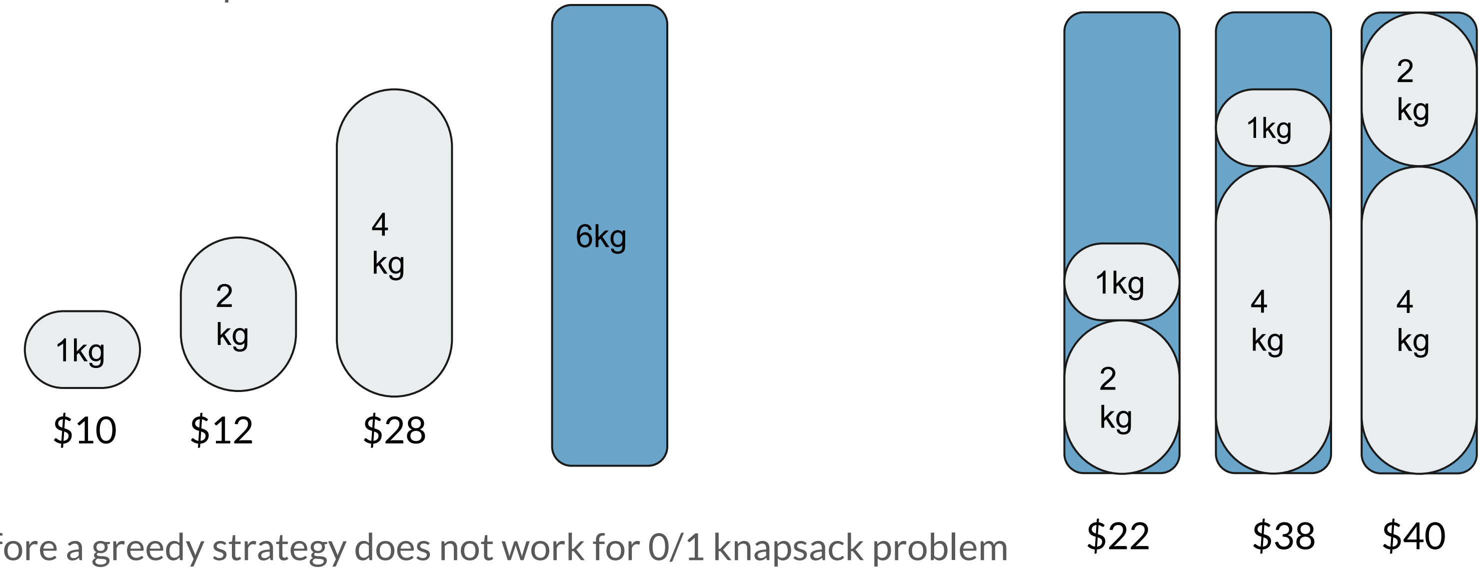
Which items should he take?

-

0/1 Knapsack Problem

Greedy choice: item with **max value per kg**.

If we select item 1 with the highest value per kg, knapsack is not filled to its capacity and solutions with item 1 are suboptimal.

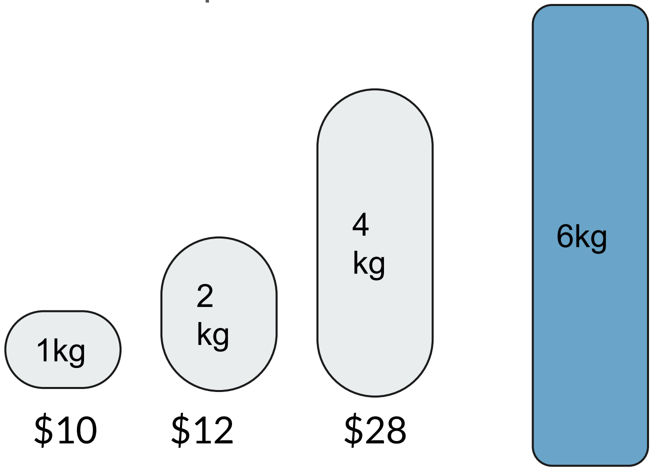


Therefore a greedy strategy does not work for 0/1 knapsack problem

0/1 Knapsack Problem

Greedy choice: item with **max value per kg**.

If we select item 1 with the highest value per kg, knapsack is not filled to its capacity and solutions with item 1 are suboptimal.



Therefore a greedy strategy does not work for 0/1 knapsack problem

Fractional Knapsack Problem

Knapsack(w[1..n], v[1..n], W): array [1..n]

    //assumed w & v such that v[i]/w[i] > v[j]/w[j] for i < j

for i = 1 to n

    x[i] = 0; w_t = 0;

while w_t < W do

    i = best remaining item

    if w_t + w[i] $\leq$ W then   //w[i] is the greedy choice

        x[i] = 1; wt = wt + w[i]

    else x[i] = (W - w_t)/w[i]

        w_t = W

return x

Fractional Knapsack Problem

If the thief can take fractions of items, then the greedy algorithm will yield an optimal solution