**Greedy Algorithms**

In an algorithm design there is no one 'silver bullet' that is a cure for all computation problems. Different problems require the use of different kinds of techniques. A good programmer uses all these techniques based on the type of problem. Some commonly-used techniques are:

1. Divide and conquer
2. Randomized algorithms
3. Greedy algorithms (This is not an algorithm, it is a **technique**.)
4. Dynamic programming

**What is a 'Greedy algorithm'?**

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Greedy algorithms have some advantages and disadvantages:

1. It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
2. **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

**Note:** Most greedy algorithms are **not** correct. An example is described later in this article.

**C. How to create a Greedy Algorithm?**

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum such things.

You are given an array **A** of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.

This is a simple Greedy-algorithm problem. In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables **currentTime** and **numberOfThings**. To complete the calculation, you must:

1. Sort the array **A** in a non-decreasing order.
2. Select each to-do item one-by-one.
3. Add the time that it will take to complete that to-do item into **currentTime**.
4. Add one to **numberOfThings**.

Repeat this as long as the **currentTime** is less than or equal to **T**.

Let **A = {5, 3, 4, 2, 1}** and **T = 6**

After sorting, **A = {1, 2, 3, 4, 5}**

After the 1st iteration:

* **currentTime** = 1
* **numberOfThings** = 1

After the 2nd iteration:

* **currentTime** is 1 + 2 = 3
* **numberOfThings** = 2

After the 3rd iteration:

* **currentTime** is 3 + 3 = 6
* **numberOfThings** = 3

After the 4th iteration, **currentTime** is 6 + 4 = 10, which is greater than T. Therefore, the answer is 3.

Therefore, the final algorithm that returns the optimal value of the objective function is:

Algorithm (P, T, N)

{

let S be an array of pairs ( C++ STL pair ) to store the scores

, C be the completion times and F be the objective function

for i from 1 to N:

S[i] = ( P[i] / T[i], i ) // Algorithm #2

sort(S)

C = 0

F = 0

for i from 1 to N: // Greedily choose the best choice

C = C + T[S[i].second]

F = F + P[S[i].second]\*C

return F

}

***Time complexity*** You have 2 loops taking O(N) time each and one sorting function taking O(N \* logN). Therefore, the overall time complexity is O(2 \* N + N \* logN) = **O(N \* logN)**.

**Proof of Correctness**

To prove that **algorithm #2** is correct, use **proof by contradiction**. Assume that what you are trying to prove is false and from that derive something that is obviously false.

Therefore, assume that this greedy algorithm does not output an optimal solution and there is another solution (not output by greedy algorithm) that is better than greedy algorithm.

**A = Greedy schedule (which is not an optimal schedule)**  
**B = Optimal Schedule (best schedule that you can make)**

**Assumption #1:** all the ( P[i] / T[i] ) are **different**.  
**Assumption #2:** (just for simplicity, will not affect the generality) **( P[1] / T[1] ) > ( P[2] / T[2] ) > .... > ( P[N] / T[N] )**

Because of assumption #2, the greedy schedule will be **A = ( 1, 2, 3, ....., N )**. Since A is not optimal (as we considered above) and A is not equal to B (because B is optimal), you can claim that **B must contain two consecutive jobs ( i, j ) such that the earlier of those 2 consecutive jobs has a larger index ( i > j ).** This is true because the only schedule that has the property, in which the indices only go up, is A = ( 1, 2, 3, ...., N ).

Therefore, **B = ( 1, 2, ..., i, j, ... , N ) where i > j**.

You also have to think about what is the profit or loss impact if you swap these 2 jobs. Think about the effect of this swap on the completion times of the following:

1. Work on k other than i and j
2. Work on i
3. Work on j

For k, there will be 2 cases:

**When k is on the left of i and j in B** If you swap i and j, then there will be no effect on the completion time of k.

**When k is on the right of i and j in B** After swapping, the completion time of k is C(k) = T[1] + T[2] + .. + T[j] + T[i] + .. T[k], k will remain same.

For i the completion time: Before swapping was C(i) = T[1] + T[2] + ... + T[i] After swapping is C(i) = T[1] + T[2] + ... + T[j] + T[i]

Clearly, the completion time for i goes up by T[j] and the completion time for j goes down by T[i].

**Loss due to the swap is (P[i] \* T[j])**  
**Profit due to the swap is (P[j] \* T[i])**

Using assumption #2, **i > j implies that ( P[i] / T[i] ) < ( P[j] / T[j] )**. Therefore **( P[i] \* T[j] ) < ( P[j] \* T[i] ) which means Loss < Profit**. This means that **swap improves B but it is a contradiction** as we assumed that B is the optimal schedule. This completes our proof.

**Where to use Greedy algorithms?**

A problem must comprise these two components for a greedy algorithm to work:

1. It has **optimal substructures**. The optimal solution for the problem contains optimal solutions to the sub-problems.
2. It has a **greedy property** (hard to prove its correctness!). If you make a choice that seems the best at the moment and solve the remaining sub-problems later, you still reach an optimal solution. You will never have to reconsider your earlier choices

# Greedy Algorithms | Set 1 (Activity Selection Problem)

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1) Sort the activities according to their finishing time  
2) Select the first activity from the sorted array and print it.  
3) Do following for remaining activities in the sorted array.  
…….a) If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

**How does Greedy Choice work for Activities sorted according to finish time?**  
Let the give set of activities be S = {1, 2, 3, ..n} and activities be sorted by finish time. The greedy choice is to always pick activity 1. How come the activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with first activity other than 1, then there is also a solution A of same size with activity 1 as first activity. Let the first activity selected by B be k, then there always exist A = {B – {k}} U {1}.(Note that the activities in B are independent and k has smallest finishing time among all. Since k is not 1, finish(k) >= finish(1)).

**How to implement when given activities are not sorted?**  
We create a structure/class for activities. We sort all activities by finish time (Refer [sort in C++ STL](https://www.geeksforgeeks.org/sort-c-stl/)). Once we have activities sorted, we apply same above algorithm.

|  |
| --- |
| // C++ program for activity selection problem  // when input activities may not be sorted.  #include <bits/stdc++.h>  using namespace std;    // A job has start time, finish time and profit.  struct Activitiy  {      int start, finish;  };    // A utility function that is used for sorting  // activities according to finish time  bool activityCompare(Activitiy s1, Activitiy s2)  {      return (s1.finish < s2.finish);  }    // Returns count of maximum set of activities that can  // be done by a single person, one at a time.  void printMaxActivities(Activitiy arr[], int n)  {      // Sort jobs according to finish time      sort(arr, arr+n, activityCompare);        cout << "Following activities are selected n";        // The first activity always gets selected      int i = 0;      cout << "(" << arr[i].start << ", " << arr[i].finish << "), ";        // Consider rest of the activities      for (int j = 1; j < n; j++)      {        // If this activity has start time greater than or        // equal to the finish time of previously selected        // activity, then select it        if (arr[j].start >= arr[i].finish)        {            cout << "(" << arr[j].start << ", "                << arr[j].finish << "), ";            i = j;        }      }  }    // Driver program  int main()  {      Activitiy arr[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6},                                         {5, 7}, {8, 9}};      int n = sizeof(arr)/sizeof(arr[0]);      printMaxActivities(arr, n);      return 0;  } |

Output:

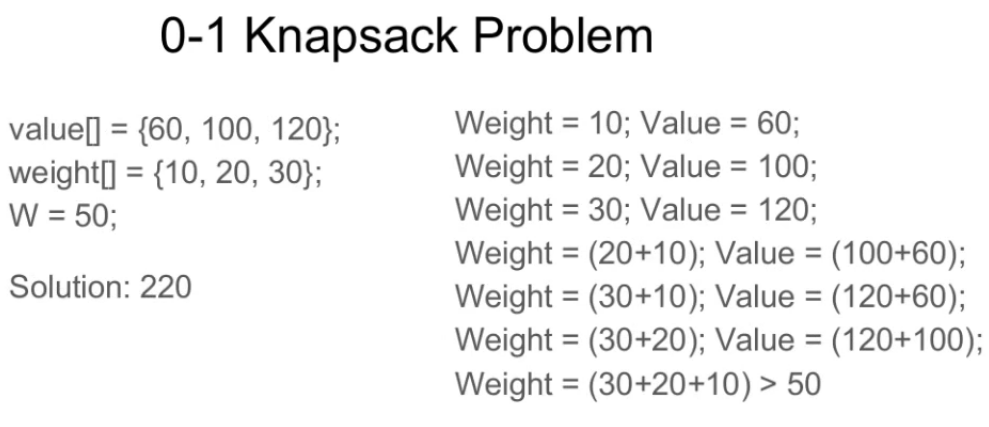
Following activities are selected

(1, 2), (3, 4), (5, 7), (8, 9),

**Time Complexity :** It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

**Dynamic Programming | Set 10 ( 0-1 Knapsack Problem)**

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).



A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

**1) Optimal Substructure:**  
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from n items is max of following two values.  
1) Maximum value obtained by n-1 items and W weight (excluding nth item).  
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

|  |
| --- |
| // A Dynamic Programming based solution for 0-1 Knapsack problem  #include<stdio.h>    // A utility function that returns maximum of two integers  int max(int a, int b) { return (a > b)? a : b; }    // Returns the maximum value that can be put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {     int i, w;     int K[n+1][W+1];       // Build table K[][] in bottom up manner     for (i = 0; i <= n; i++)     {         for (w = 0; w <= W; w++)         {             if (i==0 || w==0)                 K[i][w] = 0;             else if (wt[i-1] <= w)                   K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);             else                   K[i][w] = K[i-1][w];         }     }       return K[n][W];  }    int main()  {      int val[] = {60, 100, 120};      int wt[] = {10, 20, 30};      int  W = 50;      int n = sizeof(val)/sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

Output:

220

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

**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

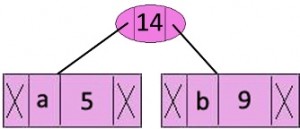
d 13

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

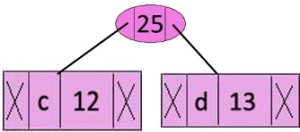
character Frequency

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.

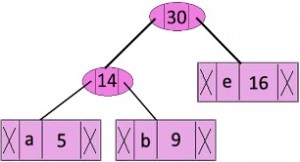
character Frequency

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)

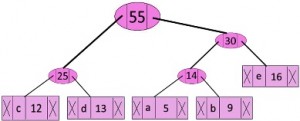
Now min heap contains 3 nodes.

character Frequency

Internal Node 25

Internal Node 30

f 45

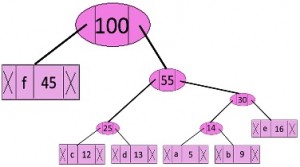
**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55  
[](http://www.geeksforgeeks.org/wp-content/uploads/fig-4.jpg)

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)

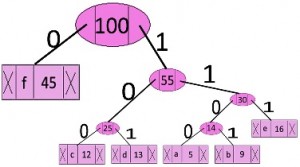
Now min heap contains only one node.

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

// C++ program for Huffman Coding

#include <bits/stdc++.h>

using namespace std;

// A Huffman tree node

struct MinHeapNode {

    // One of the input characters

    char data;

    // Frequency of the character

    unsigned freq;

    // Left and right child

    MinHeapNode \*left, \*right;

    MinHeapNode(char data, unsigned freq)

    {

        left = right = NULL;

        this->data = data;

        this->freq = freq;

    }

};

// For comparison of

// two heap nodes (needed in min heap)

struct compare {

    bool operator()(MinHeapNode\* l, MinHeapNode\* r)

    {

        return (l->freq > r->freq);

    }

};

// Prints huffman codes from

// the root of Huffman Tree.

void printCodes(struct MinHeapNode\* root, string str)

{

    if (!root)

        return;

    if (root->data != '$')

        cout << root->data << ": " << str << "\n";

    printCodes(root->left, str + "0");

    printCodes(root->right, str + "1");

}

// The main function that builds a Huffman Tree and

// print codes by traversing the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

    struct MinHeapNode \*left, \*right, \*top;

    // Create a min heap & inserts all characters of data[]

    priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare> minHeap;

    for (int i = 0; i < size; ++i)

        minHeap.push(new MinHeapNode(data[i], freq[i]));

    // Iterate while size of heap doesn't become 1

    while (minHeap.size() != 1) {

        // Extract the two minimum

        // freq items from min heap

        left = minHeap.top();

        minHeap.pop();

        right = minHeap.top();

        minHeap.pop();

        // Create a new internal node with

        // frequency equal to the sum of the

        // two nodes frequencies. Make the

        // two extracted node as left and right children

        // of this new node. Add this node

        // to the min heap '$' is a special value

        // for internal nodes, not used

        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;

        top->right = right;

        minHeap.push(top);

    }

    // Print Huffman codes using

    // the Huffman tree built above

    printCodes(minHeap.top(), "");

}

// Driver program to test above functions

int main()

{

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;

}

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2\*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).

If the input array is sorted, there exists a linear time algorithm.