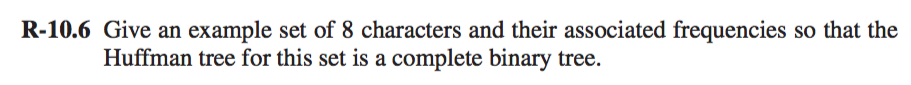
CS 600 Homework 6 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 10/12/2017

Chapter 10:



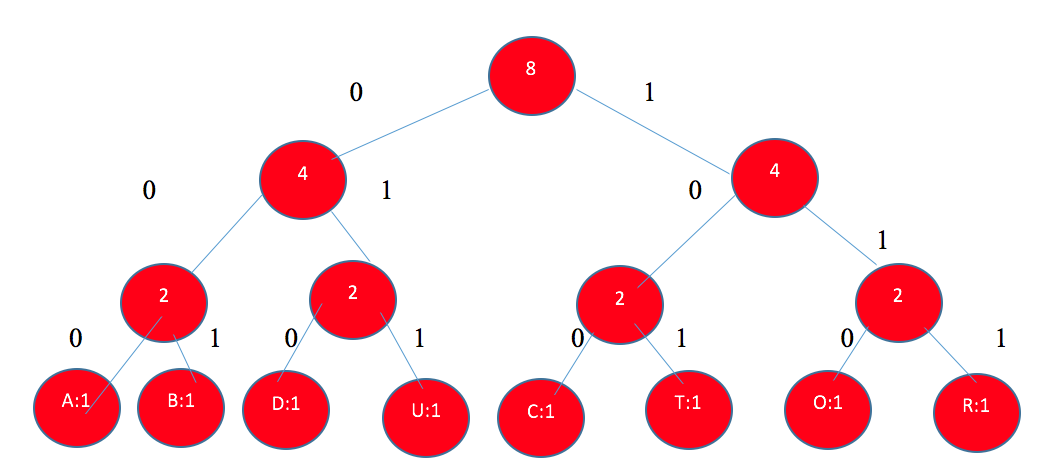
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible which indicates that it is a balanced tree.

To design a Huffman tree for a set of 8 characters such that it is a complete binary tree,

Consider the word “ABDUCTOR”.

|  |  |
| --- | --- |
| CHARACTER | FREQUENCY |
| A | 1 |
| B | 1 |
| D | 1 |
| U | 1 |
| C | 1 |
| T | 1 |
| O | 1 |
| R | 1 |

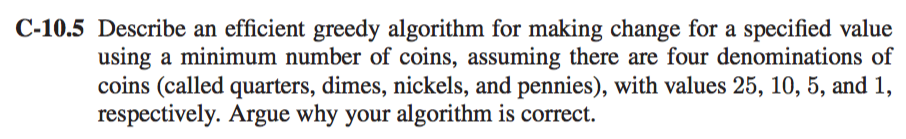
Huffman tree can be designed as:



The codes for the characters are:

|  |  |
| --- | --- |
| CHARACTERS | HUFFMAN CODE |
| A | 000 |
| B | 001 |
| D | 010 |
| U | 011 |
| C | 100 |
| T | 101 |
| O | 110 |
| R | 111 |

The Time Complexity is: O(n log n)



The greedy algorithm is an algorithmic paradigm that follows problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

The best way to solve the above problem is to use the Greedy Algorithm. This is because it will always make a greedy Choice on an item set such that the outcome would be either minimum or Maximum.

If we have to find the minimum number of coins to get a specified value. Now the best way to do this is to get the coins having the greatest value until it can’t take more of those coins and then go for smaller values.

In the Question suppose we have to make change of value N in minimum amount of coins (with values 25, 10, 5, and 1), then the best possible answer would be to first select the coins with value 25, Then value 10 and so on…

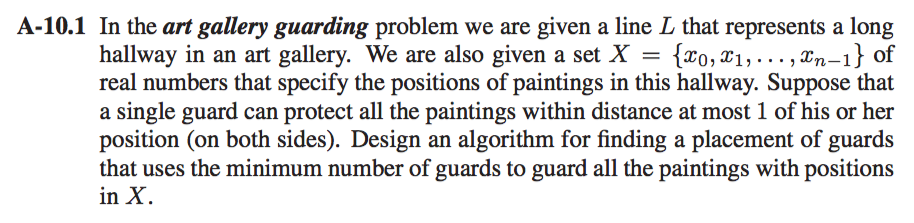
For Example,

Suppose N=31 Let’s compare in different order of coins taken for Change

|  |  |  |  |
| --- | --- | --- | --- |
| ORDER | COINS TAKEN | TOTAL | SOLUTION |
| 10,1,5,25 | 10 x 3  1 x 1 | 4 coins | **Not Optimal** |
| 1,5,10,25 | 1 x 31 | 31 coins | **Not Optimal** |
| 5,10,25,1 | 5 x 6  1 x 1 | 6 coins | **Not Optimal** |
| 25,10,5,1 | 25 x 1  5 x 1  1 x 1 | 3 coins | **Optimal** |

We can see that, by applying greedy algorithm we get a optimal Solution so we can take the quarters (value 25) until the resulting amount left is less than 25, after that we can take a dime (value 10) until the resulting amount is less than 10, after that we can take a nickel (value 5) until the resulting amount is less than 5 and then we take pennies.

If we follow this, we will get the optimal solution of which the Algorithm will take 𝑶(𝒏) 𝑡𝑖𝑚𝑒.   
& If we are sorting the coin value array then Total time would be 𝑶(𝒏 𝐥𝐨𝐠𝒏).



It is given that, all the art work is placed in a single line L in such a way that the position is a set 𝑋 {𝑥0, 𝑥1, . . . , 𝑥𝑛 − 1} where x0, x1 … represent the position of each painting.

To find the Minimum number of guards to guard the painting, we can use greedy algorithm, where the algorithm aims to place each guard within distance of atmost 1 on both sides.

To solve this let’s take the position of 1st Painting i.e. 𝑥0. Now, it is not given that the guard should be placed on the positon of painting. Therefore, we can place the guard in between the paintings.

At 1st, we don’t know whether the position of paintings is given in order. Hence, we sort the position using best sorting technique i.e. Quick Sort or Merge Sort.

Algorithm:

Input: A set of paintings P, such that X = {x0, x1, x2, …, xn-1} which specifies the position of the paintings.

Output: Minimum\_Guard = Minimum number of guards to guard the paintings at respective positions.

Consider Guard\_optimum as the optimum number of guards which would guard the paintings.

Let Guard\_ optimum = 0;

while (p != 0){

Remove painting Pi such that its distance is smallest, X0

if (Guard\_j does not have any conflict with painting Pi)

then schedule Pi for Gj

else

for the current painting P[i+2] add a new guard

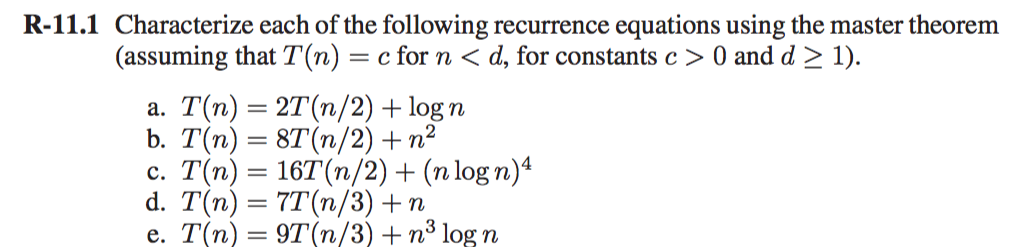
Guard\_ optimum = Guard\_ optimum + 2

Schedule painting Pi for guard Guard\_optimum

}

This can be done in 𝑶(𝒏) time. If sorting is required, then it would be 𝑶(𝒏 𝐥𝐨𝐠𝒏)

Chapter 11:



1] 𝑇(𝑛)=2𝑇(𝑛/2)+𝑙𝑜𝑔𝑛

Here, 𝑎=2 ,𝑏=2

𝑓(𝑛)=log𝑛

By Case 1,

𝑓(𝑛)= 𝑂(𝑛logba-ɛ)

𝑓(𝑛)= 𝑂(𝑛log22 - ɛ)

𝑓(𝑛)= 𝑂(𝑛1-ɛ)

Since 𝑤𝑒 𝑔𝑒𝑡 𝑠𝑎𝑚𝑒 log𝑛 𝑖𝑠 𝑂(𝑛1-ɛ), By Case 1 in Masters theorem,

𝑇(𝑛)= 𝜃(𝑛logba)

𝑻(𝒏)= 𝜽(𝒏) 𝒖𝒔𝒊𝒏𝒈 𝒎𝒂𝒔𝒕𝒆𝒓𝒔 𝒕𝒉𝒆𝒐𝒓𝒆𝒎

2] 𝑇(𝑛)=8𝑇(𝑛/2)+ 𝑛2

Here 𝑎=8,𝑏=2 and 𝑓(𝑛)=𝑛2

In this case, 𝑛logba=𝑛log28=𝑛3,

Thus, here 𝑓(𝑛) 𝑖𝑠 𝑂(𝑛3-ɛ) 𝑓o𝑟 ɛ=1

This means by Case 1 in theorem, 𝑻(𝒏) 𝒊𝒔 𝜽(𝒏3) 𝒖𝒔𝒊𝒏𝒈 𝒎𝒂𝒔𝒕𝒆𝒓𝒔 𝒕𝒉𝒆𝒐𝒓𝒆𝒎

3] 𝑇(𝑛)= 16𝑇(𝑛/2)+(𝑛𝑙𝑜𝑔𝑛)4

Here, 𝑎=16 ,𝑏=2 𝑎𝑛𝑑 𝑓(𝑛)=(𝑛𝑙𝑜𝑔𝑛)4

In this case,

𝑛logba=𝑛log216=𝑛4

Thus, here 𝑓(𝑛) 𝑖𝑠 𝜃(𝑛logba logkn) with k=1

This means by case 2 in Theorem,

𝑇(𝑛)=𝜃(nlogba logk+1 n)

𝑻(𝒏)=𝜽(𝒏4(𝒍𝒐𝒈𝒏)5)

4] 𝑇(𝑛)=7𝑇(𝑛/3)+ 𝑛

Here, 𝑎=7,𝑏=3 𝑎𝑛𝑑 𝑓(𝑛)=𝑛

In this case,

𝑛logba=𝑛log37=𝑛1.77

Thus, here by Case 1 of the Theorem, 𝑓(𝑛)=𝑛 𝑖𝑠 𝑂(𝑛1.77) where 𝜀=0.77

Hence, 𝑇(𝑛)=𝜃(𝑛lobba)

𝑻(𝒏)=𝜽(𝒏**1.77**)

5] 𝑇(𝑛)=9𝑇(𝑛/3)+ 𝑛3logn

Here, 𝑎=9,𝑏=3 𝑓(𝑛)=𝑛3logn

In this Case,

𝑛logba=𝑛log39=𝑛2

Thus, Here by case 3 of the theorem, 𝑓(𝑛)=𝑛3logn 𝑖𝑠 Ω(𝑛2) where 𝜀=1

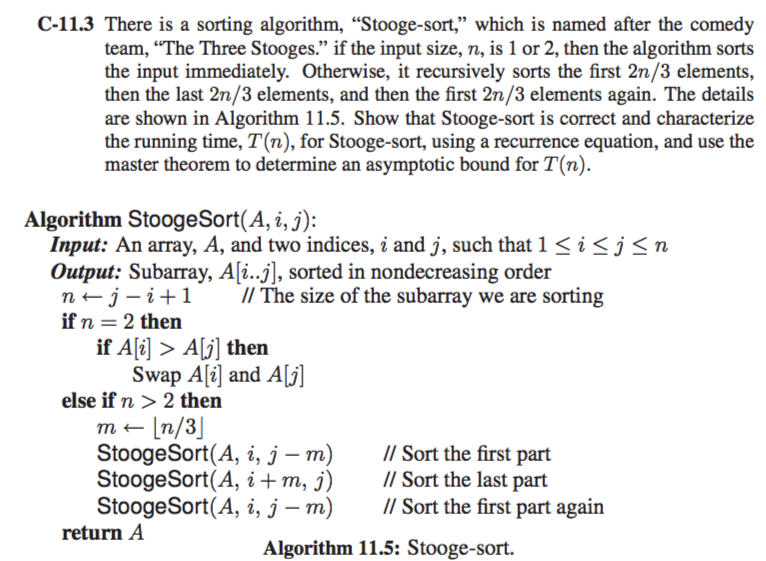
→𝑎 𝑓(𝑛/𝑏)≤ 𝛿 𝑓(𝑛)

𝐿𝐻𝑆:𝑎 𝑓(𝑛/𝑏)

→9 𝑓(𝑛/3) →9(𝑛/3)3logn/3 →𝑛3/3log n/3

So, 𝑛3/3 log n/3≤ 𝛿𝑛3 logn for 𝛿=1/3 𝑎𝑛𝑑 𝑛≥1

Hence, 𝑻(𝒏)𝒊𝒔 𝜽(**𝒏3logn**)



The algorithm works on the principle of Divide and Conquer.

Let’s consider the following example, {6,8,7,1,3,9}

As described in the algorithm, the stooge start will start by first sorting 2n/3 elements of the series. In this case n = 6. Therefore, the algorithm will start by sorting the first 4 numbers.

Stooge Sort applied on the first 4 numbers will divide the set {6,8,7,1} into groups of 3 and 1 respectively as follows: {6,8,7} and {1}

Working recursively, Stooge Sort proceeds on by further dividing the set as follows: {6,8} and {7}

Since we have two numbers in the set, the numbers get swapped if A[i] > A[i + 1]

Since the set of two is sorted, we start merging with other sets.

Hence, now, the algorithm sorts {6,8} and {7} => {6,7,8}

Then, {6,7,8} and {1}

Thus, the set of first 4 numbers is sorted: {1,6,7,8}

Now, it will sort the last for elements i.e {7,8,3,9}

Following the same process as described above, we get the following set of last 4 elements: {3,7,8,9}

While the 1st set was sorting all the large numbers were pushed towards end and when the 2nd set started sorting, the large numbers from 1st set are common in both the set. Now, in 2nd set after sorting the 1st 2 numbers are common with the 1st set. Therefore, we apply 𝑆𝑡𝑜𝑜𝑔𝑒𝑆𝑜𝑟𝑡 again on the 1st set to sort the elements completely.

So, in our Example, the Final set would be{1,6,3,7,8,9}

Here StoogeSort is applied on {1,6,3,7}

Thus, the final set will be {1,3,6,7,8,9}

**We can also solve the same problem by induction by considering the bases cases**

For the base case let n = 2. The first two lines of the algorithm will check if the two elements are sorted; if not, it exchanges them (and now they are sorted). The algorithm returns after the following if statement. Thus Stooge-Sort sorts correctly for n = 2.

Assume Stooge-Sort correctly sorts an input array A[1 to k], where k = length[A] and 1 <= k < n. In particular, Stooge-Sort correctly sorts an input array of size k = 2n/3.

Let A[1 to n] be an input array of size n = length[A].

By the induction hypothesis the first time we call Stooge-Sort(a, i, j - k) , it correctly sorts the first 2n/3 elements, so that the elements 1 to n/3 are less than elements (n + 1)/3 to 2n/3.

The call to Stooge-Sort(A, i, j - k) correctly sorts the last 2n/3 elements, so that the elements   
(n + 1)/3 to 2n/3 are less than elements 2(n + 1)/3 to n, which are the largest n/3 elements in A. The last call to Stooge-Sort(A, i, j - k) sorts correctly (by induction hypothesis) the sorted elements are less than elements 2(n+ 1)/3 … n. Thus the array A of size n

Time Complexity Using Masters Theorem,

𝑇(𝑛)=3 𝑇(2𝑛/3)+𝑂(𝑛)

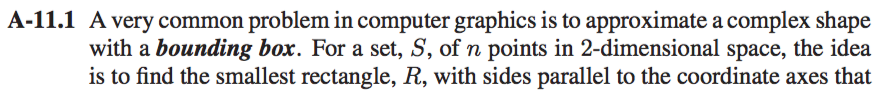
Here, 𝑎=3,𝑏=3/2 𝑓(𝑛)=𝑛

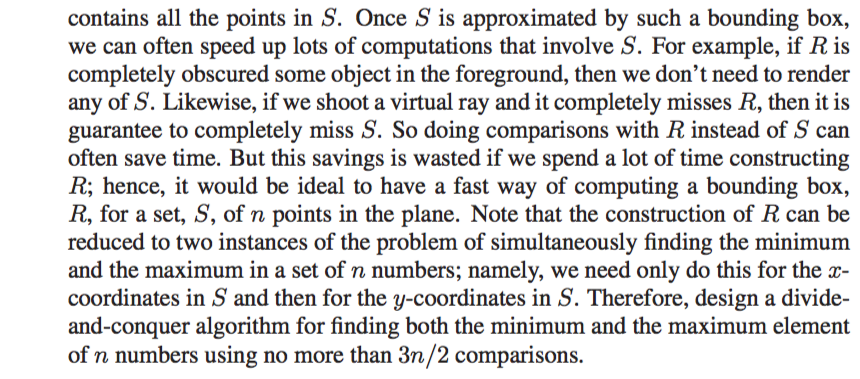
In this case,

nlogba = 𝑛log3/23 = 𝑛2.71

Thus, by using case 1, we can say that 𝑓(𝑛)=𝑛 𝑖𝑠 𝑂(𝑛2.71) for 𝜀=1.71

Hence, 𝑻(𝒏)=𝑶(𝒏**2.71**)





Suppose we have an array A containing all the 𝑛 numbers in it.

To find Maximum and Minimum from the array provided that there are no more than 𝑛 𝑐𝑜𝑚𝑝𝑎𝑟𝑖𝑠𝑜𝑛𝑠

This can be done using Divide and Conquer method:

**Algorithm** 𝑚𝑖𝑛𝑚𝑎𝑥(𝐴)

**Input**: An array A containing n numbers

**Output**: Minimum and Maximum number from array

Divide Array into 2 equal array (A1 𝑎𝑛𝑑 A2)

(𝑚𝑖𝑛1,𝑚𝑎𝑥1)=𝑚𝑖𝑛𝑚𝑎𝑥(A1)

(𝑚𝑖𝑛2,𝑚𝑎𝑥2)=𝑚𝑖𝑛𝑚𝑎𝑥(A2)

𝒊𝒇 (𝑚𝑖𝑛1>𝑚𝑖𝑛2) 𝒕𝒉𝒆𝒏

𝑟𝑒𝑡𝑢𝑟𝑛 𝑚𝑖𝑛=𝑚𝑖𝑛2

𝒆𝒍𝒔𝒆

𝑟𝑒𝑡𝑢𝑟𝑛 𝑚𝑖𝑛=𝑚𝑖𝑛1

𝒊𝒇 (𝑚𝑎𝑥1>𝑚𝑎𝑥2) 𝒕𝒉𝒆𝒏

𝑟𝑒𝑡𝑢𝑟𝑛 𝑚𝑎𝑥=𝑚𝑎𝑥1

𝒆𝒍𝒔𝒆

𝑟𝑒𝑡𝑢𝑟𝑛 𝑚𝑎𝑥=𝑚𝑎𝑥2

Time Complexity:

𝑇(𝑛)=2 𝑇(𝑛/2)+𝑂(1)

By Master’s Theorem,

𝑎=,𝑏=2 𝑎𝑛𝑑 𝑓(𝑛)=𝑂(1)

In this case:

𝑛logba=𝑛log22=𝑛

Here, 𝑓(𝑛) 𝑖𝑠 𝑂(𝑛),

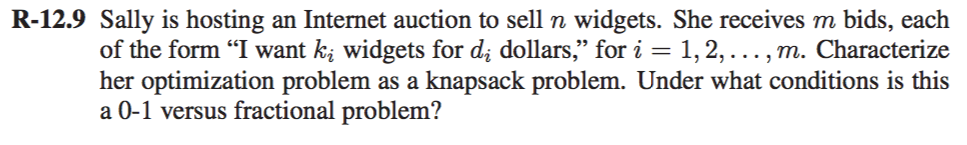
So,   
By CASE 1 in Master’s Theorem,

T(n)=𝑂(𝑛log22)=𝑂(𝑛)

So, 𝑇(𝑛)=𝑂(𝑛)

𝑇(𝑛) 𝑐𝑎𝑛 𝑏𝑒 𝑝𝑒𝑟𝑓𝑜𝑟𝑚𝑒𝑑 3𝑛/2 𝑐𝑜𝑚𝑝𝑎𝑟𝑖𝑠𝑜n𝑠,𝑆𝑖𝑛𝑐𝑒 𝑇(𝑛) 𝑖𝑠 𝑂(𝑛)

Chapter 12:



A positive benefit bf, and weight wi are the two characteristics of a knapsack problem. In this case,

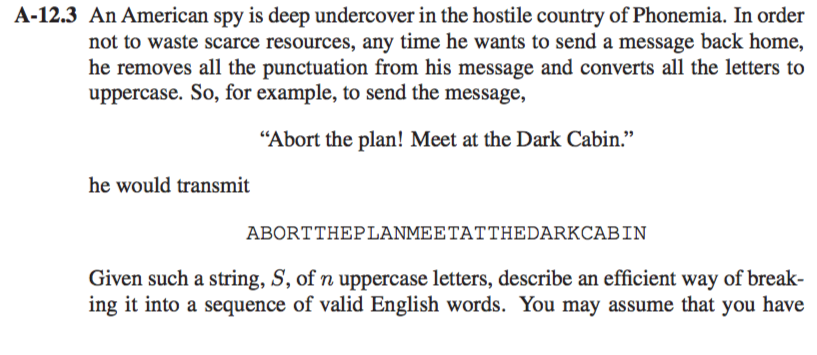
bf = benefit

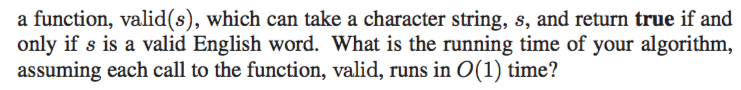
n = total weight of the items (n widgets)

ki = weight of each item

di = value of each item

In the 0-1 knapsack problem, no parts are allowed. In this case, if all the widgets are sold or none are sold then it would be considered as a 0-1 knapsack problem. If some of the widgets are sold it would amount to a fractional knapsack problem, since fractional knapsack allows fractional quantities.





Using Dynamic Programming & using memoization...

**Note**: It will also work for overlapping complex string.

Algorithm **StringBreaker**(S):

**Input**: A string S to be converted

**Output**: A string compromising of valid English words of S

//Let O be an empty list

for i ← 0 to n-1 do

if valid(S.substr(0,i)) then

O.add(S.substr(0,i))

O.add(StringBreaker(S.substr(I,n-1)))

break

return O

It normally takes O(n) time to check S.substr. So the Total Running time is O(n2)