# Tutorial #4 Greedy Algorithms
# Knapsack Problem

## Fractional Knapsack Problem

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

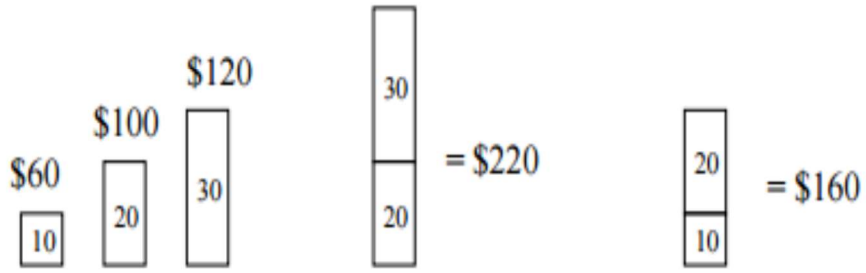
An **efficient solution** is to use Greedy approach. The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can't add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.

- Problems appear very similar, but only FRACTIONAL KNAPSACK PROBLEM can be solved greedily:
  - Compute value per pound $\frac{v_i}{w_i}$ for each item
  - Sort items by value per pound.
  - Fill knapsack greedily (take objects in order)

    $\Downarrow$

    $O(n \log n)$ time, easy to show that solution is optimal.

- Example that $0-1$ KNAPSACK PROBLEM cannot be solved greedily:




Items in value per pound order | Optimal solution for knapsack of size 50 | Greedy solution for knapsack of size 50

Note: In FRACTIONAL KNAPSACK PROBLEM we can take $\frac{2}{3}$ of \$120 object and get \$240 solution.

**Consider the input for algorithm is w,v,W**

**w is array of weights of all items**

**V is array of prices of all items**

**W is total weight of knapsack**

**Greedy-fractional-knapsack (*w, v, W*)**

**FOR *i* =1 to *n***
    do $x[i] = 0$
weight = 0
while weight < $W$
    do $i$ = best remaining item
        IF weight + $w[i] \leq W$
            then $x[i] = 1$
                weight = weight + $w[i]$
            else
                $x[i] = (w$ - weight) / $w[i]$
                weight = $W$
return $x$

## Analysis

If the items are already sorted into decreasing order of $v_i / w_i$, then

the while-loop takes a time in $O(n)$;

Therefore, the total time including the sort is in $O(n \log n)$.

## Fractional Knapsack Problem exhibits Greedy Choice Property

## Proof:

First show that as much as possible of the highest value/pound item must be included in the optimal solution.

Let $w_h$, $v_h$ be the weight available and value of the item with the highest value/pound ratio (item h).

Let L(i) be the weight of item i contained in the thief's loot L.

$$\text{Total Value V} = \sum_{i=1}^{n} L(i)\frac{v_i}{w_i}$$

If some of item h is left, and $L(j) \neq 0$ for some $j \neq h$, then replacing j with h will yield a higher value.

$$L(j)\ \frac{v_j}{w_j} \leq L(j)\frac{v_h}{w_h}$$

$\frac{v_j}{w_j} \leq \frac{v_h}{w_h}$ True by definition of h

## Activity Selection Problem

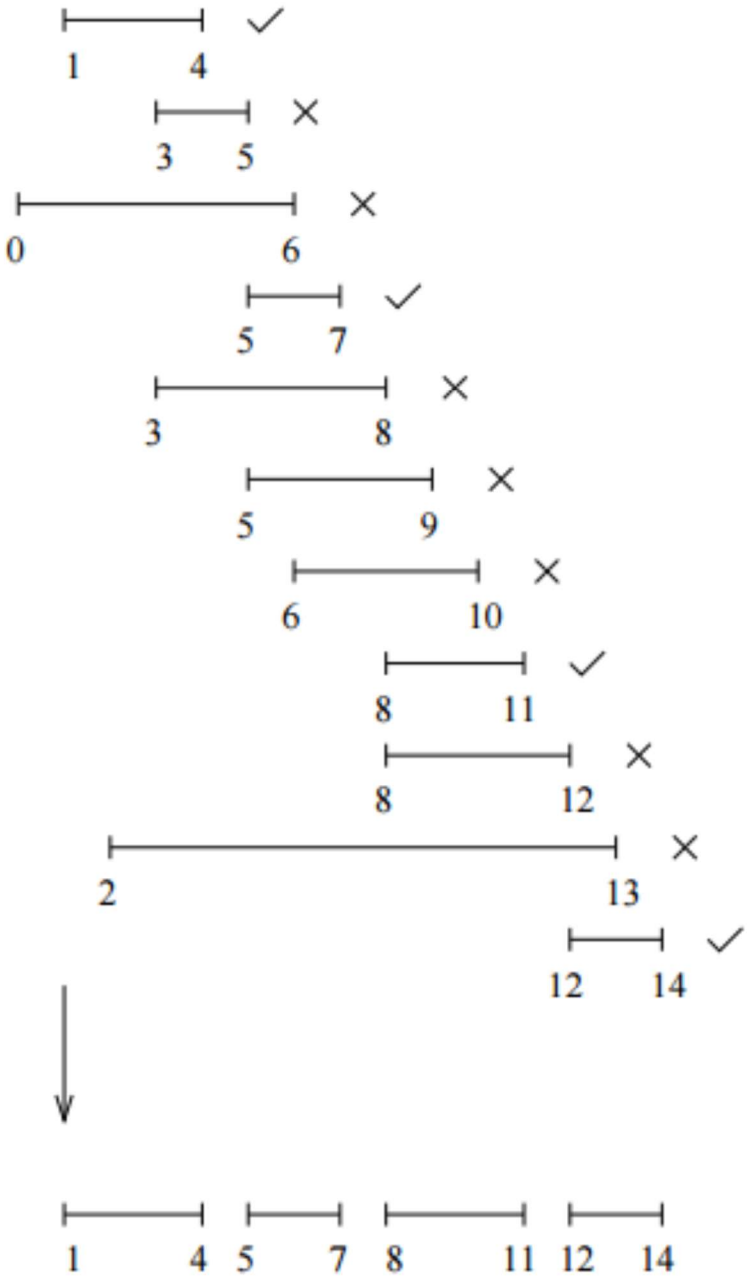
Problem: Given a set A = {A1, A2, · · · , An} of n activities with start and finish times (si , fi), $1 \leq i \leq n$, select maximal set S of "non-overlapping" activities.

Greedy solution:

- Sort activity by finish time (let A1, A2, · · · , An denote sorted sequence).
- Pick first activity A1.
- Remove all activities with start time before finish time of A1.
- Recursively solve problem on remaining activities.
- Running time is obviously O (n log n).

Example:

- 11 activities sorted by finish time: $(1,4), (3,5), (0,6), (5,7), (3,8), (5,9), (6,10), (8,11), (8,12), (2,13), (12,14)$



# Huffman Algorithm

Huffman codes compress data very effectively: savings of 20% to 90% are typical, 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.

Suppose we have a 100,000-character data file observe that the characters in the file occur with the frequencies given

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

That is, only 6 different characters appear, and the character a occurs 45,000 times. We have many options for how to represent such a file of information. Here,
we consider the problem of designing a ***binary character code*** (or ***code*** for short) in which each

character is represented by a unique binary string, which we call a ***codeword***.

If we use a ***fixed-length code***, we need 3 bits to represent 6 characters:
a = 000, b = 001, . . . , f = 101. This method requires 300,000 bits to code the entire file.

A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short code words and infrequent characters long code words.

**There is a problem in variable length codes**

For Example
If we encoded characters as following
A 0
B 01
C 011
D 00
E 1
If we have the code 0000000 when decode it
It Can be AAAAAA or DAAD or DDD or AADD

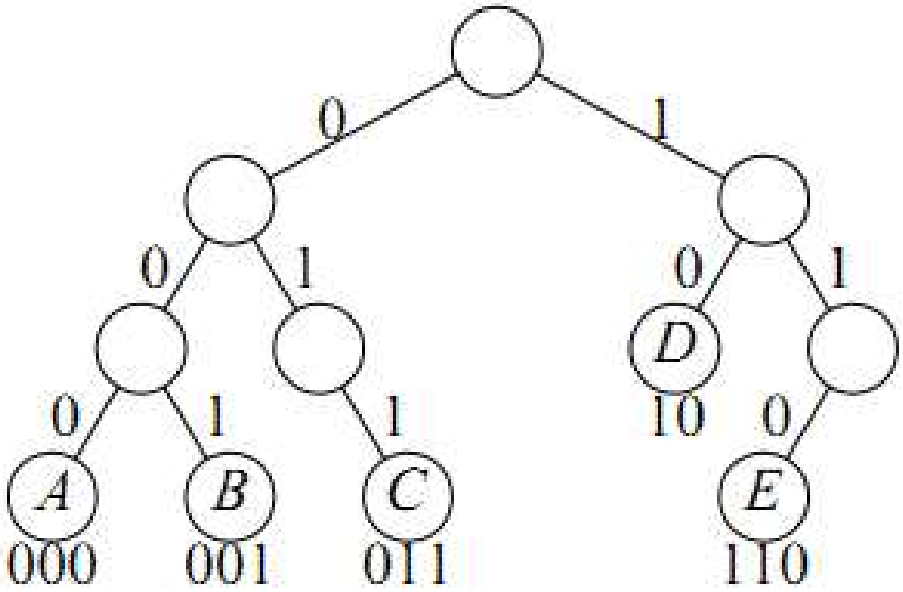
Problem is that A is prefix of D and A,E prefix of B
AE is 01 an B is 01

To Solve the prefix codes problem
We can use prefix free codes

**Prefix Free Code:**

**No Codes are prefix of other codes**
**Definition**
**Any prefix free code is equivalent to a binary tree every left child is 0 and every right child is 1**

**Example**



# Huffman Coding

There are mainly two major parts in Huffman Coding

**1)** Build a Huffman Tree from input characters.

**2)** Traverse the Huffman Tree and assign codes to characters.

***Steps to build Huffman Tree***

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

**1.** Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

**2.** Extract two nodes with the minimum frequency from the min heap.

**3.** Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

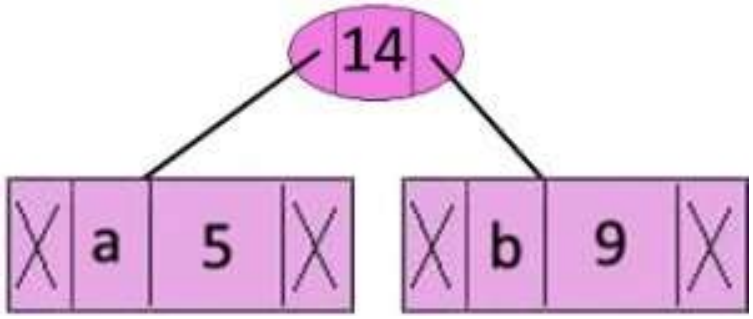
**4.** Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

| Character | Frequency |
|---|---|
| a | 5 |
| b | 9 |
| c | 12 |
| d | 13 |
| e | 16 |
| f | 45 |

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

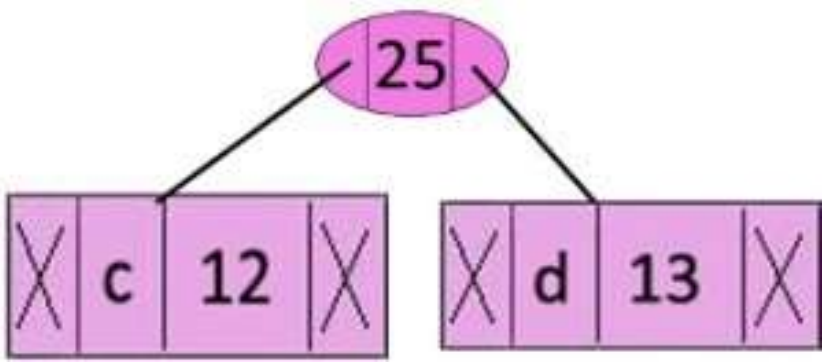



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

| **character** | **Frequency** |
|---|---|
| c | 12 |
| d | 13 |
| Internal Node | 14 |

| | |
|---|---|
| e | 16 |
| f | 45 |

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

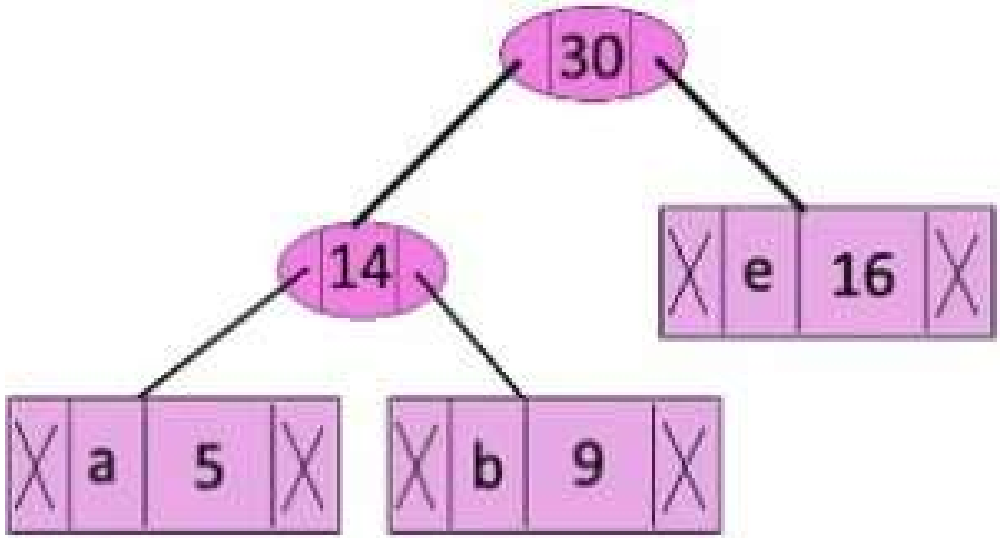



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

| **Character** | **Frequency** |
|---|---|
| Internal Node | 14 |
| e | 16 |
| Internal Node | 25 |

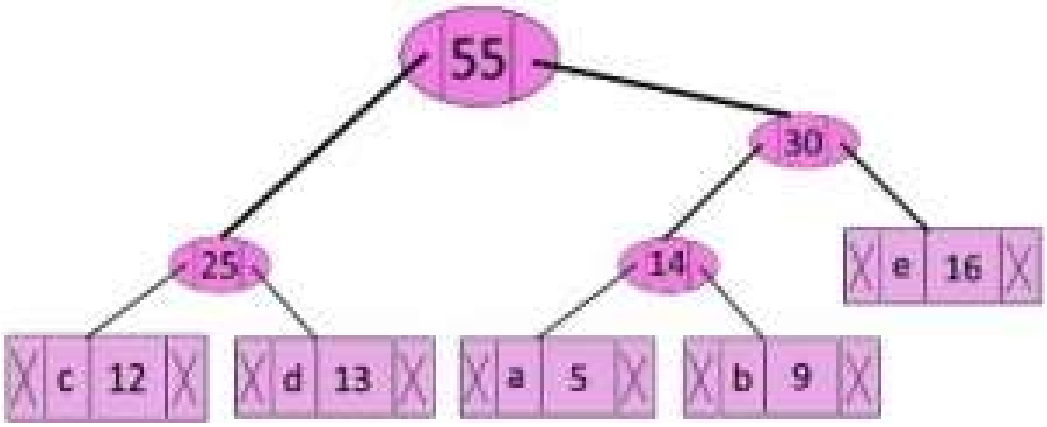
| | |
|---|---|
| f | 45 |

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30




Now min heap contains 3 nodes.

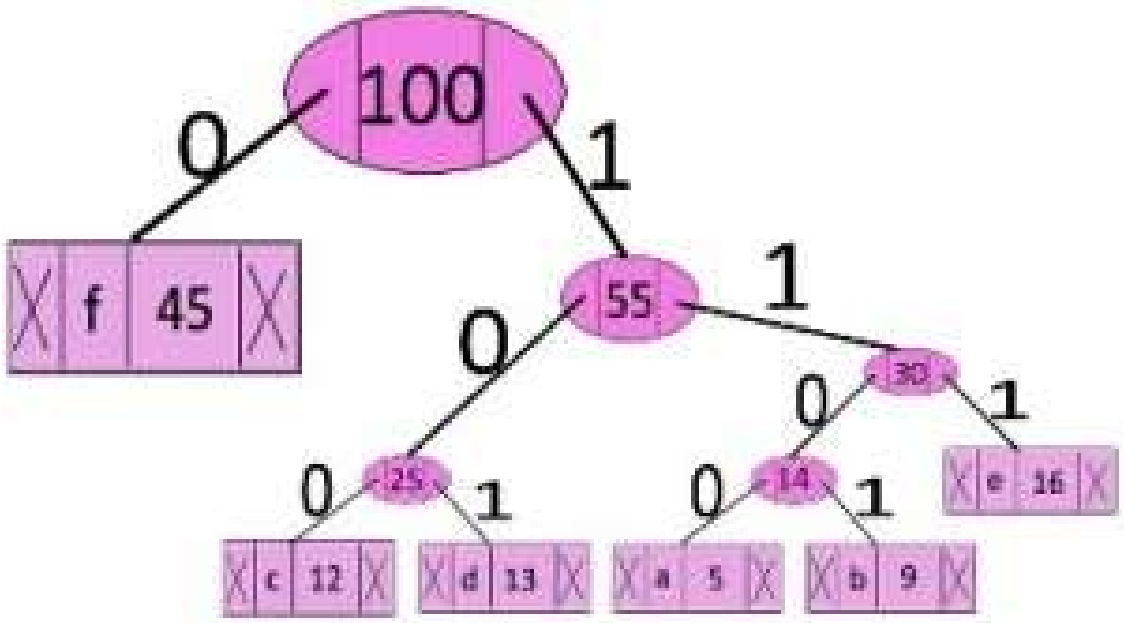
| **Character** | **Frequency** |
|---|---|
| Internal Node | 25 |
| Internal Node | 30 |
| f | 45 |



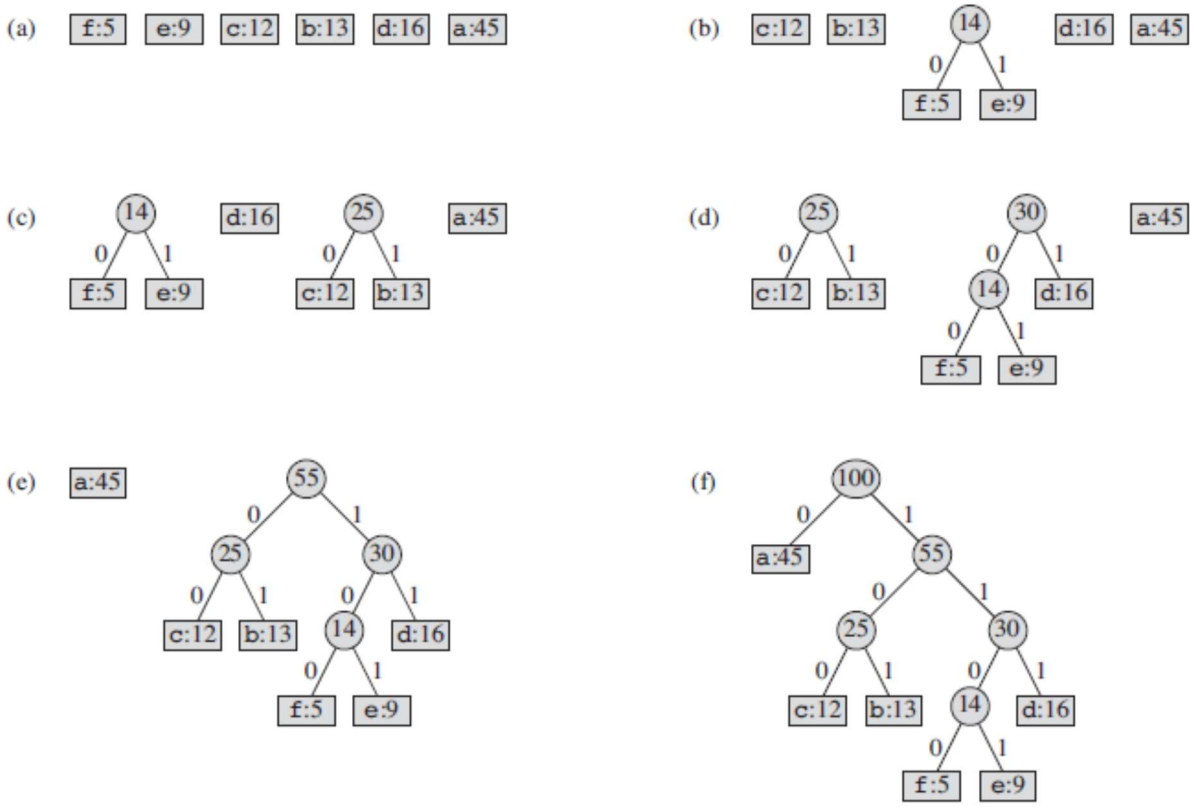

Now min heap contains 2 nodes.

| **Character** | **Frequency** |
|---|---|
| f | 45 |
| Internal Node | 55 |

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100




## All Steps done in previous example




## The codes are as follows:

| Character | code-word |
|---|---|
| f | 0 |
| c | 100 |
| d | 101 |
| a | 1100 |
| b | 1101 |
| e | 111 |

Huffman Code Algorithm

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

In the pseudocode that follows, we assume that C is a set of n characters and that each character c ϵ C is an object with an attribute c.*freq* giving its frequency.

The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of C leaves and performs a sequence of |C|-1 "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

## Analyzing Running Time of Huffman Algorithm

To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap

For a set C of n characters,
we can initialize Q in line 2 in O(n) time using the **BUILD-MIN-HEAP**

The **for** loop in lines 3–8 executes exactly (n-1) times, and since each heap operation requires time

O(lg n) the loop contributes  O(n lg n). to the running time.

Thus, the total running time of HUFFMAN on a set of n characters is O(n lg n).

**Why Huffman is considered as greedy Algorithm?**

The binary tree and Huffman's greedy algorithm look at the occurrence of each character and store it as a binary string in an optimal way.
he idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
We create a 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.