# Greedy Algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See [this](http://www.cs.binghamton.edu/~dima/cs333/knapsack.ppt)) can be solved using Greedy, but [0-1 Knapsack](https://www.geeksforgeeks.org/archives/18430) cannot be solved using Greedy

Following are some standard algorithms that are Greedy algorithms.  
**1) [Kruskal’s Minimum Spanning Tree (MST)](http://en.wikipedia.org/wiki/Kruskal%27s_algorithm):** In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.  
**2)**[**Prim’s Minimum Spanning Tree**](http://en.wikipedia.org/wiki/Prim%27s_algorithm)**:** In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.  
**3) [Dijkstra’s Shortest Path](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm):**The Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.  
**4)**[**Huffman Coding**](http://en.wikipedia.org/wiki/Huffman_coding)**:** Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

**Standard Greedy Algorithms :**

1. [Activity Selection Problem](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)
2. [Egyptian Fraction](https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/)
3. [Job Sequencing Problem](https://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/)
4. [Huffman Coding](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)
5. [Efficient Huffman Coding for sorted input](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding-set-2/)
6. [Activity Selection Problem](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example 1 : Consider the following 3 activities sorted by

by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most two activities. The

maximum set of activities that can be executed

is {0, 2} [ These are indexes in start[] and

finish[] ]

Example 2 : Consider the following 6 activities

sorted by by finish time.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

A person can perform at most four activities. The

maximum set of activities that can be executed

is {0, 1, 3, 4} [ These are indexes in start[] and

finish[] ]

# Greedy Algorithm for Egyptian Fraction

Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3 is a unit fraction. Such a representation is called Egyptial Fraction as it was used by ancient Egyptians.

Following are few examples:

Egyptian Fraction Representation of 2/3 is 1/2 + 1/6

Egyptian Fraction Representation of 6/14 is 1/3 + 1/11 + 1/231

Egyptian Fraction Representation of 12/13 is 1/2 + 1/3 + 1/12 + 1/156

We can generate Egyptian Fractions using [Greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/). For a given number of the form ‘nr/dr’ where dr > nr, first find the greatest possible unit fraction, then recur for the remaining part. For example, consider 6/14, we first find ceiling of 14/6, i.e., 3. So the first unit fraction becomes 1/3, then recur for (6/14 – 1/3) i.e., 4/42.

Job Sequencing Problem | Set 1 (Greedy Algorithm)

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:

Input: Four Jobs with following deadlines and profits

JobID Deadline Profit

a 4 20

b 1 10

c 1 40

d 1 30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID Deadline Profit

a 2 100

b 1 19

c 2 27

d 1 25

e 3 15

Output: Following is maximum profit sequence of jobs

c, a, e

# Greedy Algorithms | Set 3 (Huffman Coding)

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.  
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

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

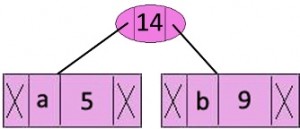
c 12

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.  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-1.jpeg)  
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

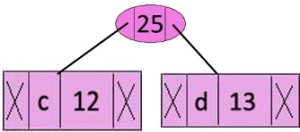
c 12

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  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-2.jpg)  
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.

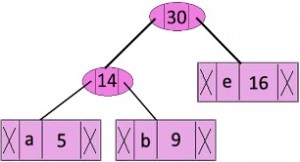
character Frequency

Internal Node 14

e 16

Internal Node 25

f 45

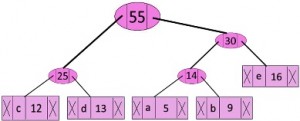
**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-3.jpg)  
Now min heap contains 3 nodes.

character Frequency

Internal Node 25

Internal Node 30

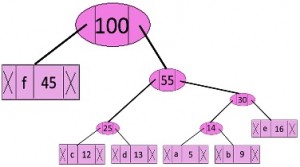
f 45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-4.jpg)  
Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

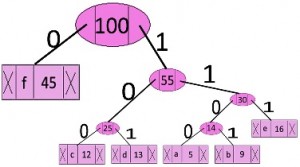
**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-5.jpg)  
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

**Steps to print codes from Huffman Tree:**  
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

[](http://www.geeksforgeeks.org/wp-content/uploads/fig-6.jpg)  
The codes are as follows:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

### A Simple Coding Example

We'll look at how the string "go go gophers" is encoded in ASCII, how we might save bits using a simpler coding scheme, and how Huffman coding is used to compress the data resulting in still more savings.

With an ASCII encoding (8 bits per character) the 13 character string "go go gophers" requires 104 bits. The table below on the left shows how the coding works.

|  |  |  |
| --- | --- | --- |
| **coding a message**  **ASCII coding** |  | **3-bit coding** |
| |  |  |  | | --- | --- | --- | | **char** | **ASCII** | **binary** | | g | 103 | 1100111 | | o | 111 | 1101111 | | p | 112 | 1110000 | | h | 104 | 1101000 | | e | 101 | 1100101 | | r | 114 | 1110010 | | s | 115 | 1110011 | | space | 32 | 1000000 | |  | |  |  |  | | --- | --- | --- | | **char** | **code** | **binary** | | g | 0 | 000 | | o | 1 | 001 | | p | 2 | 010 | | h | 3 | 011 | | e | 4 | 100 | | r | 5 | 101 | | s | 6 | 110 | | space | 7 | 111 | |

The string "go go gophers" would be written (coded numerically) as 103 111 32 103 111 32 103 111 112 104 101 114 115. Although not easily readable by humans, this would be written as the following stream of bits (the spaces would not be written, just the 0's and 1's)

1100111 1101111 1100000 1100111 1101111 1000000 1100111 1101111 1110000 1101000 1100101 1110010 1110011

Since there are only eight different characters in "go go gophers", it's possible to use only 3 bits to encode the different characters. We might, for example, use the encoding in the table on the right above, though other 3-bit encodings are possible.

Now the string "go go gophers" would be encoded as 0 1 7 0 1 7 0 1 2 3 4 5 6 or, as bits:

000 001 111 000 001 111 000 001 010 011 100 101 110 111

By using three bits per character, the string "go go gophers" uses a total of 39 bits instead of 104 bits. More bits can be saved if we use fewer than three bits to encode characters like g, o, and space that occur frequently and more than three bits to encode characters like e, p, h, r, and s that occur less frequently in "go go gophers". This is the basic idea behind Huffman coding: to use fewer bits for more frequently occurring characters. We'll see how this is done using a tree that stores characters at the leaves, and whose root-to-leaf paths provide the bit sequence used to encode the characters.