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

Proving correctness

- How do we prove the greedy algorithms are correct?
- When in programming contests this is usually overkill, but at a workplace it is generally not
- There are two main common arguments that tackle this, but often novel methods are needed
- These are usually called "Greedy stays ahead" and "Exchange arguments"

Proving correctness

- "Greedy stays ahead" aims to prove that during each greedy step it consistently stays ahead of all other possible choices
- "Exchange arguments" aims to show that you can turn any solution into the greedy solution with a sequence of "exchanges" without making them any worse, so the greedy solution is as good as it gets
- Sometimes a part of this is showing that the greedy solution outputs a valid solution at all, as that is not obvious in every case

Proving correctness

- Consider the lilypads again.
- We'll now prove it's optimal, which is more work than it may seem at first
- First we prove that the greedy solution reaches the final lily pad if there is a path there
- Note again this is not true for many minor variants of the problem!

Proving correctness

By contradiction; suppose it did not. Let the positions of the lilypads be $x_1 < x_2 < \cdots < x_m$. Since our algorithm didn't find a path, it must have stopped at some lilypad x_k and not been able to jump to a future lilypad. In particular, this means it could not jump to lilypad $k + 1$, so $x_k + r < x_{k+1}$. Since there is a path from lilypad 1 to the lilypad m, there must be some jump in that path that starts before lilypad $k + 1$ and ends at or after lilypad $k + 1$. This jump can't be made from lilypad k, so it must have been made from lilypad s for some $s < k$. But then we have $x_s + r < x_k + r < x_{k+1}$, so this jump is illegal. We have reached a contradiction, so our assumption was wrong and our algorithm always finds a path.

Proving correctness

- Let's now show it actually stays ahead of any optimal solution
- Let J be the set of jumps from our greedy algorithm and J^* be an optimal set of jumps
- Then $|J| \geq |J^*|$ since it's optimal
- Let $p(i, J)$ be the position after taking the first i jumps in J
- Let's prove that for all i we have $p(i, J) \geq p(i, J^*)$

Proving correctness

We proceed by induction. As a base case, if $i = 0$, then $p(0, J) = 0 \geq 0 = p(0, J*)$ since the frog hasn't moved. For the inductive step, assume that the claim holds for some $0 \leq i < |J^*|$. We will prove the claim holds for $i + 1$ by considering two cases:

- Case 1: $p(i, J) \geq p(i + 1, J^*)$. Since each jump moves forward, we have $p(i + 1, J) \geq p(i, J)$, so we have $p(i + 1, J) \geq p(i + 1, J^*)$.
- Case 2: $p(i, J) < p(i + 1, J^*)$. Each jump is of size at most r, so $p(i + 1, J^*) \leq p(i, J^*) + r$. By the inductive hypothesis, we know $p(i, J) \geq p(i, J^*)$, so $p(i + 1, J^*) \leq p(i, J) + r$. Therefore, the greedy algorithm can jump to position at least $p(i + 1, J^*)$. Therefore, $p(i + 1, J) \geq p(i + 1, J^*)$.

So $p(i + 1, J) \geq p(i + 1, J^*)$, completing the induction.

Proving correctness

- And now we are almost done!
- Finally we just have to prove that $|J| = |J^*|$, which would mean the greedy is always optimal

Proving correctness

Since J^* is an optimal solution, we know that $|J^*| \leq |J|$. We will prove $|J^*| \geq |J|$. Suppose for contradiction that $|J^*| < |J|$. Let $k = |J^*|$. From before, we have $p(k, J^*) \leq p(k, J)$. Because the frog arrives at position n after k jumps along series J^*, we know $n \leq p(k, J)$. Because the greedy algorithm never jumps past position n, we know $p(k, J) \leq n$, so $n = p(k, J)$. Since $|J^*| < |J|$, the greedy algorithm must have taken another jump after its k-th jump, contradicting that the algorithm stops after reaching position n. We have reached a contradiction, so our assumption was wrong and $|J^*| = |J|$, so the greedy algorithm produces an optimal solution.

Proving correctness

- That was a lot of effort!
- Now, which kind of greedy proof was that?

Proving correctness

- That was a lot of effort!
- Now, which kind of greedy proof was that?
- It was a "Greedy stays ahead" proof, so let us next see an exchange proof

Proving correctness

- To consider exchange arguments we have to do things slightly out of order and nab an algorithm from the future week of graph theory
- Consider a set of houses, we want to lay fibre cable between them such that each house is connected to every other house through some set of cables
- Doesn't have to be a direct connection, just some path exists between them
- For each pair of houses we are given the cost of laying a cable between them
- What's the cheapest cable-laying procedure?

Proving correctness

- Our greedy algorithm will be as follows
- Start with just a single house and consider it "active"
- Choose the cheapest cable that goes between an active house and one that is not active
- Make the newly connected house active, and keep going
- Now we prove this is optimal!

Proving correctness

Let T be the set of cables chosen by our algorithm and T^* be some optimal set. Let $c(T)$ denote the total cost of a set of cables. We will prove $c(T) = c(T^*)$. If $T = T^*$ the result is obvious, so we can assume $T \neq T^*$. Then let (u, v) be a cable in $T \setminus T^*$. Let S be the set of active houses when (u, v) was added to T and H be the set of all houses. Then (u, v) is the cheapest edge between S and $H \setminus S$. Since T^* connects all houses it must contain some path from u to v. This path begins in S and ends in $H \setminus S$ so there must be some cable (x, y) such that $x \in S$ and $y \in H \setminus S$. Since (u, v) is the cheapest such edge we must have $c(\{(u, v)\}) \leq c(\{(x, y)\})$. Let $T' = T^* \cup \{(u, v)\} \setminus \{(x, y)\}$.

Proving correctness

Since every house in S can reach every other house in S without using (u, v), T' is valid for those houses, and the same goes for $H \setminus S$. But then T' allows any house in S to reach u, then go to v, then to any house in $H \setminus S$. So T' is a valid set of connections. But note that $c(T') = c(T^*) - c(\{(x, y)\}) + c(\{(u, v)\}) \leq c(T^*)$. Since T^* is optimal this means $c(T') \geq c(T^*)$, so $c(T') = c(T^*)$. Note that $|T \setminus T'| = |T \setminus T^*| - 1$, so if we repeat this same argument once for each edge in $T \setminus T^*$ we will have converted T^* into T without changing T, thus $c(T) = c(T^*)$.

Proving correctness

- Very mathy!
- But that is an example of an "Exchange argument" proof of correctness