

Greedy Algorithms: More Examples

Some of the "easier" greedy solutions to discover are ones that involve the following principle: when working in parallel, start the tasks that take the longest first. The next two problems illustrate this principle.

NCPC 2013: Planting Trees

Here is the link to the problem:

<https://open.kattis.com/problems/plantingtrees>

We must plant all of our trees, one per day, to get ready for a party. We want to throw the party as soon as possible. For each tree, we know how long it will take to grow.

The later we plant a tree that takes a long time to grow, the worse off we are. Thus, the greedy strategy that works is to plant the trees in order from longest to shortest growing time. We can use the exchange argument to prove that this strategy is correct.

The proof roughly looks like this:

Let's say a competing algorithm claims to be better than ours. We plant our trees in the order:

$t_1, t_2, \dots$, where $t_1 \geq t_2 \geq \dots \geq t_n$.

There must be some first index i , such that they choose t_j in position i instead of t_i , where $t_j \leq t_i$ and $j > i$. For our algorithm the i^{th} tree will be ready in $i + 1 + t_i$ days from the start of growing trees, but for the competing algorithm it will be done in $j + 1 + t_i$ days. This is at least as many days our algorithm. In addition, for our tree j , it will be done in $j + 1 + t_j$ days, which is ALSO less than or equal to in $j + 1 + t_i$ days, contradicting the fact that the competing algorithm beats our algorithm.

UCF Locals 2014: Fujiyama Thursday

The problem is here on page 7 of this document:

<https://lpc.ucfprogrammingteam.org/localFiles/local2014Problems.pdf>

This problem is inspired by an actual tradition that still lives on for UCF's programming team to this day (2026), of team members going out for sushi dinner on Thursday evenings.

In the problem there are several cars that will be driving to dinner. Each car holds 4 people. Each car takes a different amount of time to get to the restaurant. Each person takes a different amount of time to eat. Arrange the cars so that the total amount of time it takes from starting to drive to dinner to the point in time where all people finish eating is minimize.

Using the same principle as the problem above, we need to get the people who take the longest to eat to dinner first, so put these people in the fastest cars. So, sort the people's eating times in reverse order and the car driving times in regular sorted order and match accordingly!

Last Alphabetic Subsequence (Top Coder SRM 518 D1 250)

Problem: Given an alphabetic string, find the subsequence that comes last alphabetically in the string.

Any subsequence that starts at the “latest possible letter” is better than other competing subsequences. Thus, our first letter should be the alphabetically last letter that occurs in the string. As we are iterating through the letters, replace our “current answer” completely, if we see a letter that comes after the first letter in our current answer.

Adding a letter to any string makes it alphabetically after its prefix, so by default, we would always want to add a new letter to a non-empty string.

If we have some string “geb”, for example, if the next letter is “f”, then we see that “gf” is better than “gef” or “gebf”.

Once we make this observation, then our algorithm is streamlined as follows:

- 1) Keep a stack of letters – initialize it as empty.

Repeat the following steps as you read through each letter in the input string:

- 2) When reading in a new letter, pop all letters off the stack that come strictly before the current letter, alphabetically.
- 3) Push the current letter on the stack.

When we finish, the subsequence we desire can be read off the stack from the bottom to the top. (In code, if we were using a stack, we’d pop each item off sequentially, storing them from back to front.)

The key in developing this algorithm is to note that given any previous subsequence that is the ‘best’, we develop our new best by “building off” the old best.

Up and Down – 2014 Google Code Jam Round 2

The problem is as follows:

You are given a sequence of distinct integers $A = [A_1, A_2, \dots, A_N]$, and would like to rearrange it into an *up and down* sequence (one where $A_1 < A_2 < \dots < A_m > A_{m+1} > \dots > A_N$ for some index m , with m between 1 and N inclusive).

The rearrangement is accomplished by swapping two *adjacent* elements of the sequence at a time. Predictably, you are particularly interested in the minimum number of such swaps needed to reach an *up and down* sequence.

When studying sorting algorithms, we prove that the average case run-time of algorithms like bubble sort must be $\theta(n^2)$ for sorting n items because each swap removes at most one inversion from the array and the average array has about $\frac{1}{4}n^2$ inversion. Thus, our analysis for this problem should focus on a similar count – the number of inversions in the array. Note: an inversion in an array is a pair of items such that $i < j$ but $a[i] > a[j]$. Namely, when looking at the pair of items in relation to each other, they are “out of order.” We can count inversions by looping through each pair of elements in an array and adding 1 if a pair is inverted. In a random array, we expect about half of all pairs to be inverted.

The difference in this problem is that there is some split point (we don’t know where) and to the left of that split point there are no inversions and to the right of it, if we consider the list in reverse order, there are no inversions. Given the type of move suggested, this means that we can remove at most one inversion per swap.

Thus, our proof of correctness looks like this:

- 1) We must have the array ordering be one of these possible sequences. (UUUDD or UDDDD, etc., where U represents a pair going up and D a pair going down.)
- 2) If we consider all of these possibilities and look at the minimum number of inversions (with respect to this specific problem), then it’ll be impossible to have any competing set of swaps beat that number of inversions.
- 3) Finally, if we know the number of inversions in a subarray there always exists a way to remove one inversion per swap (otherwise the array would be sorted!) until no inversions exist.

Once we sketch this out, our algorithm is fairly easy to come up with.

- 1) Try each possible location as the place for the maximum element.
- 2) For each of these locations, count the number of inversions in the appropriate subarrays.
- 3) Take the minimum of all values calculated in step 2.