☰

← Notes

## ▲ Greedy Algorithm

23    Greedy      CodeMonk

### A. Introduction

It's unfortunate that in algorithm design there is no one "silver bullet" that is the cure for all computation problems. Different problems require the use of different kind of techniques, and a good programmer makes use of all these techniques depending on the type of the problem. Some commonly used techniques are:

1. Divide and Conquer
2. Randomized Algorithms
3. Greedy Algorithms (It's not an algorithm, it's a **technique** )
4. Dynamic Programming

### B. What is Greedy Algorithm?

There is no formal definition of what exactly a greedy algorithm is. So let's consider an informal definition of what greedy algorithm usually looks like. A greedy algorithm, as the name suggests, always makes the choice that looks best at that moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Now the question is how do we decide which choice is the best (or optimal), for that we need to define a function (which we call an **objective function**) that needs to be optimized (either maximized or minimized) at the given point. Greedy algorithm makes greedy choices at each step such that the objective function is optimized. The Greedy Algorithm has only one shot to compute the optimal solution as **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 Divide and Conquer, it is quite unclear whether they are fast or slow because at each level of recursion, on one hand the size of problem is getting smaller and on the other hand the number of subproblems is increasing.
3. The difficult part is that **we have to work much harder to understand correctness issues** for greedy algorithms. Even with the correct algorithm it's hard to prove

why it is correct. Proving that a greedy algorithm is correct is more of an art than science. It takes a lot of creativity.

**NOTE:** Most greedy algorithms are **NOT** correct ( Example of this is given later in this article )

### C. How to come up with a Greedy Algorithm?

Let's take an example. Suppose you are suffering from a lethal disease and you have only **T years** left to live. Now you want to try all the different things you always wanted to do. You are given an array **A** of integers denoting the time taken to complete different things in years. You want to know what is the maximum number of different things you can do in the limited time you have.

As we can clearly see that this is a simple Greedy Problem. In each iteration, we have to greedily select the things which will take minimum time to complete and also maintain two variables **currentTime** and **numberOfThings**. So, simply sort the array **A** in nondecreasing order. Then, select the things one by one and add the time taken to complete that thing into **currentTime** and also add one to **numberOfThings**. Repeat as long as the **currentTime** is smaller than or equal to T.

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

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

After 1$^{st}$ iteration, **currentTime** = 1 and **numberOfThings** = 1
After 2$^{nd}$ iteration, **currentTime** = 1 + 2 = 3 and **numberOfThings** = 2
After 3$^{rd}$ iteration, **currentTime** = 3 + 3 = 6 and **numberOfThings** = 3
After 4$^{th}$ iteration, **currentTime** = 6 + 4 = 10 which is greater than T, answer will be 3.

**Implementation:**

```cpp
#include <iostream>
#include <algorithm>

using namespace std;
const int MAX = 105;
int A[MAX];

int main()
{
    int T, N, numberOfThings = 0, currentTime = 0;
```

```
    cin >> N >> T;
    for(int i = 0;i < N;++i)
        cin >> A[i];
    sort(A, A + N);
    for(int i = 0;i < N;++i)
    {
        currentTime += A[i];
        if(currentTime > T)
            break;
        numberOfThings++;
    }
    cout << numberOfThings << endl;
    return 0;
}
```

The above given example is very trivial and as soon as you read the problem it's very apparent that we can apply Greedy Algorithm to it.

Let's take a bit harder problem - the **Scheduling** problem. You made a list of all the jobs you need to do today, along with the time required to complete the work, and have also assigned a priority ( or weight ) to each work. You want to determine in what order you should sequence the jobs to get the most optimum result.

To solve any problem the first thing we need to look at is **what do we have** ( Inputs ). In the above problem we have **an integer N, number of jobs we need to do**, and **2 lists P and T, one for priority ( or weight ) and other for time required to complete the work**. Now we need to understand what criteria we need to optimize. To explain that, you need to understand about the completion times of these jobs.

**Completion times** is the **total time needed to complete the work**,
$C(j) = T[1] + T[2] + …. + T[j]$ where $1 <= j <= N$

This is because jth work has to wait for the first (j-1) jobs to complete and then requires T[j] time to complete itself.

For example, let T = {1, 2, 3}
So completion times will be
$C(1) = T[1] = 1$
$C(2) = T[1] + T[2] = 1 + 2 = 3$
$C(3) = T[1] + T[2] + T[3] = 1 + 2 + 3 = 6$

In some sense, we obviously want completion times to be as small as possible. But it's

not as simple. In a given sequence, we might observe that the jobs that are earlier on have small completion times and jobs towards the end have big completion times. So what is the optimal way to do that. That depends on our objective function. There are many objective functions in the "Scheduling" problem. Lets discuss about one.

Our **objective function F** is the **weighted sum of completion times.**

F = P[1] * C(1) + P[2] * C(2) + ...... + P[N] * C(N)

We want to minimize the objective function.

First look at some special cases where it is reasonably intuitive what should be the optimal thing to do. Looking at these special cases will then bring forth a couple of natural greedy algorithms. We will then figure out how to narrow these down to just one candidate which we will later prove to be correct.

Lets think about **2 special cases.**

1. If the times required by different jobs are the same, i.e., **T[i] = T[j] where 1 <= i, j <= N**, but they have different priorities then what order do you think it makes sense to schedule the jobs.
2. If the priorities of different jobs are the same, i.e., **P[i] = P[j] where 1 <= i, j <= N**, but they have different lengths then in what order do you think we must schedule the jobs.

So if the times required by different jobs are the same, then we should prefer a work with larger priority.

Look at the objective function that we need to minimize. Let the time required to complete the different work be t.

T[i] = t where 1 <= i <= N

So completion times will be,
C(1) = T[1] = t
C(2) = T[1] + T[2] = 2 * t
C(3) = T[1] + T[2] + T[3] = 3 * t
...
C(N) = N * t

no matter what sequence we use. To make the objective function as small as possible we want the highest priority to be associated with the smallest completion time.

In the second case, if the priorities of different jobs are the same, we always want to

favor work which requires less time to complete. Let the priorities of different jobs be p.

F = P[1] * C(1) + P[2] * C(2) + ...... + P[N] * C(N)
F = p * C(1) + p * C(2) + ...... + p * C(N)
F = p * (C(1) + C(2) + ...... + C(N))

To minimize the value of F we need to minimize (C(1) + C(2) + ...... + C(N)) which can be done if we do work, which require less time to complete, first.

Now we have 2 rules. **One to prefer higher priority and other to prefer less time required.**

Next step is to move beyond special cases, which we understand well, to the general case which perhaps we do not understand. Now all of the priorities are different and all of the times required are different. So if we have 2 jobs and both of these rules give us the same advice. If there is one job that is both having higher priority and takes less time to complete than the other, clearly that job should be done first. But what if both of these rules give us conflicting advice? What if we have a pair of jobs where one of them, on one hand, has higher priority but, on the other hand, takes more time ( i.e. P[i] > P[j] but T[i] > T[j] ). Which one should go first?

We have 2 parameters ( priority and time required ) to select the best work. Can we somehow aggregate these 2 parameters into a single score such that if we sort the jobs from higher score to lower score we will always get an optimal solution?

Remember the 2 rules.

1. As we prefer higher priorities, so **higher priorities should lead to a higher score**
2. As we prefer less time required, so **higher time required should decrease the score**

What kind of simplest possible mathematical function we can use. It takes, as input, 2 numbers ( priority and time required ) and returns a single number ( score ) and the function must satisfy the above 2 properties. ( There are infinite number of such functions )

Lets take two of the simplest functions that have these properties.

**Algorithm #1**: order the jobs by **decreasing value of ( P[i] - T[i] )**
**Algorithm #2**: order the jobs by **decreasing value of ( P[i] / T[i] )**

For simplicity we are assuming that there are **no ties.**

Now we have two algorithms and at least one of them is wrong. So lets rule out one of

the algorithms by showing an example where it does not do the right thing.

Let T = {5, 2} and P = {3, 1}
According to Algorithm #1 ( P[1] - T[1] ) < ( P[2] - T[2] ), so second work should be done
first and our objective function will be
F = P[1] * C(1) + P[2] * C(2) = 1 * 2 + 3 * 7 = 23

According to Algorithm #2 ( P[1] / T[1] ) > ( P[2] / T[2] ), so first work should be done first
and our objective function will be
F = P[1] * C(1) + P[2] * C(2) = 3 * 5 + 1 * 7 = 22

Algorithm #1 is not giving the optimal answer. So Algorithm #1 is not ( **always** ) correct.
Please remember that Greedy Algorithms are often **WRONG**. Just because Algorithm #1
is not correct does not imply that Algorithm #2 is guaranteed to be correct. It does
however turn out that in this case. Algorithm #2 is, happily, always correct.

So our final Algorithm which 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 and t
    , 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
}
```

Lets discuss about its time complexity. We have 2 loops taking O(N) time each and one
sorting function taking O(N * logN). So overall time complexity will be O(2 * N + N *
logN) = **O(N * logN)**.

## D. Proof of Correctness

Now its time to do the hard part. To prove **Algorithm #2** is correct we will use **proof by
contradiction**. This mean that we assume that what we are trying to prove is false and

from that we derive something that is obviously false. So assume that this greedy algorithm does not output an optimal solution and there is some other solution, not output by greedy algorithm, which is better than greedy algorithm. Let **A = Greedy Schedule ( which is not an optimal schedule ), B = Optimal Schedule ( best schedule we 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, 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 ), we 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 )**. Its true because the only schedule that has the property, that the indices only go up, is A = ( 1, 2, 3, ...., N ).

So **B = ( 1, 2, ..., i, j, ... , N )**
**where i > j** Now we have to think about what is the profit or loss if we swap these 2 jobs. First think about what will be the effect of this swap on the completion times of 1. work k other than i and j 2. work i 3. work j

For k, there will be 2 cases. First, when k is on the left of i and j in B. So if we swap i and j then there will be no affect on the completion time of k. Second, when k is on the right of i and j in B. After swapping completion time of k is C(k) = T[1] + T[2] + .. + T[j] + T[i] + .. T[k] Again it will remain same.

For i, before swapping, the completion time was C(i) = T[1] + T[2] + ... + T[i] After swapping, completion time for i is C(i) = T[1] + T[2] + ... + T[j] + T[i] Clearly completion time for i goes up by T[j]. Similarly 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.

### E. Where to use Greedy Algorithms?

A problem must exhibit these two ingredients in order for a greedy algorithm to work:

1. It has **optimal substructures**. Optimal solution to the problem contains optimal

solutions to the sub-problems.

2. It has a **greedy property** (hard to prove its correctness!). If we make a choice that seems best at the moment and solve the remaining subproblems later, we still reach optimal solution. We never have to reconsider our previous choices.

For example:

1. Activity Selection Problem
2. Fractional Knapsack Problem
3. Scheduling Problem

## F. Examples

The greedy method is quite powerful and works well for a wide range of problems. Many algorithms can be viewed as applications of the Greedy Algorithms like:

1. Minimum Spanning Tree
2. Dijkstra's algorithm for shortest paths from a single source
3. Huffman codes ( data-compression codes )
4. and many more.

## G. Practice Problems

1. https://www.hackerearth.com/problem/algorithm/my-girlfriend-and-her-love-for-cats-1/
2. https://www.hackerearth.com/problem/algorithm/chandu-and-consecutive-letters/
3. https://www.hackerearth.com/problem/algorithm/little-jhool-and-his-punishment/

Like 〈 2        **Tweet** 〈 8     8+1 〈 0

---

✏ AUTHOR

### Akash Sharma
💼 Problem Setter and Tester ...
📍 Dehradun
📄 6 notes

---

**Write Note**

**My Notes**

**Drafts**

## TRENDING NOTES

### Mo's algorithm
written by Mike Koltsov

### Greedy Algorithm
written by Akash Sharma

### Dijkstra's Algorithm in Brief
written by Bob Hob

### Minimum Spanning Tree
written by Akash Sharma

### Heaps and Priority Queues
written by Prateek Garg

more ...

| About Us | HackerEarth | Developers |
|---|---|---|
| Blog | API | Judge Environment |
| Engineering Blog | Chrome Extension | Solution Guide |
| Updates & Releases | CodeTable | Problem Setter Guide |
| Team | HackerEarth Academy | Practice Problems |
| Careers | Developer Profile | HackerEarth Challenges |
| In the Press | Resume | College Challenges |
| | Campus Ambassadors | |
| | Get Me Hired | |
| | Privacy | |
| | Terms of Service | |

## Recruit

Developer Sourcing

Lateral Hiring

Campus Hiring

FAQs

Customers

Annual Report

## Connect with us

Map data ©2015 Google

III<sup>rd</sup> Floor, Salarpuria Business Center, 4<sup>th</sup> B Cross Road, 5<sup>th</sup> A Block, Koramangala Industrial Layout, Bangalore, Karnataka 560095, India.
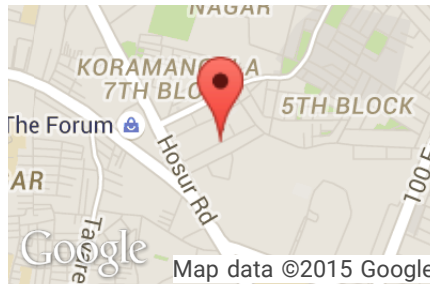
## Reach us

contact@hackerearth.com

+91-80-4155-4695

+1-650-461-4192

Copyright © 2015 HackerEarth Inc.