Chapter 5
Greedy Algorithms

In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed "bottom-up". Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Greedy method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (non-optimal solution strategies), are often used in finding good approximations.

To prove that a greedy algorithm is optimal we must show the following two characteristics are exhibited.

- Greedy Choice Property
- Optimal Substructure Property

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. Any amount of item can be put into the bag i.e. x_i fraction of item can be collected, where $0<=x_i<=1$. Here the objective is to collect the items that maximize the total profit earned.

Algorithm

Take as much of the item with the highest value per weight (v_i/w_i) as you can. If the item is finished then move on to next item that has highest (v_i/w_i), continue this until the knapsack is full. v[1 … n] and w[1 … n] contain the values and weights respectively of the n objects sorted in non increasing ordered of v[i]/w[i] . W is the capacity of the knapsack, x[1 … n] is the solution vector that includes fractional amount of items and n is the number of items.

```
GreedyFracKnapsack(W,n)
{
        for(i=1; i<=n; i++)
```

```
                x[i] = 0.0;
                tw = W;
            for(i=1; i<=n; i++)
            {
                if(w[i] > tw)
                break;
                else
                x[i] = 1.0;

                tempw -= w[i];
            }
            if(i<=n)
            x[i] = tw/w[i];
        }
```

Analysis:

We can see that the above algorithm just contain a single loop i.e. no nested loops the running time for above algorithm is O(n). However our requirement is that v[1 … n] and w[1 … n] are sorted, so we can use sorting method to sort it in O(nlogn) time such that the complexity of the algorithm above including sorting becomes O(nlogn).

Job Sequencing with Deadline

We are given a set of n jobs. Associated with each job I, di≥0 is an integer deadline and pi≥0 is profit. For any job i profit is earned iff job is completed by deadline. To complete a job one has to process a job for one unit of time. Our aim is to find feasible subset of jobs such that profit is maximum.

Example

n=4, (p_1,p_2,p_3,p_4)=(100,10,15,27), (d_1,d_2,d_3,d_4)=(2,1,2,1)

n=4, (p1,p4,p3,p2)=(100,27,15,10), (d1,d4,d3,d2)=(2,1,2,1)

	Feasible Solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

We have to try all the possibilities, complexity is O(n!).

Greedy strategy using *total profit* as optimization function to above example.

Begin with J=ϕ

- Job 1 considered, and added to J → J={1}
- Job 4 considered, and added to J → J={1,4}
- Job 3 considered, but discarded because not feasible → J={1,4}
- Job 2 considered, but discarded because not feasible → J={1,4}

Final solution is J={1,4} with total profit 127 and it is optimal

Algorithm

Assume the jobs are ordered such that $p[1]\geq p[2]\geq\ldots\geq p[n]$

d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs n are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i] is the ith job in the optimal solution, 1<=i<=k.

```
JobSequencing(int d[], int j[], int n)
 {
        for(i=1;i<=n;i++)
        {//initially no jobs are selected
                J[i]=0;
        }
         for (int i=1; i<=n; i++)
        {
                d=d[i];
                for(k=d;k>=0;k--)
                {
                        if(j[k]==0)
                        {
                                J[k]=i;
                        }
                 }
        }
}
```

Analysis

First for loop executes for O(n) times .

In case of second loop outer for loop executes O(n) times and inner for loop executes for at most O(n) times in the worst case. All other statements takes O(1) time. Hence total time for each iteration of outer for loop is O(n) in worst case.

Thus time complexity= $O(n) + O(n^2) = O(n^2)$.

Huffman Coding

Huffman coding is an algorithm for the lossless compression of files based on the frequency of occurrence of a symbol in the file that is being compressed. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters, i.e., 0 represents the first character and 1 represents the second character. Using two bits we can represent four characters, and so on. Unlike ASCII code, which is a fixed-length code using seven bits per character, Huffman compression is a variable-length coding system that assigns smaller codes for more frequently used characters and larger codes for less frequently used characters in order to reduce the size of files being compressed and transferred.

For example, in a file with the following data: ‘XXXXXXYYYYZZ’. The frequency of "X" is 6, the frequency of "Y" is 4, and the frequency of "Z" is 2. If each character is represented using a fixed-length code of two bits, then the number of bits required to store this file would be 24, i.e., (2 x 6) + (2x 4) + (2x 2) = 24. If the above data were compressed using Huffman compression, the more frequently occurring numbers would be represented by smaller bits, such as: X by the code 0 (1 bit),Y by the code 10 (2 bits) and Z by the code 11 (2 bits), the size of the file becomes 18, i.e., (1x 6) + (2 x 4) + (2 x 2) = 18. In this example, more frequently occurring characters are assigned smaller codes, resulting in a smaller number of bits in the final compressed file. Huffman compression was named after its discoverer, David Huffman.

To generate Huffman codes we should create a binary tree of nodes. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and $n-1$ internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths. The process essentially begins with the leaf nodes containing the probabilities of the symbol they represent, then a new node whose children are the 2 nodes with smallest probability is created, such that the new node's probability is equal to the sum of the children's probability. With the previous 2 nodes merged into one node and with the new node being now considered, the procedure is repeated until only one node remains, the Huffman tree. The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

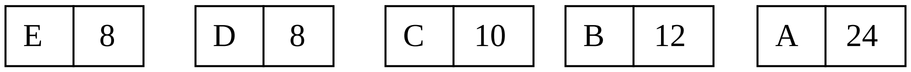
Example

The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

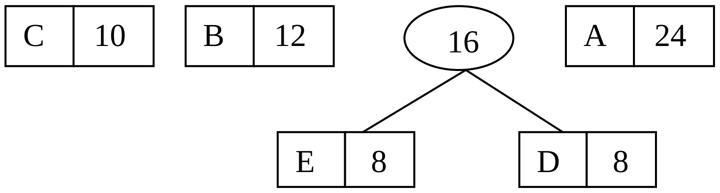
Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

----> total 186 bit (with 3 bit per code word)

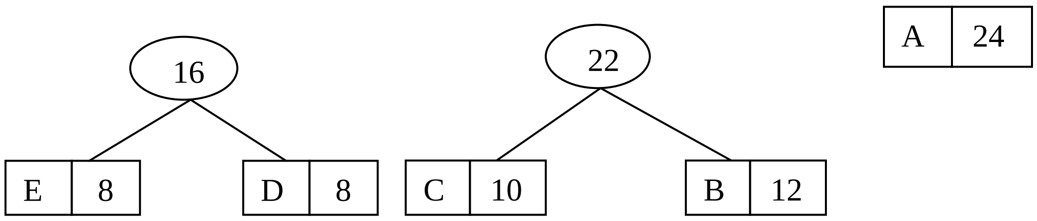
Step1:



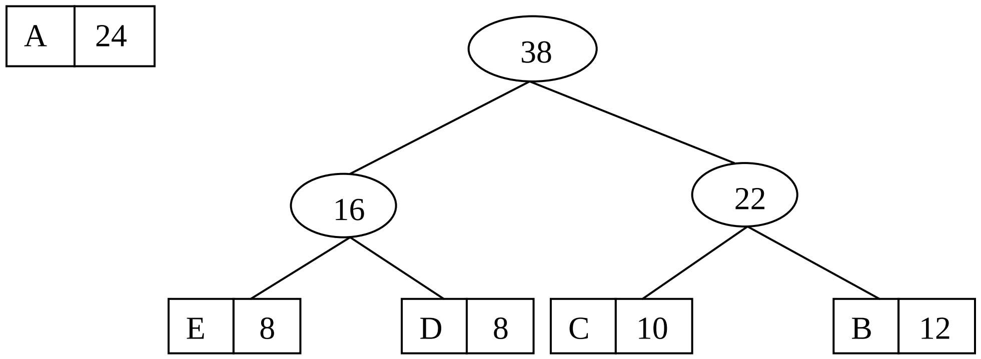
Step2:



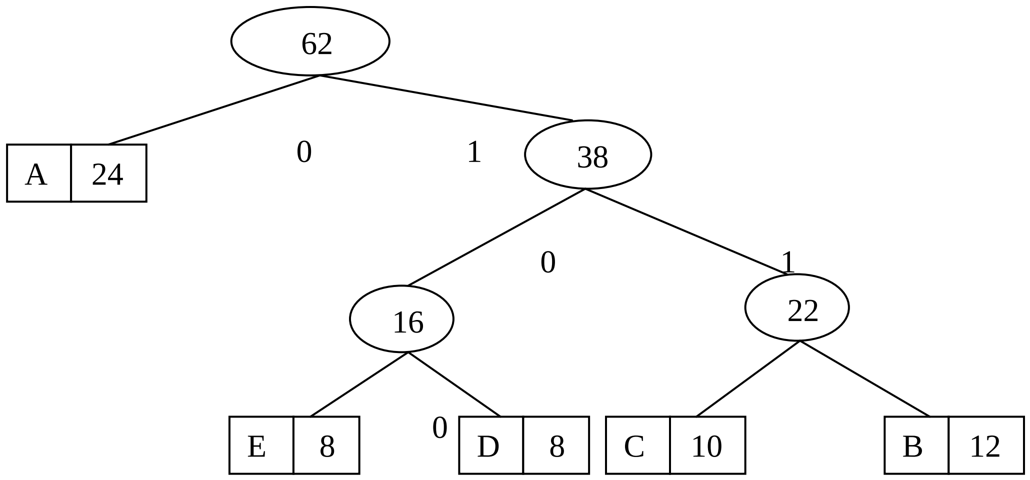
Step3:



Step4:



Step5:



Symbol	Frequency	Code	Code length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

Total length of message: 138 bit

Algorithm

A greedy algorithm can construct Huffman code that is optimal prefix codes. A tree corresponding to optimal codes is constructed in a bottom up manner starting from the |C| leaves and |C|-1 merging operations. Use priority queue Q to keep nodes ordered by frequency. Here the priority queue we considered is binary heap.

```
HuffmanAlgo(C)
{
        n = |C|;   Q = C;
        For(i=1; i<=n-1; i++)
        {
                z = Allocate-Node();
                x = Extract-Min(Q);
                y = Extract-Min(Q);
                left(z) = x;   right(z) = y;
                f(z) = f(x) + f(y);
                Insert(Q,z);
        }
}
```

Analysis

We can use BuildHeap(C) to create a priority queue that takes O(n) time. Inside the for loop the expensive operations can be done in O(logn) time. Since operations inside for loop executes for n-1 time total running time of Huffman algorithm is O(nlogn).