Greedy

Agenda

1. Motivating Problem
2. What is the greedy approach?
3. Pros and cons
    a. Pros
    b. Cons
4. How to prove the correctness of the greedy approach?
5. Example
    a. Problem Analysis
6. To Solve

1. Motivating Problem

Imagine that you have a list of tasks that you need to complete, Each task has an estimated time of completion, and you only have T free time to work on the tasks. Your target is to maximize the number of tasks you can complete in your free time.

How can you do so?

We can think about two different approaches to get the optimal solution. The first one is the brute force approach, which means you need to try all different sets of tasks.

For example, let A be an array of length N that holds the completion times of N tasks.

$A = [6, 1, 5, 3], T = 10$

To try all sets, you need to try 16 different sets of A. That's because the array A has 4 values, and the number of different sets is equal to 2^4 which is 16 sets.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int n, freeTime;
   cin >> n >> freeTime;
   vector<int> completionTime(n);
   for (int i = 0; i < n; ++i) {
       cin >> completionTime[i];
   }
   int maxTasksCnt = 0;
   for (int mask = 0; mask < 1 << n; ++mask) {
       int spentTime = 0, tasksCnt = 0;
       for (int bit = 0; bit < n; ++bit) {
           if ((mask >> bit) & 1) {
               spentTime += completionTime[bit];
               ++tasksCnt;
```

```
            }
        }
        if (spentTime <= freeTime)
            maxTasksCnt = max(maxTasksCnt, tasksCnt);
    }
    cout<< maxTasksCnt;
    return 0;
}
```

While this solution is totally correct, it has a time complexity of $O(2^N)$, which makes it only efficient for small values of N.

A smarter way of thinking is to ask yourself about the best task to start with. Which is always the task with the least completion time. Similarly, the best task to work on next is the task that has the second least completion time, and so on.

By selecting the task that has the least completion time at each moment, you greedily select only one of the available options which appears to be the optimal choice at that moment, and you claim that it will lead to the overall optimal solution. This is called the greedy approach.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, freeTime;
    cin >> n >> freeTime;
    vector<int> completionTime(n);
    for (int i = 0; i < n; ++i) {
        cin >> completionTime[i];
    }
    sort(completionTime.begin(), completionTime.end());
    int curTime = 0, maxTasksCnt = 0;
    for (int i = 0; i < n; ++i) {
        curTime += completionTime[i];
        if (curTime > freeTime)
            break;
```

```
        maxTasksCnt++;
    }
    cout << maxTasksCnt;
    return 0;
}
```

Time complexity: $O(N * log(N))$.

2. What is the greedy approach?

The greedy approach is a problem-solving strategy that involves making locally optimal decisions at each step, aiming for an optimal overall solution. It builds the solution incrementally by selecting the best available option without considering potential consequences or exploring all possibilities. It only focuses on immediate gains. Thus, it does not backtrack or reconsider previous choices.

While the greedy approach assumes that locally optimal decisions lead to a globally optimal solution, this is not always the case, and it does not guarantee the best solution in all scenarios.

3. Pros and cons:

- **Pros:**
    - **Simplicity**: Greedy algorithms are often easy to understand and implement.
    - **Efficiency**: Greedy algorithms typically have a fast runtime due to their greedy nature.
    - **Intuitive**: The greedy approach aligns with human intuition, as it focuses on making locally optimal choices at each step.
- **Cons:**
    - **Lack of global optimization**: The greedy approach may not always lead to the globally optimal solution, as it prioritizes immediate gains without considering the long-term consequences.

- **Dependency on problem structure**: The effectiveness of the greedy approach depends on the problem's specific structure and constraints. It may not be applicable or suitable for every problem.
- **Difficulty in identifying the greedy choice**: Determining the correct greedy choice at each step can be challenging, requiring careful analysis and insight into the problem.

4. How to prove the correctness of the greedy approach?

In order to prove the correctness of a greedy solution, you need to prove that it satisfies the following two properties:

- **Greedy choice property:** a globally optimal solution can be obtained by greedily selecting a locally optimal choice.
- **Optimal substructure property:** a globally optimal solution to the problem contains the optimal solutions of all its subproblems.

For example, to prove that the solution to the above problem is correct, let's prove that it satisfies both properties:

- **Greedy choice property:** The greedy approach selects the task with the least completion time at each moment. By assuming the existence of a better alternative solution, we can make a swap to improve it. This either maintains the same number of completed tasks or increases it. Therefore, the greedy approach leads to a globally optimal solution.
- **Optimal substructure property:** The greedy approach guarantees the optimal substructure property by always choosing the task with the least completion time. This ensures that the locally optimal choices contribute to the overall optimal solution. Hence, the optimal substructure property is satisfied.

5. Example:

Consider a set of requests for a room. Only one person can reserve the room at a time, and you want to allow the maximum number of requests. Each request is a period of time $[S, E]$. Which ones should we schedule?

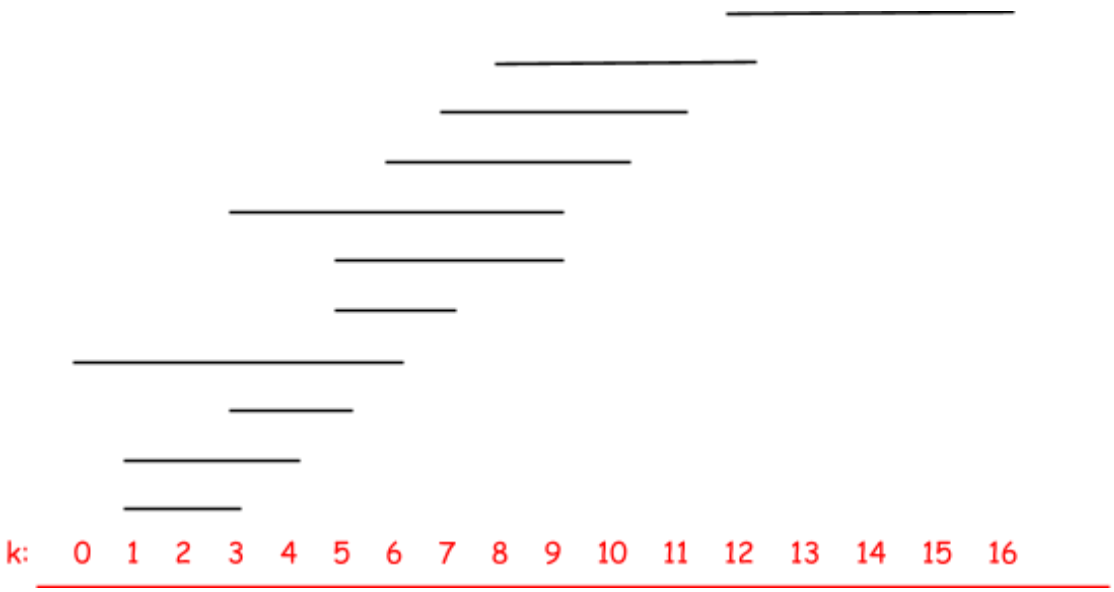
```
Input:
(1, 4), (1, 3), (5, 9), (0, 6), (3, 5), (5, 7), (3, 9), (6, 10), (12, 16), (7, 11), (8, 12)

Output:
(1, 3), (3, 5), (5, 7), (7, 11), (12, 16)
```

Problem Analysis:

- The overall problem is to find the maximum number of requests that can be scheduled with no limit for the overall end time (the room is initially available all the time). The subproblem is to find the maximum number of requests to schedule until a time value of k.
- To approach this problem greedily, we will solve each subproblem incrementally in ascending order of k until the overall problem is solved.

The timeline of the requests along with the value of k is as follows:



Let's solve the subproblems for each k that corresponds to an end time of a received request:

1. At $k = 3$, the request $(1, 3)$ is received so we can schedule it.
2. At $k = 4$, the request $(1, 4)$ is received. We have two solutions that are both optimal and last no longer than 4:

   – $(1, 3)$.
   – $(1, 4)$.

   Let's stop here for a bit. If you're to pick only one solution of them, which one is more likely to allow you to schedule more requests in the future? It is the solution that ends the earliest of all solutions, which is the first one. Thus, from now on we'll ignore the second option.
3. At $k = 5$, the request $(3, 5)$ is received. One optimal solution exists.

   – $(1, 3), (3, 5)$.
4. At $k = 6$, the request $(0, 6)$ is received. Two valid solutions exist but only one is optimal.

   – $(0, 6)$ -> will be ignored because we have a more optimal solution.
   – $(1, 3), (3, 5)$ -> optimal.
5. At $k = 7, 8$, the request $(5, 7)$ is received. One optimal solution exists.

   – $(1, 3), (3, 5), (5, 7)$ -> optimal.
6. At $k = 9$, two requests $(5, 9), (3, 9)$ are received. Both of them conflict with the optimal solution we got so far. The question is: do we need to reconsider previous solutions that don't conflict with the new requests? The answer is no, because even if such solutions exist, these solutions are not optimal which means they have less number of scheduled requests, adding one of the newly received requests to them will at most make them have the same number of scheduled requests as the current optimal solution. Thus, no need to do this.
7. At $k = 10$, the request $(6, 10)$ is received which conflict with the current

optimal solution and will be ignored as explained.

8. At $k = 11$, the request $(7, 11)$ is received. It doesn't conflict with the current optimal solution, thus the optimal solution is expanded.
   – $(1, 3), (3, 5), (5, 7), (7, 11)$.

9. At $k = 12$, the request $(8, 12)$ is received which conflict with the current optimal solution and will be ignored as explained.

10. At $k = 16$, the request $(12, 16)$ is received. It doesn't conflict with the current optimal solution, thus the optimal solution is expanded.
    – $(1, 3), (3, 5), (5, 7), (7, 11), (12, 16)$.

And we're done!

Algorithm steps:

1. Sort the requests based on end times to prioritize requests that ends the earliest.
2. Initialize the variable $last$ with -1, representing the last reserved time. Also, let $requests$ be a vector that holds the optimal solution.
3. Loop through the requests from left to right. For each request i:
   a. If S_i is greater than $last$, then no conflict with the current optimal solution. Thus, this request is scheduled and added to our optimal solution:
      i. Add it to vector $requests$.
      ii. Update $last$ to E_i.
4. Print the vector $requests$.

It's easy to notice that this solution satisfies the two properties of the greedy approach: each local optimal greedy decision leads to the overall optimal solution, and the overall optimal solution to the problem contains the optimal solutions of all its subproblems.

Code:

```
#include <bits/stdc++.h>
using namespace std;

bool cmp(pair<int,int> &i, pair<int,int> &j) {
   return (i.second < j.second || (i.second == j.second && i.first <
j.first));
}

int main() {
   int n, l, r;
   vector<pair<int,int>> v, requests;
   cin >> n;
   for (int i = 0; i < n; ++i) {
       cin >> l >> r;
       v.emplace_back(l, r);
   }
   sort(v.begin(), v.end(), cmp);
   int last = -1;
   for (int i = 0; i < n; ++i) {
       if (v[i].first > last) {
           last = v[i].second;
           requests.emplace_back(v[i]);
       }
   }
   cout << requests.size() <<"\n";
   for (auto request: requests)
       cout << request.first <<" "<< request.second <<"\n";
   return 0;
}
```

Time complexity: $O(N * log(N))$.

6. To Solve:

- Restaurant
- Matchmaker
- Dominated Subarray
- Regular Bracket Sequence