Chapter 17: Greedy Algorithms

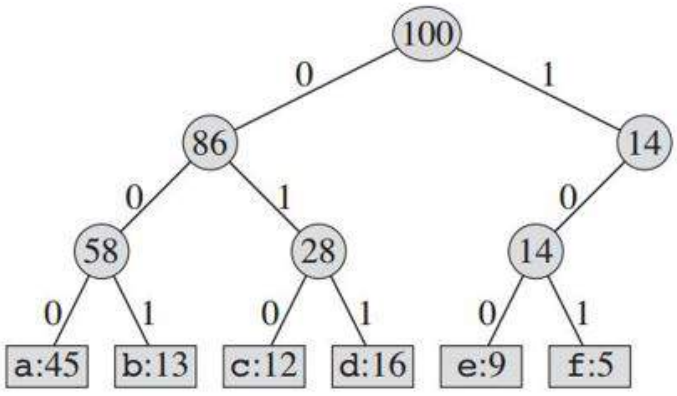
Section 17.1: Huffman Coding

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Huffman code was proposed by David A. Huffman in 1951.

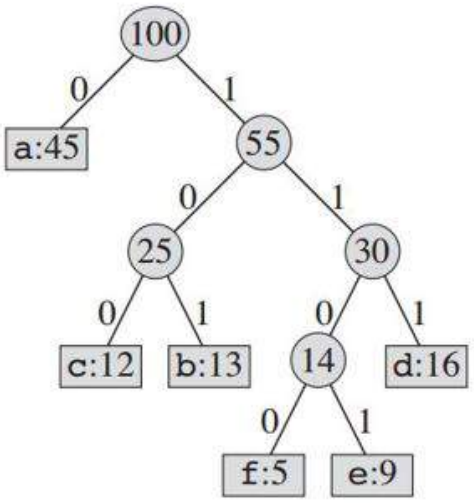
Suppose we have a 100,000-character data file that we wish to store compactly. We assume that there are only 6 different characters in that file. The frequency of the characters are given by:

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

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *Binary Character Code* in which each character is represented by a unique binary string, which we call a **codeword**.



Fixed-length Codeword



Variable-length Codeword

The constructed tree will provide us with:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

If we use a **fixed-length code**, we need three bits to represent 6 characters. This method requires 300,000 bits to code the entire file. Now the question is, can we do better?

A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. This code requires: (45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 + 5 X 4) X 1000 = 224000 bits to represent the file, which saves approximately 25% of memory.

One thing to remember, we consider here only codes in which no codeword is also a prefix of some other codeword. These are called *prefix codes.* For variable-length coding, we code the 3-character file *abc* as 0.101.100 = 0101100, where "." denotes the concatenation.

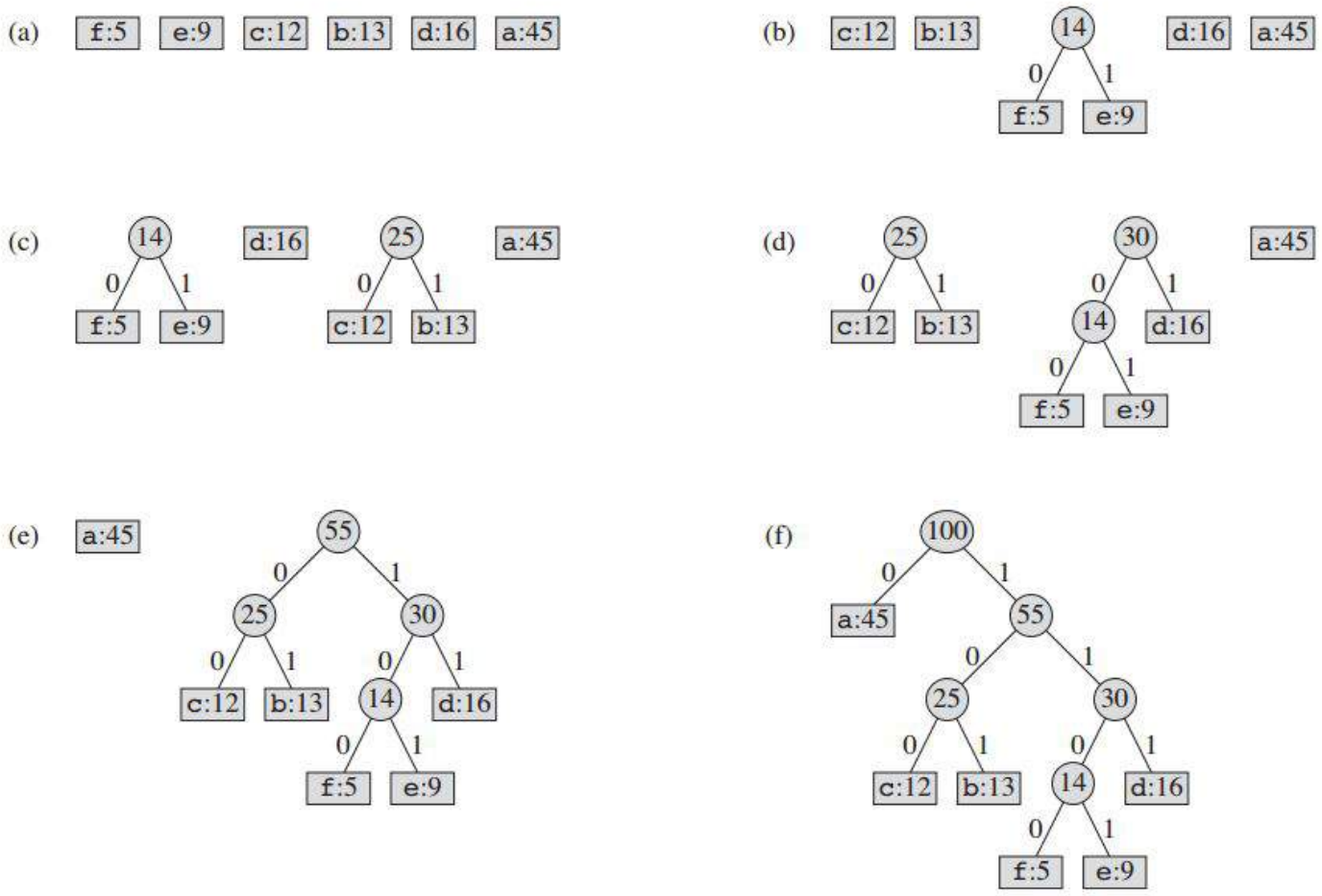
Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. For example, 001011101 parses uniquely as 0.0.101.1101, which decodes to *aabe*. In short, all the combinations of binary representations are unique. Say for example, if one letter is denoted by 110, no other letter will be denoted by 1101 or 1100. This is because you might face confusion on whether to select 110 or to continue on concatenating the next bit and select that one.

Compression Technique:

The technique works by creating a *binary tree* of nodes. These can stored in a regular array, the size of which depends on the number of symbols, **n**. A node can either be a *leaf node* or an *internal node*. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. *Priority queue* is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the two nodes of highest priority from the queue.
    2. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.
    3. Add the new node to the queue.
3. The remaining node is the root node and the Huffman tree is complete.

For our example:



The pseudo-code looks like:

```
Procedure Huffman(C):     // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
    Z.left = x = Q.pop
    Z.right = y = Q.pop
    Z.frequency = x.frequency + y.frequency
    Q.push(Z)
end while
Return Q
```

Although linear-time given sorted input, in general cases of arbitrary input, using this algorithm requires pre-sorting. Thus, since sorting takes **O(nlogn)** time in general cases, both methods have same complexity.

Since **n** here is the number of symbols in the alphabet, which is typically very small number (compared to the length of the message to be encoded), time complexity is not very important in the choice of this algorithm.

Decompression Technique:

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired

character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately. The pseudo-code:

```
Procedure HuffmanDecompression(root, S):   // root represents the root of Huffman Tree
n := S.length                              // S refers to bit-stream to be decompressed
for i := 1 to n
    current = root
    while current.left != NULL and current.right != NULL
        if S[i] is equal to '0'
            current := current.left
        else
            current := current.right
        endif
        i := i+1
    endwhile
    print current.symbol
endfor
```

Greedy Explanation:
Huffman coding looks at the occurrence of each character and stores it as a binary string in an optimal way. The idea is to assign variable-length codes to input input characters, length of the assigned codes are based on the frequencies of corresponding characters. We create a binary tree and operate on it in bottom-up manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

References:

- Introduction to Algorithms - Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen
- Huffman Coding - Wikipedia
- Discrete Mathematics and Its Applications - Kenneth H. Rosen

Section 17.2: Activity Selection Problem

The Problem

You have a set of things to do (activities). Each activity has a start time and a end time. You aren't allowed to perform more than one activity at a time. Your task is to find a way to perform the maximum number of activities.

For example, suppose you have a selection of classes to choose from.

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

Remember, you can't take two classes at the same time. That means you can't take class 1 and 2 because they share a common time 10.30 A.M to 11.00 A.M. However, you can take class 1 and 3 because they don't share a common time. So your task is to take maximum number of classes as possible without any overlap. How can you do that?

Analysis

Lets think for the solution by greedy approach.First of all we randomly chose some approach and check that will work or not.

- **sort the activity by start time** that means which activity start first we will take them first. then take first to last from sorted list and check it will intersect from previous taken activity or not. If the current activity is not intersect with the previously taken activity, we will perform the activity otherwise we will not perform. this approach will work for some cases like

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	11.00AM

the sorting order will be 4-->1-->2-->3 .The activity 4--> 1--> 3 will be performed and the activity 2 will be skipped. the maximum 3 activity will be performed. It works for this type of cases. but it will fail for some cases. Lets apply this approach for the case

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	3.00P.M

The sort order will be 4-->1-->2-->3 and only activity 4 will be performed but the answer can be activity 1-->3 or 2-->3 will be performed. So our approach will not work for the above case. Let's try another approach

- **Sort the activity by time duration** that means perform the shortest activity first. that can solve the previous problem . Although the problem is not completely solved. There still some cases that can fail the solution. apply this approach on the case bellow.

Activity No.	start time	end time
1	6.00 A.M	11.40A.M
2	11.30 A.M	12.00P.M
3	11.40 P.M	2.00P.M

if we sort the activity by time duration the sort order will be 2--> 3 --->1 . and if we perform activity No. 2 first then no other activity can be performed. But the answer will be perform activity 1 then perform 3 . So we can perform maximum 2 activity.So this can not be a solution of this problem. We should try a different approach.

The solution

- **Sort the Activity by ending time** that means the activity finishes first that come first. the algorithm is given below

> 1. Sort the activities by its ending times.
> 2. If the activity to be performed do not share a common time with the activities that previously performed, perform the activity.

Lets analyse the first example

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

sort the activity by its ending times , So sort order will be 1-->5-->2-->4-->3.. the answer is 1-->3 these two activities will be performed. ans that's the answer. here is the sudo code.

> 1. sort: activities
> 2. perform first activity from the sorted list of activities.
> 3. Set : Current_activity := first activity
> 4. set: end_time := end_time of Current activity
> 5. go to next activity if exist, if not exist terminate .
> 6. if start_time of current activity <= end_time : perform the activity and go to 4
> 7. else: got to 5.

see here for coding help http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

Section 17.3: Change-making problem

Given a money system, is it possible to give an amount of coins and how to find a minimal set of coins corresponding to this amount.

Canonical money systems. For some money system, like the ones we use in the real life, the "intuitive" solution works perfectly. For example, if the different euro coins and bills (excluding cents) are 1€, 2€, 5€, 10€, giving the highest coin or bill until we reach the amount and repeating this procedure will lead to the minimal set of coins.

We can do that recursively with OCaml :

```
(* assuming the money system is sorted in decreasing order *)
let change_make money_system amount =
  let rec loop given amount =
    if amount = 0 then given
    else
      (* we find the first value smaller or equal to the remaining amount *)
      let coin = List.find ((>=) amount) money_system in
      loop (coin::given) (amount - coin)
  in loop [] amount
```

These systems are made so that change-making is easy. The problem gets harder when it comes to arbitrary money system.

General case. How to give 99€ with coins of 10€, 7€ and 5€? Here, giving coins of 10€ until we are left with 9€ leads obviously to no solution. Worse than that a solution may not exist. This problem is in fact np-hard, but acceptable solutions mixing **greediness** and **memoization** exist. The idea is to explore all the possibilies and pick the one with the minimal number of coins.

To give an amount X > 0, we choose a piece P in the money system, and then solve the sub-problem corresponding to X-P. We try this for all the pieces of the system. The solution, if it exists, is then the smallest path that led to 0.

Here an OCaml recursive function corresponding to this method. It returns None, if no solution exists.

```
(* option utilities *)
let optmin x y =
  match x,y with
  | None,a | a,None -> a
  | Some x, Some y-> Some (min x y)

let optsucc = function
  | Some x -> Some (x+1)
  | None -> None

(* Change-making problem*)
let change_make money_system amount =
  let rec loop n =
    let onepiece acc piece =
      match n - piece with
      | 0 -> (*problem solved with one coin*)
             Some 1
      | x -> if x < 0 then
               (*we don't reach 0, we discard this solution*)
               None
             else
               (*we search the smallest path different to None with the remaining pieces*)
               optmin (optsucc (loop x)) acc
    in
    (*we call onepiece forall the pieces*)
    List.fold_left onepiece None money_system
  in loop amount
```

Note: We can remark that this procedure may compute several times the change set for the same value. In practice, using memoization to avoid these repetitions leads to faster (way faster) results.