Competitive programming proofs

Siddharth Bhat

Monsoon, second year of the plague

Contents

0.1 Codeforces 1389C: woodcutters

Problem link is https://codeforces.com/problemset/problem/1389/C.

- Solution is greedy.
- It's hopefully clear that it's always optimal to have the first tree lean left and the last tree lean right. So we now have the prove that the construction for trees '[1..n-2]' is optimal.
- Let O_* be the optimal solution, O be the solution discovered by the above algorithm. Let index i be the first index where O_* and O decide to do something different to a tree. We have $i > 0, i < n-2$ since we assume that the algorithms agree on the first and last tree.
- We have 6 cases at index 'i': the possibilities are 'L' for leaning left, 'U' for standing up, and 'R' for leaning right.
  - $O_* = L, O = U$: If there is enough space to lean left, O will choose to lean left. This violates the construction of O.
  - $O_* = L, O = R$: Same as above; If there is space to lean left, O will choose to lean left. This violates the construction of O.
  - $O_* = U, O = L$: Since O and O_* agree upto i and O chooses to lean left, there must be enough space to lean left for O_*. This creates a strictly better solution, which violates the optimality of O_*.

  - $O_* = U, O = R$: Now what? we can't control O based on O_* or vice versa. We'll return to this case at the end, as it's the most complex.
  - $O_* = R, O = L$: There must be enough space for O_* to lean L. $O_* = R$ only limits the space available for the next tree. So we can modify O_* to lean left and reach the same optimality.
  - $O_* = R, O = U$: If it's possible to lean right, then O will choose to do so. This violates the construction of O.

- $O_* = U, O = R$: This is the complex case.
  - In this case, let's consider the sequence of trees felled by O_* after i.
  - Let us suppose that O_* fells many trees rightward, followed by a tree that is felled in the direction d_* (where $d_* \in \{L, U\} \neq R$. Formally, $O_*[i, i+1, k] = U; R^k; d_*$ for some $d_* \neq R$. (That is, $d_* \equiv O_*[k]$).
  - If such a d_O does not exist, then O can mimic what O_* does, since O's decision to move $O[i]$ right does not impact any tree in the future.
  - If such a d_* does exist, then let us consider what d_* is.
  - If $d_* \equiv O_*[k]$ is U, then we can have $O[k] = U$.
  - If $d_* \equiv O_*[k]$ is L, then we can have $O[k] = U$.

0.2 Codeforces 1175 C: Electrification

sliding window argument.

I will proof this by a simple contradiction argument Assume that at least one of the N nearest neighbours of x does not belong to a continuous window of size N containing x. Let us assume this nearest neighbour is Aj. Without loss of generality, assume Aj<x (same proof works for Aj>x). As the window is not continuous as per our assumption, there must exist some element Ak between Aj and x which is not a nearest neighbour of x. But distance of Ak from x is clearly less than distance of Aj from x. So if Aj is a nearest neighbour, Ak must also be a nearest neighbour. That contradicts our assumption.

0.3 Codeforces 190D: Non-Secret Cypher

> Yeputons It's kind of optimization. Consider problem 190D - Non-Secret Cypher. The stupid solution goes over all subarrays and checks whether it's good or not. Now we notice that if subarray is 'good', then all its superarrays is 'good' too. Let's go over all left borders $1 \le x \le N$ and for each of them find r_x. rx is a minimal number such that subarray $x \dots r_x$ is good. Obviously, all subarrays which starts at x and ends after r_x is good. If you notice that $r_1 \le r_2 \le \dots \le r_N$, you can use 'two pointers' method. The first pointer is x and the second one is r_x. When you move the first one right, the second one *can only move right* too. No

matter how many operations you perform in one step, algo's running time is $O(n)$, because each of pointers makes $\leq N$ steps.

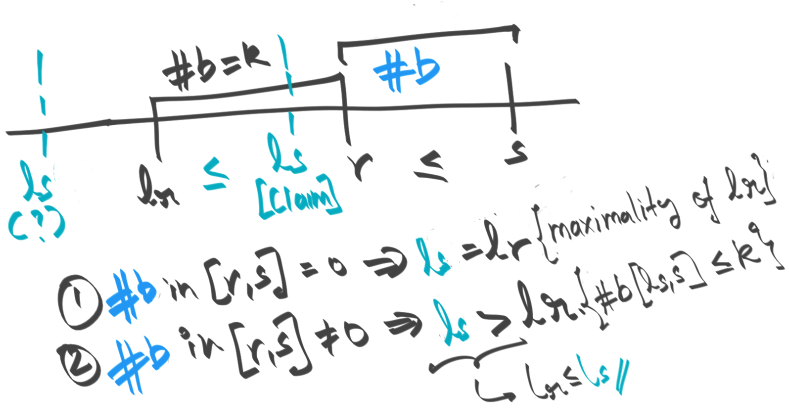
0.4 Codeforces 1389C:

- It's safe to look for a longest satisfying match with DP, and if the match is of odd length, to decrement it to get an even length string.
- This would NOT be safe if we had to chop of an arbitrary length δ!
  - To instill fear, suppose $\delta = 5$, and that on some problem instance, the best string had length 9, second best had length 7.
  - Since the DP only keeps track of the longest length, we would have tracked 9, and then subtracted 5 to report $9-5=4$ which is much less than the real second-optima of 7.
- The reason this is safe when we chop off length 1 is because:
  - We already know that, say, 9 is the longest string length. There are no longer solutions that can compete. So we can must only worry about shorter solutions
  - Amongst the second largest solutions, we must worry about the second largest solution being $8, 7, 6, \ldots$.
  - Since we only decrement by 1, our "mutated" largest solutoin will be 8. This is as large as the maximum second-largest solution.
  - So we do not lost any second-best optima, since we mutate the first best optima to drop by 1, which makes it the best-in-class second best optima.

0.5 Codeforces 676C: Vasya and String

- Problem link: https://codeforces.com/contest/676/problem/C
- Here, we learn how to write a proof of correctness for sliding window where the window length is *upper bounded*. That is, we have a *maximum* length of sliding window that we cannot exceed.
- *Key Idea 1*: consider all windows that contain legal solutions. Suppose we have two windows $[l_1, r_1]$ and $[l_2, r_2]$ where $r_1 < r_2$ (that is, window 1 ends before window 2). Then, we wish to show that $l_1 < l_2$ (that is, window 1 begins before window 2).
- This condition implies that when we make progress by moving $r_1 \to r_2$, we can similarly make forward progress moving $l_1 \to l_2$. We would not have to go backwards, like $l_2 \leftarrow l_1$. So when we move from r to $(r+1)$, we need to decide how much we must move l. We know that this will not lose solutions from the prior invariant.

Let's consider a simpler subproblem, where are given a string of `'a'`s and `'b'`s and we must find the longest substring with at most k `'b'`s. The original problem can be solved by solving this problem twice, once with `'a, b'` and once by swapping these letters.



- At this subproblem, let $[p, x]$ and $[q, y]$ be two legal intervals. So $s[p \ldots x]$ has at most k `'b'`s, as does $s[q \ldots y]$. Suppose that $x \le y$ (that is, $y = x + \epsilon$) for some $\epsilon \ge 0$. We wish to show that $p \le q$ (that is, $q = p + \delta$ for some $\delta \ge 0$).

```
// https://codeforces.com/contest/676/submission/121164663
void main() {
  ...
  int best = 0;
```

```
  for (int c = 'a'; c <= 'b'; ++c) {
    int l = 0;
    int changed = 0;
    // [l, r]
    for (int r = 0; r < n; ++r) {
      // change to c
      if (s[r] != c) { changed++; }
      while (changed > k) {
          if (s[l] != c) { changed--; }
          l++;
      }
      best = max(best, r - l + 1);
    }
  }
  cout << best;
}
```

Chapter 1

Binary search

1.1 Binary Search With Closed-Closed Intervals

```
// precondition: arr[0] <= arr[1] .. <= arr[n-1]
int binsearch(int *arr, int v, int n) {
  int left = 0, right = n-1;
  while(left <= right) {
      int mid = left + (right - left)/2;
      if (arr[mid] == v) {
          return mid;
      } else if (arr[mid] > v) {
          // search in left subrange.
          right = mid-1;
      } else {
          // search in right subrange:
          // arr[mid] < v.
          left = mid+1;
      }
  }
}
```

1.2 Leftmost Index That Satisfies Predicate

Key idea, separate out the binary search range from the variable that tracks the best index. One style of implementing this is to ensure the the `left` variable keeps track of the leftmost index that satisfies this property. But this burdens the variable with two tasks:

- keeping track of the satisfying leftmost index
- keep track of the range we are exploring

It is far better to separate these two concerns. So we keep a `best` variable to keep track of the leftmost index we have seen so far. This frees us to freely manipulate the `left`, `right` variables as we like.

```
// FFFFF TTTTTT
//       ^
//       |
//     output
int left = 0, right = n;
int best = -1;
while(left <= right) {
    int mid = left + (right - left)/2;
    bool p = solve(mid);
    if (p) {
        best = min(best, mid);
        // explore left
        // find smaller indexes with T
        right = mid-1;
    } else {
        // property is not true
        // need to move higher.
        left = mid+1;
    }
}
cout << best;
```

1.3 Rightmost Index That Satisfies Predicate

```
// TTTTTT FFFFFFF
//       ^
//       |
//     output
int left = 0, right = n;
int best = -1;
while(left <= right) {
    int mid = left + (right - left)/2;
    bool p = solve(mid);
    if (p) {
        best = max(best, mid);
        // explore right
        // find larger indexes with T
        left = mid+1;
    } else {
        // property is not true,
        // need to move lower.
```

```
            right = mid-1;
        }
    }
```