Algorithms

Solve any problem to achieve a rank

View Leaderboard

Topics: Basics of Greedy Algorithms

Basics of Greedy Algorithms

TUTORIAL PROBLEMS

Introduction

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.

Implementation

```
#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;
    }
```

This example is very trivial and as soon as you read the problem, it is apparent that you can apply the Greedy algorithm to it.

Consider a more difficult problem-the **Scheduling** problem.

You have the following:

- List of all the tasks that you need to complete today
- Time that is required to complete each task
- Priority (or weight) to each work.

You need to determine in what order you should complete the tasks to get the most optimum result.

To solve this problem you need to analyze your inputs. In this problem, your inputs are as follows.

- Integer N for the number of jobs you want to complete
- Lists P: Priority (or weight)
- List T: Time that is required to complete a task

To understand what criteria to optimize, you must determine the total time that is required to complete each task.

C(j) = T[1] + T[2] + ... + T[j] where 1 <= j <= N

This is because **jth** work has to wait till the first (j-1) tasks are completed after which it requires T[j] time for completion.

For example, if T = {1, 2, 3}, the completion time 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

You obviously want completion times to be as short as possible. But it's not that simple.

In a given sequence, the jobs that are queued up at the beginning have a shorter completion time and jobs that are queued up towards the end have longer completion times.

What is the optimal way to complete the tasks?

This depends on your objective function. While there are many objective functions in the "Scheduling" problem, your objective function **F** is the **weighted sum of the completion times**.

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

This objective function must be minimized.

Special cases

Consider the special cases that is reasonably intuitive about what the optimal thing to do is. Looking at these special cases will bring forth a couple of natural greedy algorithms after which you will have to figure out how to narrow these down to just one candidate, which you will prove to be correct.

The two special cases are as follows:

1. If the time required to complete different tasks is the same i.e. **T[i] = T[j] where 1 <= i, j <= N**, but they have different priorities then in what order will it make sense to schedule the jobs?

2. If the priorities of different tasks 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?

If the time required to complete different tasks is the same, then you should give preference to the task with the higher priority.

Case 1 ?

Consider the objective function that you need to minimize. Assume that the time required to complete the different tasks is **t**.

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

Irrespective of what sequence is used, the completion time for each task will be as follows:
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

To make the objective function as small as possible the highest priority must be associated with the shortest completion time.

Case 2

In the second case, if the priorities of different tasks are the same, then you must favor the task that requires the least amount of time to complete. Assume that the priorities of the different tasks is **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, you must minimize (C(1) + C(2) + ... + C(N)), which can be done if you start working on the tasks that require the shortest time to complete.

There are two rules. Give preference to tasks that:

- Have a higher priority
- Take less time to complete

The next step is to move beyond the special cases, to the general case. In this case, the priorities and the time required for each task are different.

If you have 2 tasks and both these rules give you the same advice, then the task that has a higher priority and takes less time to complete is clearly the task that must be completed first. But what if both these rules give you conflicting advice? What if you have a pair of tasks where one of them has a higher priority and the other one requires a longer time to complete? (i.e. P[i] > P[j] but T[i] > T[j]). Which one should you complete first?

Can you aggregate these 2 parameters (time and priority) into a single score such that if you sort the jobs from higher score to lower score you will always get an optimal solution?

Remember the 2 rules.

1. Give preference to higher priorities so that the **higher priorities lead to a higher score**.
2. Give preference to tasks that require less time to complete so that the **more time that is required should decrease the score**.

?

You can use a simple mathematical function, which takes 2 numbers (priority and time required) as the input and returns a single number (score) as output while meeting these two 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 you have two algorithms and at least one of them is wrong. Rule out the algorithm that does not do the right thing.

T = {5, 2} and P = {3, 1}

According to the algorithm #1 (P[1] - T[1]) < (P[2] - T[2]), therefore, the second task should be completed first and your 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]), therefore, the first task should be completed first and your objective function will be:

F = P[1] * C(1) + P[2] * C(2) = 3 * 5 + 1 * 7 = 22

Algorithm #1 will not give you the optimal answer and, therefore, algorithm #1 is not (**always**) correct.

Note: Remember that Greedy algorithms are often **WRONG**. Just because algorithm #1 is not correct, it does not imply that algorithm #2 is guaranteed to be correct. It does, however, turn out that in this case algorithm #2 is always correct.

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 and
their indices

        , 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.

For example:

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

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, such as (includes but is not limited to):

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

Contributed by: Akash Sharma

?