Greedy Algorithms

Atli FF

September 9, 2024

School of Computer Science

Reykjavík University

Greedy algorithms

- An algorithm that always makes *locally* optimal moves is called greedy
- For some kinds of problems this will give a *globally* optimal solution as well
- Seeing when this is the case can be very tricky, and if used in the wrong context the solution will get a WA verdict

Submitting greedy solutions

- The tricky thing with these solutions are that it's often hard to know if you've made a mistake and thus get WA or if there's some hole in the greedy algorithm
- It's often easy to think of all kinds of greedy solutions, but they are very often wrong
- Generally one would like to consider complete search or dynamic programming (will see this later) first, but some problems do require greedy solutions

Coin change

- A classical example is making change. Say you want to sum up n and have only denominations of $1, 5$ and 10, what's the least amount of coins you can give back?
- The greedy solution would be to just always give the biggest coin you can that's not too much. So for say 24 we'd do $10, 10, 1, 1, 1, 1$.
- Is this always optimal?

Coin change

- Well, it turns out to depend on the denominations. Say we have denominations of $1, 8$ and 20.
- For $n = 24$ we then give back $20, 1, 1, 1, 1$ instead of the optimal $8, 8, 8$.
- We will come back to this problem when we solve the general case using dynamic programming.

Lilypad jump

- Consider a frog jumping on a sequence of lily pads, there is one at $x = 0$ and one at $x = n$, with some amount of lilypads in between
- The frog can jump at most distance r
- When at a given lily pad, what's the best move?

Lilypad jump

- Clearly just jump as far right as possible!
- But be careful, this is very contingent on the frog being able to jump any distance in $[0, r]$
- If it could jump any distance in $[r/2, r]$, it would not be true for example

Taxi assignment

- Let's consider another problem. You are managing a taxi company and today n drivers showed up and you have m cars.
- But not all drivers and cars are created equal. Car i has h_i horsepower and driver j can only handle at most g_j horsepower.
- What's the greatest number of drivers you can pair to cars such that they can handle their car?

The greedy step

- The greedy idea here is to simply pair each car to the worst driver that can still handle that car.
- Thus we start by sorting the drives and cars and then simply linearly walk through each and pair them together.
- It might not be obvious, but this actually gives the best answer.

Implementation

```
int main() {
    int n, m; cin >> n >> m;
    vi a(n), b(m);
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; i < m; ++i) cin >> b[i];
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    int ans = 0;
    for(int i = 0, j = 0; i < m; ++i) {
        while(j < n && a[j] < b[i]) j++;
        if(j < n) ans++, j++;
    }
    cout << ans << '\n';
}
```

Sorting

- Greedy algorithms very often involve sorting
- More generally they often involve always picking the "extremal" option out of the local options, in some sense
- Biggest, shortest, cheapest, first, etc.

Job scheduling

- Say we have a list of jobs, each starting at some time s_j and finishing at some time f_j
- What's the largest amount of jobs we can complete if they can't overlap?

Solution

- The solution is shockingly simple, but not obviously correct
- Order the jobs by completion time f_j and then walk through them
- If you can complete a job in addition to the ones you've already picked, pick it
- The jobs you've picked by the end are the solution

Proof of correctness

- Why is this correct though? Let's prove it.
- Suppose the algorithm is not optimal. Say we pick jobs of indices $i_1, i_2, \ldots, i_k$ but a better solution picks $j_1, j_2, \ldots, j_l$.
- Say the solutions agree on the first r jobs (possibly 0).
- Now neither i_{r+1} nor j_{r+1} clash with the jobs $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$. But because we ordered things by end time, we must have that job i_{r+1} ends no later than j_{r+1}. But then we could just as well have picked i_{r+1}. But this holds for any r, so by induction we have that $i_1, \ldots, i_k$ is no worse than $j_1, \ldots, j_l$, which gives a contradiction.
- Thus the algorithm is optimal.

Many more

- There are many many more and we will see plenty in the course
- Many famous algorithms are famous because they perform non-trivial greedy steps
- Dijkstra's algorithm, Huffman coding, Kruskal's algorithm, Horn satisfiability and many more