NOI.PH Training: DP 4

DP Optimizations

Kevin Charles Atienza

Contents

1 Introduction

This module is mostly about DP, but we'll motivate its theme with a non-DP problem.

Problem 1.1. There are n cities isolated from each other and from the rest of the world. In the name of globalization, you wish to make them all interconnected.

The cities, labelled 1 to n, are located in a circle. You can only build roads or highways between two consecutive cities i and $i+1$. (This wraps around, so city $n+1$ is the same as city 1.) You have three available actions:

1. Build an *airport* at city i. This costs 0 dollars. Initially, there are no airports.
2. Build a *highway* between two cities i and $i+1$. The cost varies depending on how many highways have already been built. Specifically, the cost of the kth highway built is $ak+b$. (In other words, the costs are $a+b$, $2a+b$, $3a+b$, ...)
3. Build a *road* between two cities i and $i+1$. This costs c_i dollars.

We consider a city to be *connected to the world* if it has an airport, or is connected to some city that has airport via a series of roads or highways.

What is the minimum cost to connect all n cities to the world, assuming you can only build at most p airports? Also, you must answer this question for each p from 1 to n.

Constraints:

- $1 \le n \le 2 \cdot 10^5$
- $1 \le a, b, c_i \le 10^{12}$.

Before proceeding, I encourage you to try solving this first! Your goal is to find an $\mathcal{O}(n^2)$ solution, which is easily obtainable just from what you've learned so far from previous modules.

In the following part, I will describe this $\mathcal{O}(n^2)$ solution. If you've already found one, feel free to skip it and move to the next part.

1.1 Towards a quadratic solution

To begin with, here are some observations:

- If $p = n$, then we can just place airports everywhere, so the answer is 0.
- If $p = n-1$, then we only need to build one highway or road, whichever one's cheaper. In fact, we can connect *any pair* of consecutive cities whatsoever, because after doing so, we can simply place an airport in one of the cities in this pair, and place the remaining $n-2$ airports in the remaining cities.
- You can proceed with the previous line of thought to arrive at the following general observation: if there are p airports, then we only need to add enough highways or roads to ensure that there are p *connected components*, and then we can place an airport in some city in each component. But every graph with n nodes and p components has at least $n-p$ edges, so we need to add at least $n-p$ highways or roads. In addition, if there are no cycles, then there are *exactly* $n-p$ edges.

Of course, the fewer the edges, the cheaper, so we want exactly $n-p$ edges. But in our case, it's impossible to create a cycle with only $n-p$ edges (why?), so in fact, adding *any* $n-p$ edges is sufficient as well, and we'll always end up with p connected components.

So we want to add $n-p$ highways or roads. We don't know how many of these will be highways and how many will be roads, so let's just try all possibilities. If there are r roads, then there will be $h := n-p-r$ highways. Since the highway cost doesn't depend on which pair of cities we're connecting, it makes sense to build roads first and make them as cheap as possible. After building the r roads, we just build h highways in some unconnected pairs, and the cost will always be

$$(a+b)+(2a+b)+(3a+b)+\ldots+(ha+b).$$

So now, we want to build r roads. If we're minimizing the cost, it makes sense to sort the possible roads in increasing cost, and then attempt to build them in that order. As we've seen before, we won't form any cycle this way. Thus, we may simply build the r cheapest roads.

1.2 A quadratic solution

Let's now summarize what we know so far. First, we want to sort $c_1, c_2, \ldots, c_n$. Let $s_1, s_2, \ldots, s_n$ be their sorted version. Then for any given p, the answer must be

$$ans(p) = \min_{\substack{h\geq 0\\ r\geq 0\\ h+r=n-p}} \left((a+b)+(2a+b)+\ldots+(ha+b)\right)+\left(s_1+s_2+\ldots+s_r\right).$$

By precomputing the prefix sums of the arrays $[a+b, 2a+b, 3a+b, \ldots]$ and $[s_1, s_2, s_3, \ldots]$, the inner expression can be computed in $\mathcal{O}(1)$ given h and r. Since there are $\mathcal{O}(n)$ possibilities for the pairs (h, r), the whole answer can be computed in $\mathcal{O}(n)$. Finally, solving it for each p from 1 to n means this solution runs in $\mathcal{O}(n^2)$ time.

1.3 A faster-than-quadratic solution

Now, can we do better than this?

The above expression is sometimes called a "min-plus convolution", because it's of the form

$$A[k] = \min_{\substack{x\geq 0\\ y\geq 0\\ x+y=k}} R[x]+S[y]$$

for some arrays R and S, namely the prefix sums of $[a+b, 2a+b, \ldots]$ and $[s_1, s_2, \ldots]$, respectively. Sometimes, you'll see this in the following equivalent form

$$A[k] = \min_{0\leq x\leq k} R[x]+S[k-x].$$

Our goal is to compute the array $[A[0], \ldots, A[n-1]]$ faster than the straightforward $\mathcal{O}(n^2)$.

Now, as it turns out, for *arbitrary* arrays R and S there doesn't seem to be any known way of doing a min-plus convolution faster than quadratic time. Indeed, searching the internet with the keywords "min-plus convolution" (and the related "max-plus convolution") reveals that our

best algorithms for it currently run in essentially quadratic time.[1] In fact, it is an open question whether there exists an algorithm for it that runs in $\mathcal{O}(n^{2-\varepsilon})$ for some $\varepsilon > 0$.

So clearly, we can't be solving the general min-plus convolution here, because if we can do so quickly, and the arrays R and S can be anything, then we can reduce min-plus convolution to this problem, solve the said open question, publish a paper, and become famous. This suggests that R and S aren't "arbitrary arrays"; they probably have some special properties that we can exploit. Let's look for them.

Remember how R and S were constructed? They are prefix sums of the arrays $[a+b, 2a+b, \ldots]$ and $[s_1, s_2, \ldots]$. This already tells us at least that R and S must be increasing, so they already can't be "arbitrary arrays". In fact, we can say more: the two arrays $[a+b, 2a+b, \ldots]$ and $[s_1, s_2, \ldots]$ are **increasing** as well, so not only are R and S increasing, but their difference arrays are increasing as well. In other words, R and S are **convex** arrays. The term "convex" comes from the fact that if you plot the values of R (or S) as a line graph, then the resulting shape is convex. (See Figure 1.)

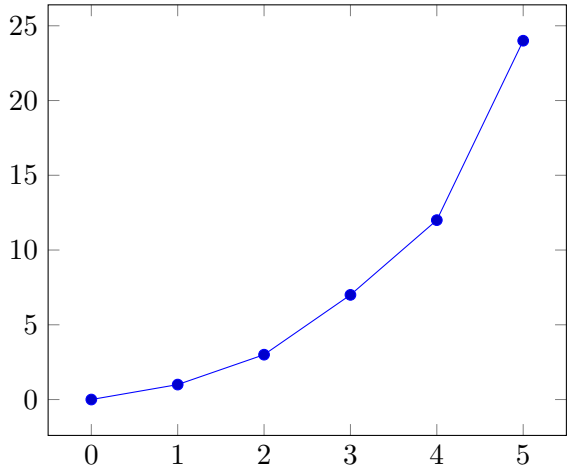


Figure 1: Illustration of convexity.

Note that convex arrays don't have to be increasing. For example, $[9, 4, 1, 0, 1, 4, 9]$ is convex too. R and S just happen to be increasing in addition to being convex.

Being convex helps us a lot! To begin with, consider a fixed k, and suppose we're trying to compute $A[k]$. Our expression for $A[k]$ says that it's the smallest among the following values:

$$\begin{aligned} R[0] + S[k] \\ R[1] + S[k-1] \\ R[2] + S[k-2] \\ \vdots \\ R[k-1] + S[1] \\ R[k] + S[0]. \end{aligned}$$

[1] Some search results say that our current best algorithms run in $\mathcal{O}\left(n^2 \frac{(\log\log n)^3}{(\log n)^2}\right)$ time, which is only slightly better than $\mathcal{O}(n^2)$, and looks quite scary to be honest. It probably even has a huge constant factor.

Let's now look at the differences between consecutive values of this sequence:

$$\begin{aligned}
&(R[1] - R[0]) + (S[k-1] - S[k]) \\
&(R[2] - R[1]) + (S[k-2] - S[k-1]) \\
&(R[3] - R[2]) + (S[k-3] - S[k-2]) \\
&\vdots \\
&(R[k] - R[k-1]) + (S[0] - S[1]]).
\end{aligned}$$

Recall that R and S are convex, which means that the first addends,

$$(R[1] - R[0], R[2] - R[1], R[3] - R[2], \ldots)$$

form an increasing sequence, and the second addends

$$(S[k-1] - S[k], S[k-2] - S[k-1], S[k-3] - S[k-2], \ldots)$$

also form an increasing sequence. (Verify the latter!) But this means that their "sum" is also increasing, which means that the sequence

$$(R[0] + S[k], R[1] + S[k-1], R[2] + S[k-2], \ldots),$$

i.e., their running sum, is also **convex**! Thus, $A[k]$ is the smallest value in a convex sequence.

Finding the minimum of any convex sequence is easy. There are two straightforward ways. One is to do "ternary search". Another is to simply binary search the difference array to look for the place where it switches signs (from negative to nonnegative). Both of these run in $\mathcal{O}(\log n)$, which means we can compute a single $A[k]$ in $\mathcal{O}(\log n)$ time, and we can compute the whole array A—and thus solve Problem 1.1—in $\mathcal{O}(n \log n)$ time!

Exercise 1.1. Why can't we just say that the minimum of a convex array is either its first element or last element? I mean, that's true for R and S!

We can actually do better—we can solve it in $\mathcal{O}(n)$ time! I'll leave it as an exercise.

Exercise 1.2. Suppose that

$$A[k] = R[x] + S[y]$$

for some $x + y = k$. Prove that

$$A[k+1] = \min(R[x+1] + S[y], R[x] + S[y+1]).$$

Use this to solve Problem 1.1 in $\mathcal{O}(n)$ time (excluding the sorting).

The above problem—and solution—illustrates the idea behind this module. The module is called **DP optimization**, and it's all about optimizing standard techniques (mostly DP) by exploiting nontrivial observations about the values being operated on.

In other words, the general theme is: "special properties may mean some optimizations are possible."

Note: Don't just memorize the techniques! Memorizing is one thing, but it's more important to understand their essence, and grasp the kind of thinking needed to come up with them, so you may be able to extend their uses to more novel problems. (Keep in mind that IOI loves novel ideas.)

2 Convexity

Our first trick is the so-called "convex hull trick". We will explain how its basic version works with a motivating problem.

Problem 2.1. There are n houses, represented as points on the plane. The ith house has coordinates (x_i, y_i). Your goal is to destroy all houses by firing laser beams.

A laser beam destroys all houses contained inside (or on the boundary of) a centered axis-aligned rectangle, and firing a laser beam incurs a cost equal to the rectangle's area. A **centered axis-aligned rectangle** is a rectangle with corners (x, y), $(x, -y)$, $(-x, -y)$ and $(-x, y)$ for some nonnegative x and y. (Such a rectangle is degenerate if $x = 0$ or $y = 0$. Degenerate rectangles are allowed and are considered to have area 0.) You may choose a different x and y for each firing, and there's no limit to the number of laser beams you may fire.

What is the minimum cost to destroy all houses?

Try solving it before moving on.

Here are easy observations:

- Any laser beam that destroys (x_i, y_i) also destroys all of $(\pm x_i, \pm y_i)$. Thus, we can replace (x_i, y_i) by $(|x_i|, |y_i|)$, and assume that all coordinates are nonnegative.
- We can discard all houses where $x_i = 0$ or $y_i = 0$ since such houses can be destroyed at 0 cost using degenerate rectangles. Therefore, we can further assume that all coordinates are positive.

Now, suppose we want to destroy some subset S of houses with *one* laser beam. What is the minimum cost to do so? It should be clear that we need to choose $x = \max_{i \in S} x_i$ and $y = \max_{i \in S} y_i$ for our rectangle, which incurs a cost of $4xy$. Using this observation, we can now do bitmask DP: our state will be the set of remaining houses to destroy, and the transition involves iterating through all subsets of that set and destroying that set with one firing. This yields an $\mathcal{O}(3^n)$ solution, but we want something faster.

The theme of this module is to look for nontrivial observations among the values. We've already partially done that above, by reducing to the case where all coordinates are positive, which simplified the logic in the previous paragraph (otherwise we'd have had to sprinkle absolute values everywhere). In fact, we can do more with the following *nontrivial insight*: Notice that we can also discard a house (x_i, y_i) if there's another house (x_j, y_j) such that $x_i \le x_j$ and $y_i \le y_j$ (i.e., northeast of it), since any laser beam destroying house j also destroys house i, and we have to destroy house j at some point anyway. Therefore, we can assume that there's no house northeast (or southwest) of any house.

But now, try to draw stuff on paper to see what the remaining points look like! Since every point can only be northwest or southeast of any house, it means that the points form some sort of "ladder" going down-right! In other words, the remaining points must be of the form $\{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$ where

$$0 < x_1 < x_2 < \ldots < x_k$$

and

$$y_1 > y_2 > \ldots > y_k > 0.$$

This is the sort of nontrivial structure we can exploit!

How can we exploit this? Well, first of all, for any subset S, it becomes easy to compute $x = \max_{i \in S} x_i$ and $y = \max_{i \in S} y_i$. Since x_i is increasing and y_i is decreasing, we simply have

$x = x_v$ and $y = y_u$ where $u = \min S$ and $v = \max S$. Furthermore, since x_i is increasing and y_i is decreasing, any laser beam that destroys all houses in the set S must also destroy all houses between u and v. Therefore, the only subsets S we need to consider are those consisting of consecutive points!

This now leads to the following DP recurrence: let $opt(v)$ be the optimal cost of destroying all houses $(x_1, y_1), \ldots, (x_v, y_v)$. Then we simply have

$$opt(v) = \min_{1 \le u \le v} opt(u-1) + 4x_v y_u,$$

where $4x_v y_u$ represents the cost of destroying all houses from u to v in one beam.

The base case is $opt(0) = 0$. This DP solution clearly runs in $\mathcal{O}(n^2)$ time, and we've once again seen our theme at work: finding nontrivial properties mean that certain optimizations are possible.

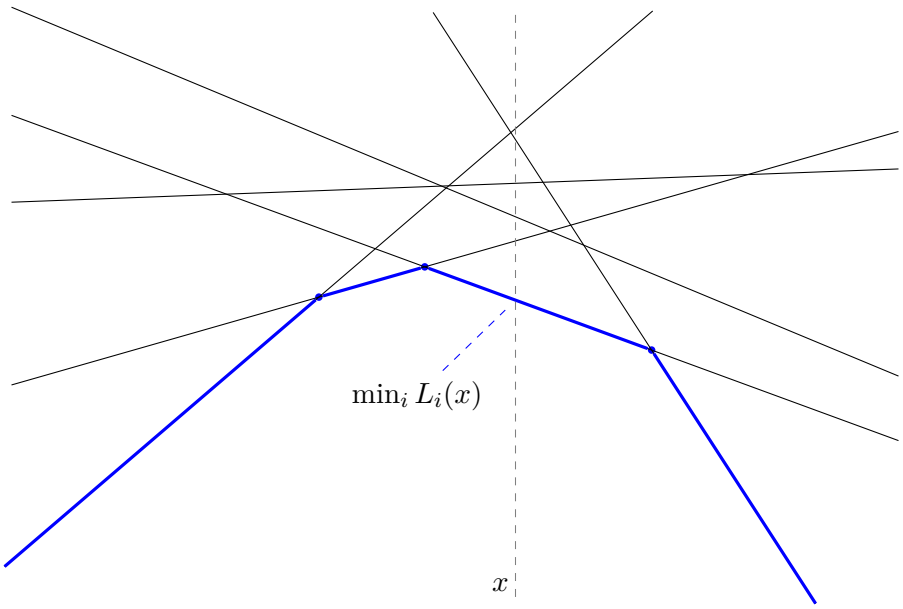
2.1 The convex hull trick

However, this isn't still quite the "convex hull trick"—it's just normal DP. The convex hull trick involves exploiting yet another nontrivial property, which has something to do with the structure of our recurrence for $opt(v)$ above.

Consider the function $x \mapsto ax + b$. Its graph is clearly a line. Now, suppose we have a bunch of these line functions, say $L_1, L_2, \ldots L_m$, and we want to find the minimum value among them at a single point, x. In other words, we want to find the value

$$\min_{1 \le i \le m} L_i(x).$$

It turns out that, after some preprocessing, we can do this faster than $\mathcal{O}(m)$! The key is to remember that the graphs of the L_is are lines. Now, try drawing a bunch of lines on the plane, and consider a fixed x. Which line gives the minimum value for that x? It's clearly a line at the "lower envelope"!



So if we have a bunch of lines, we can preprocess them by computing this "lower envelope", i.e., by determining the endpoints of the different "pieces". (The envelope is a piecewise-linear

function.) Then we can find the minimum value by simply evaluating at the appropriate piece! This can be done in logarithmic time by binary searching to find the appropriate piece.

To compute the lower envelope, we may use the fact that the slopes of the pieces are decreasing as we go from left to right. So, we can simply sort the lines in decreasing order of slopes, and try to update the envelope by adding each "line" one by one. Since the new line has a smaller slope than all existing pieces, we only need to drop the last few pieces of the envelope to update it. Using a stack, this can be done in linear time (excluding sorting by slope).

Now, how does this help in optimizing the computation of $opt(v)$? Well, notice that our formula for $opt(v)$ is basically an evaluation on the lower envelope! To see this, let L_u be the line function $x \mapsto opt(u-1) + 4xy_u$. Then we can rewrite our recurrence for $opt(v)$ as

$$opt(v) = \min_{1 \le u \le v} L_u(x_v),$$

which is exactly of the form discussed above! Therefore, we only need $\mathcal{O}(\log n)$ time to evaluate $opt(v)$ after processing the lines L_u, which means we have a solution that runs in $\mathcal{O}(n \log n)$ time!

...well, not quite. Notice that $opt(u-1)$ appears in L_u. In other words, we can only add L_u to our envelope after we've been able to compute $opt(u-1)$. But this is no problem: because we're computing $opt(v)$ in increasing order, we've already computed $opt(u-1)$ for $1 \le u \le v$, and after computing $opt(v)$, we can just add the line L_{v+1} to our envelope! By doing the construction of the envelope (with our stack procedure) interleaved with this DP computation, we are able to solve the whole problem in $\mathcal{O}(n \log n)$ time!

Exercise 2.1. By exploiting the fact that the x_i are increasing, show how to remove the need for binary searching to find the right piece, and thus allow us to be able to solve the problem in $\mathcal{O}(n)$ time (excluding the initial sorting of the points).

The idea of processing the lower (or upper) envelope to answer certain "min" or "max" queries quickly is called the **convex hull trick**.[2] This usually works when the value you're minimizing/maximizing on looks like a bunch of lines evaluated at a single point.

2.2 Dynamic envelope, and the poor man's convex hull

The previous section describes the basic application of the convex hull trick. It's "basic" because the lines are naturally added in decreasing order, and we're evaluating on the envelope in increasing order of x. In other problems, it may be trickier to apply, and you need a more dynamic structure. In general, you need a data structure that handles the following operations on the lower envelope quickly:[3]

- **Update.** Update the envelope by adding a new line function $L : x \mapsto ax + b$.
- **Query.** Given x, evaluate the lower envelope at x.

[2] You're probably asking, "where's the convex hull?" The reason for naming the trick the "convex hull trick" is that computing the lower/upper envelope of the line functions $x \mapsto a_i x + b_i$ is more-or-less equivalent to computing the convex hull of the points (a_i, b_i). I find this surprising! If you're curious to know about the details, ask us in Discord!

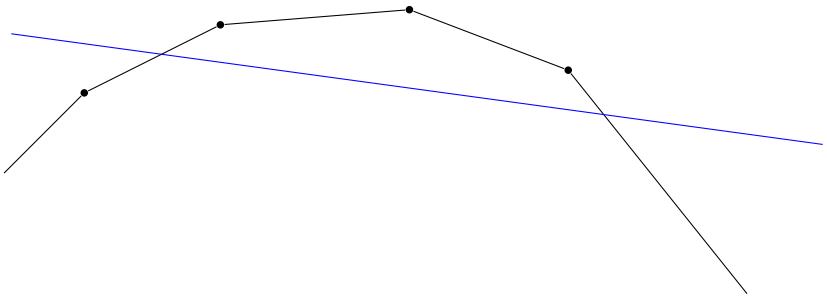
[3] There's a corresponding data structure for the upper envelope with similar details. Alternatively, notice that the upper envelope of line functions $L_1, L_2, \ldots, L_m$ simply corresponds to the lower envelope of the line functions $-L_1, -L_2, \ldots, -L_m$.

Still, in other problems, you need extra operations, such as range sums along the hull.

In this section, my goal is to describe one such data structure that can handle these operations, which I will call the **poor man's convex hull**. I won't describe in detail how to do the range sum operations, but I'll just note that the structure can easily be modified to perform range sums quickly.

By the way, the "poor man's convex hull" is not the usual way people implement the "convex hull trick". It is my own invention, and it simply involves a segment tree.

The idea that we will exploit is the fact that when adding a new line to the envelope, only a contiguous portion of the envelope gets updated:



Thus, we may view adding a new line as a *range update* on the envelope!

Actually, at this point, you already have the key idea behind the data structure, so if you want to try coming up with this data structure by yourself, now's the best time to do so!

Now, let's discuss the details. Suppose for simplicity that we know, beforehand, that our lower envelope will only be evaluated at the x values $x_1 < x_2 < \ldots < x_k$. (We can still handle the case where we don't know these x_i values beforehand by using techniques from the DS 3 module.) Let E be the lower envelope, and $E(x)$ be the lower envelope evaluated at x, so that $E(x) = \min_L L(x)$ across all line functions L added so far to the structure. The main idea of the *poor man's convex hull* is to build a segment tree on top of the array

$$[E(x_1), E(x_2), \ldots, E(x_k)].$$

As usual with segment trees, each node corresponds to a subsegment of this array. The ith leaf node will correspond to $E(x_i)$ and will contain the following information:

- The index i.
- The value $E(x_i)$.
- The line L_i corresponding to the value $E(x_i)$, so that $L_i(x_i) = E(x_i)$.

In general, every node corresponds to some contiguous set of indices $[i, j]$ and will contain the following information:

- The indices i and j.
- The values $E(x_i)$ and $E(x_j)$.
- The lines L_i and L_j corresponding to the values $E(x_i)$ and $E(x_j)$, so that $L_i(x_i) = E(x_i)$ and $L_j(x_j) = E(x_j)$.
- Some additional auxiliary data for us to perform lazy propagation (and possibly other range queries).

Evaluating the envelope at a single point is now straightforward: it simply involves going down the appropriate leaf and returning the value there. (Just don't forget to do the lazy propagation as you go!)

Now, we need to discuss adding a new line L. As we have seen, this corresponds to a range update. The trickiest part is to know which range to update. To do this, we remember that the slopes will always be decreasing. Therefore, the pieces of the envelope to the left of the updated range will have larger slopes than L, and the pieces to the right will have smaller slopes. Thus, we can partition the indices into one of three *types*:

1. If $E(x_i) < L(x_i)$ and L_i has a larger slope than L.
2. If $E(x_i) \geq L(x_i)$.
3. If $E(x_i) < L(x_i)$ and L_i has a smaller slope than L.

Thus, the portion we need to update are exactly those indices that are of type 2.

Exercise 2.2. Show that the types are nondecreasing from $i = 1$ to $i = k$.

Thus, all we need to do is to find the transition from type 1 to type 2, and the transition from type 2 to type 3. We can do this as we go down the segment tree; since L_i and L_j are stored on each node, the types can be computed as we go!

I'll leave the rest of the details to you, and just note that each update can be done in $\mathcal{O}(\log k)$ time.

As you can see, this structure is basically just a fancified segment tree. It's one of the reasons I call it the "poor man's convex hull"—it's not quite as fancy as the so-called Li Chao Tree that other people use more often. Another reason for calling it "poor" is that it usually has a poor constant factor; in some problems with very tight bounds, it may be too slow to pass.

If you want to to learn more about this Li Chao Tree thing, you can read about it here: https://cp-algorithms.com/geometry/convex_hull_trick.html .

By the way, to simplify the implementation, we can assume that initially, the lower envelope consists of a single "line" L_{high}, where $L_{\text{high}} = 0x + \infty$, where "$\infty$" represents a very large number, so evaluating this line at any x will always be larger than in any other line we'll ever consider. This means that we can initialize the whole structure with this line, and initially, we have $E(x_i) = L_{\text{high}}(x_i) = \infty$.

2.3 A more general case

The convex hull DP optimization generally applies if our transition formula involves optimizing over lines, i.e., we can write it as

$$opt(j) = \min_{1 \leq i \leq j} (\alpha_i + \beta_i x_j)$$

where α_i and β_i are allowed to depend on $opt(i-1)$ (or maybe $opt(i-2)$, etc.). Here, we need to dynamically maintain the lower envelope of the lines

$$L_i(x) = \alpha_i + \beta_i x.$$

A common special case, as we've seen above, is

$$opt(j) = \min_{1 \leq i \leq j} (opt(i-1) + \beta_i x_j).$$

This generally arises from problems where you're asked to "partition" n things. (More on this later.)

In the general case, we have no assurance that the slopes β_i are monotonic, so we may need something like the poor man's convex hull.

Actually, there's a slightly more "general" form. I'll illustrate it with an example. Suppose our recurrence is

$$opt(j) = \min_{1 \le i \le j} \left(opt(i-1) + \beta_i x_j + x_j^2 \right).$$

There's a quadratic term this time, so we seem to be minimizing on *parabolas* instead of lines. Do we need envelopes of parabolas now??

No! Notice that x_j^2 doesn't depend on i, so we can just take it out:

$$opt(j) = \min_{1 \le i \le j} \left(opt(i-1) + \beta_i x_j \right) + x_j^2.$$

And now, we see that we're still minimizing on lines after all!

More generally, for something like

$$opt(j) = \min_{1 \le i \le j} \left(\alpha_i + \beta_i x_j + y_j \right),$$

you can just take y_j out. My advice to help you notice such trickery better is just to separate the terms into the ones that depend only i (such as α_i), those that depend only on j (such as y_j), and those that depend on both (such as $\beta_i x_j$). The only thing we need for the convex hull trick is that the term that depends on both can be factored into two terms, one depending on i (i.e., β_i) and another depending on j (i.e., x_j).

This trick generally works in problems where you're asked to partition stuff. A fairly general kind of "partitioning" problem (which may also be a subproblem of a bigger problem) looks like this:

Problem 2.2. Given a cost function C an integer n, partition $[0, n]$ into some number of intervals

$$[i_0, i_1], [i_1, i_2], \ldots, [i_{k-1}, i_k]$$

such that:

- $0 = i_0 < i_1 < \ldots < i_k = n$.
- The following total cost is minimized: $C(i_0, i_1) + C(i_1, i_2) + \ldots + C(i_{k-1}, i_k)$.

The laser beam problem above (Problem 2.1) is of this form (after some preprocessing).

Assuming C can be computed in $\mathcal{O}(1)$ (if not, you need to adjust the time complexities a bit), this can generally be solved by a recurrence like one we had before:

$$opt(y) = \min_{0 \le x < y} \left(opt(x) + C(x, y) \right).$$

This is just normal DP and can always be done in $\mathcal{O}(n^2)$, but using this module's theme, if we can find special properties of C, then we may be able to do it faster. For example, if C is of the form $C(x, y) = a(x)b(y) + c(x) + d(y)$, then we can use the convex hull trick to solve it in $\mathcal{O}(n \log n)$. Generally, we want the only term in C that's dependent on both x and y to look like $\alpha f(x) g(y)$.

Finally, we've mostly used our data structure (poor man's convex hull) to optimize DP, but of course, its uses don't stop at DP. It's just a useful *data structure*! Anytime you require dynamically updating a lower/upper envelope of lines, you can use it, even if you're not doing DP.

3 Monotonicity

Let's move on to techniques that somehow exploit certain kinds of *monotonicity* properties. Two techniques will be presented here.

3.1 A motivating problem

We'll start with the first technique. As before, we'll motivate it with a non-DP problem. Although the technique is often applied to DP problems, the technique actually has two parts, one of which is applicable more generally and isn't necessarily tied to DP. Thus, we will focus heavily on understanding that aspect of the technique in isolation, so that we can focus on the heart of the technique without any confounding details. Once we understand the technique by itself, we will see that applying it to optimizing a DP recurrence will be straightforward.

The following is an abridged version of the problem **Alienmons**, written by Cisco for Algolympics Finals 2022:

Problem 3.1 (Alienmons)**.** You have been conscripted into the Survey Corps, and your job is to capture Alienmons for the researchers.

There are two kinds of Alienmons that we can capture: Red and Blue. Each (Red and Blue) has an independent "Research Level", which starts at Level 0 and can go as high as Level n; your Research Level of some color increases by catching and submitting Alienmons of that color. In particular, you are given an array $r_1, r_2, r_3, \ldots, r_n$, where r_i describes the number of Red Alienmons that you need to submit in order to go from Red Research Level $i-1$ to Level i; another array $b_1, b_2, b_3, \ldots, b_n$ describes the similar requirements for advancing in Blue Research Levels. Your "Total Research Level" is equal to your Red Research Level plus your Blue Research Level.

As an additional restriction, the research team needs more and more Red Alienmons to understand them sufficiently, so it is always the case that $r_1 \le r_2 \le r_3 \le \cdots \le r_n$ (but no such restriction is guaranteed for b).

For each j from 1 to $2n$, answer the following question: Find the minimum number of Alienmons that I need to catch in order to go from Total Research Level 0 to Total Research Level j. Each question should be considered and answered independently.

As usual, you should try it first before proceeding.

3.1.1 A quadratic-time solution

If we allow $\mathcal{O}(n^2)$-time solutions, then the problem is pretty straightforward. Let $B[i]$ be the total cost to go from Blue Research Level 0 to i, so $B[i]$ is just $b_1 + b_2 + \cdots + b_i$. We similarly define $R[i]$ for Red Research Levels, where $R[i] = r_1 + r_2 + \cdots + r_i$. For some Total Research Level k, let's suppose we achieve it by reaching Blue Research Level x, and Red Research Level $k-x$; this has a cost of $B[x] + R[k-x]$. The minimum number of Alienmons needed to reach Total Research Level k is the minimum possible value of the prior expression, over all x. Denoting the answer by $A[k]$, we therefore have

$$A[k] = \min_{0 \le x \le k} \left(B[x] + R[k-x] \right).$$

Hey, this is one of those min-plus convolutions again. It's even more blatant this time! There's a slight hiccup though—this min-plus convolution runs into out-of-bounds issues when $k > n$.

But you should find it easy to fix. I can think of three ways:

- In the convolution, limit x further so that both $x \leq n$ and $k - x \leq n$ hold.
- Pad B and R with extra values that won't affect the answer, e.g., ∞ (or some really large number).
- Only do the convolution for $k \leq n$, and do a separate (mirror-image) convolution for $k > n$.

Any of these is fine, though the last one will make our explanation cleanest. (Though it may be tricky to implement.) So in what follows, we will only compute $A[k]$ for $0 \leq k \leq n$.

Anyway, if we directly compute the min-plus convolution, then it will take $\mathcal{O}(n^2)$ time.

3.1.2 Towards faster solutions

Okay, we got our quadratic-time solution. Can we do better? Unfortunately, we can't just apply our earlier $\mathcal{O}(n)$ solution since B and R are not necessarily convex! (Actually, R is convex, but we'll get to that.)

Well... let's investigate some stuff!

Here's one idea that might come to mind. The answer for some k is found by choosing the x that minimizes the given expression. But what are the actual values of x that achieve this minimum? Keeping the module's theme in mind (and using the problem-solving technique "try concrete cases first"), let's look at those values, hoping to find some patterns or properties that we can exploit. Let's check for the sample input:

$$r = [1, 1, 2, 2]$$
$$b = [1, 3, 1, 4].$$

For some k, let x_k be equal to the value of x which minimizes the expression $B[x] + R[k - x]$. In case of ties, we arbitrarily always pick the largest x.

k	0	1	2	3	4	5	6	7	8
x_k	0	0	0	1	1	1	3	3	4

How curious. It seems like x_k is nondecreasing! Let's try it again with another random small test case:

$$r = [1, 2, 2, 5, 5, 7, 7, 8]$$
$$b = [2, 3, 8, 8, 5, 6, 9, 7].$$

The minimums here for each k are achieved with the following values of x:

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_k	0	0	1	1	1	2	2	2	2	2	5	6	6	6	6	8	8

It really does seem like $0 \leq x_0 \leq x_1 \leq x_2 \leq x_3 \leq \ldots$! You can try doing it for more random cases, and you'll notice that it always seems to be true.

This definitely doesn't always happen with all DP in general, so there must be something special about the setup of this problem. We'll try to prove it later, but for now, assuming that x_k is indeed nondecreasing, so what?

Here's what. When looking for x_k, we don't have to explore the entire search space, because we can pick up from where we left off.

More precisely, we don't have to begin our search for x_k from 0; since $x_{k-1} \le x_k$, we can just start the search from x_{k-1}.

$$A[k] = \min_{x_{k-1} \le x \le k} (B[x] + R[k-x])$$
$$x_k = (\text{the } x \text{ maximizing the above, so that } A[k] = B[x_k] + R[k-x_k])$$

Unfortunately, computing using this recurrence for all $k \le n$ still gives $\mathcal{O}(n^2)$ in the worst case. We only have to check all values of x from x_{k-1} to k, but you can imagine that there are test cases where x_k stays low, even as k grows large.

The answer is that we need to leverage the nondecreasing nature of x_k more creatively than just the $k-1 \to k$ transition. For this, we use another observation: we don't have to compute $A[k]$ in order of k from 0 to n. We can answer it in any order that is most convenient for us!

This is nice because if you think about it, the property

$$0 \le x_0 \le x_1 \le x_2 \le x_3 \le \dots$$

allows us to introduce a lower bound *and* an upper bound on x. By doing it increasing k, we're depriving ourselves of the upper bound part.

Suppose we want to compute some $A[k]$, and we have already computed $A[\ell]$ and $A[r]$ for ℓ and r such that $\ell \le k \le r$. That means $x_\ell \le x_k \le x_r$, which means that the optimal value of x is bounded between x_ℓ and x_r.

$$A[k] = \min_{x_\ell \le x \le x_r} (B[x] + R[k-x])$$
$$x_k = (\text{the } x \text{ maximizing the above})$$

All that remains is for us to decide the optimal order in which to evaluate each $A[k]$ so that the total amount of work being done is "balanced". And well, we are bounding things from above and below—wouldn't it be natural to try to split the work in half at each step? Here comes the *divide and conquer* part!

3.1.3 The divide and conquer optimization

We are going to tackle this piece by piece. First, we are going to introduce the divide and conquer structure by itself. Once that is established, we will modify the function to accommodate our insight that x_k is nondecreasing.

We'll first create the A array, initially containing garbage. We will then fill the answers $A[0], \dots, A[n]$ in, but not in order. For simplicity, we'll assume that our functions will have access to this array.[4] Now, define `solve_range(l, r)` to be a function which executes the subroutine:

> "Populate $A[\ell], A[\ell+1], \dots, A[r]$ with the correct values."

We populate the entire array by calling `solve_range(0, n)`. (Remember to handle $n < k \le 2n$ in a similar, mirrored way.)

We can formulate `solve_range` as follows:

- Evaluate an arbitrary index k first, with $\ell \le k \le r$), using our formula for $A[k]$.

[4] For example, via globals, or better, as a `static` variable of some function, or a regular local variable but captured by lambda functions.

- Now that $A[k]$ has been computed, we can solve for the remaining indices in $[\ell, r]$ by simply recursively calling `solve_range(l, k-1)` and `solve_range(k+1, r)`.
- The base case is when $\ell > r$. (What do we do in that case?)

In pseudocode:

```
def solve_range(l, r):
    # base case
    if l > r: return

    # choose k
    k = (choose some index between l and r)

    # compute A[k]
    A[k] = min(B[x] + R[k-x] for x from 0 to k)

    # solve the rest recursively
    solve_range(l, k-1)
    solve_range(k+1, r)
```

Nothing smart is happening here. We're just populating A, computing each term in the straightforward manner, but in this weird nonlinear order. It's still $\mathcal{O}(n^2)$.

Now, here's the heart of the optimization. We'll utilize the fact that the x_ks are nondecreasing, by limiting the range of indices x we check. We will utilize the fact that when we call `solve_range(l, r)` recursively, $A[\ell]$ and $A[r]$ have already been computed, and thus x_ℓ and x_r as well. Since $\ell \le k \le r$, we can limit our search for x_k to the range $x_\ell \le x_k \le x_r$.

To do this, we augment `solve_range` with additional arguments `xl` and `xr`, and redefine `solve_range(l, r, xl, xr)` to be a function which executes the subroutine:

> "Populate $A[\ell], A[\ell+1], \ldots, A[r]$ with the correct values, assuming that for each $k = \ell, \ldots, r$, the optimal splitting point x_k lies between x_ℓ and x_r."

We populate the entire array by calling `solve_range(0, n, 0, n)`.

In pseudocode again:

```
def solve_range(l, r, xl, xr):
    # base case
    if l > r: return

    # choose k
    k = (choose some index between l and r)

    # compute A[k] and xk
    A[k] = min(B[x] + R[k-x] for x = xl..xr)
    xk = (the largest x = xl..xr such that A[k] = B[x] + R[k-x])

    # solve the rest recursively
    solve_range(l, k-1, xl, xk)
    solve_range(k+1, r, xk, xr)
```

Note how x_k serves as an upper bound for all $j < k$ and as a lower bound for all $j > k$.

Exercise 3.1. This pseudocode has off-by-one issues! Can you fix it?

Hint: What happens if $x_r > k$?

What is the running time of this algorithm?

In the first call to `solve_range(0, n, 0, n)`, x goes from 0 to n, doing $\mathcal{O}(n)$ work. This calls `solve_range(0, k-1, 0, xk)` and `solve_range(k+1, n, xk, n)`. In the former call, x goes from 0 to x_k; in the latter call, x goes from x_k to n.

Note that x_k is not necessarily the midpoint of 0 to n. But the work is still "balanced" because both calls together have x going from 0 to n, so the total work of this "layer" is still $\mathcal{O}(n)$.

Similarly, the next "layer" has four recursive calls, and the total work of this layer is still $\mathcal{O}(n)$:

1. 0 to $x_{k'}$
2. $x_{k'}$ to x_k
3. x_k to $x_{k''}$
4. $x_{k''}$ to n

Here, k' and k'' are the `k`s we chose for the left and right recursive calls, respectively. Again, $x_{k'}$ and $x_{k''}$ are not necessarily midpoints, but the work is still balanced because in this layer, x goes from 0 to n.

In general, this argument shows that the total work done by a given layer of recursive calls is bounded by $\mathcal{O}(n)$. Therefore, the total work is $\mathcal{O}(n \times (\text{number of layers}))$.

Now, up to this point we still haven't specified how to choose k—all we said is that it's between ℓ and r—so if we choose it badly, then we may end up with $\Theta(n)$ layers and a running time of $\mathcal{O}(n^2)$. This is all the more worrying since as we've seen above, we can't necessarily ensure that the interval $[x_\ell, x_r]$ halves in size at every recursion layer. However, notice that the base case condition is "$\ell > r$", *not* "$x_\ell > x_r$", so the number of layers isn't determined by $[x_\ell, x_r]$ but by $[\ell, r]$. And we have control over the latter! So now, here comes the *divide and conquer* bit: we can *always* halve the interval $[\ell, r]$ by choosing k to be the **midpoint** of $[\ell, r]$. Doing it this way, we'll quickly reach the base case $\ell > r$ in only $\mathcal{O}(\log n)$ layers. Therefore, our solution runs in $\mathcal{O}(n \log n)$ time. Neat!

3.1.4 Proving monotonicity

Actually, we're not done yet. We still haven't proved that x_k is nondecreasing! "Proof by examples" is not a valid proof technique, so we should prove it rigorously.

As explained previously, we can't prove that x_k is nondecreasing for *general* arrays B and R since no one knows yet how to do min-plus convolutions in $\mathcal{O}(n^{2-\varepsilon})$ time. Thus, we have to rely on the fact that R **is convex**. Note that B is not necessarily convex—if it were, we can just use our $\mathcal{O}(n)$ solution from earlier.

It's worth going through the proof because it's useful scaffolding for when you want to prove that x_k is nondecreasing in other contexts.

Theorem 3.2. If $k \le k'$, then $x_k \le x_{k'}$. Equivalently,

$$x_0 \le x_1 \le \ldots \le x_n.$$

Proof. We will fix k and k', and show that $x_k \le x_{k'}$.

Recall that x_k is the value of x that minimizes $B[x] + R[k - x]$. Because this x minimizes this expression, for any other x, **here's what we currently know**:

$$B[x_k] + R[k - x_k] \le B[x] + R[k - x]. \tag{1}$$

Now, consider k' such that $k \le k'$. We want to show that $x_{k'}$ is at least as large as x_k.

Recall that $x_{k'}$ is the value of x that minimizes $B[x] + R[k' - x]$. So if we want to say that $x_{k'} \ge x_k$, i.e., that the optimal splitting point is at least x_k, then we are claiming that the splitting points less than x_k cannot be optimal. For this, it's sufficient to show that x_k is a better splitting point than any of them. Formally, for any $x < x_k$, **this is what we want to be able to show**:

$$B[x_k] + R[k' - x_k] \le B[x] + R[k' - x]. \tag{2}$$

Starting at *what we know* (1), how do we arrive at *what we want to show* (2)? Let's work backwards. If we "subtract the two inequalities" (which, formally, you can't do, but this is just scratchwork), we get this inequality that **would be nice**:

$$R[k' - x_k] - R[k - x_k] \le R[k' - x] - R[k - x]. \tag{3}$$

If we can show that this is true, then our proof would be complete: we can add it to both sides of the what-we-know inequality (1), and that would yield the inequality that we want to show (2).

So is this actually true? Yes! Let's expand each side of the inequality in terms of sums of r:

$$r_{k-x_k+1} + r_{k-x_k+2} + \cdots + r_{k'-x_k} \le r_{k-x+1} + r_{k-x+2} + \cdots + r_{k'-x}.$$

Both sides of the inequality have exactly $k' - k$ terms (you can check this). Also, we can naturally pair them up in the following way:

$$\begin{aligned} r_{k-x_k+1} &\le r_{k-x+1} \\ r_{k-x_k+2} &\le r_{k-x+2} \\ &\vdots \\ r_{k'-x_k} &\le r_{k'-x}. \end{aligned}$$

Each of these individual inequalities is true because $x < x_k$ and r was given to be nondecreasing in the problem statement. Sum them all up to recover the inequality that we would like.

The actual inequality that we wanted to show (2) then follows from (1) and (3). For $x < x_k$:

$$\begin{gathered} B[x_k] + R[k - x_k] \le B[x] + R[k - x] \\ + \left(R[k' - x_k] - R[k - x_k] \le R[k' - x] - R[k - x]\right) \\ \downarrow \\ B[x_k] + R[k' - x_k] \le B[x] + R[k' - x] \end{gathered}$$

This completes the proof. □

Exercise 3.3. Where in this proof did we use the fact that R is convex?

By the way, you don't have to explicitly mention the quadrangle inequality to prove that x_k is nondecreasing. Often, you can simply prove it directly. (Your proof might implicitly use the quadrangle inequality, but that's okay.) Here's one way to prove it directly:

Proof. We wish to prove that $x_k \le x_{k+1}$, where x_k is the x that minimizes the expression

$$B[x] + R[k - x].$$

Enumerating these values, we get

$$\begin{aligned}
B[0] + R[k] \\
B[1] + R[k-1] \\
B[2] + R[k-2] \\
\vdots \\
B[x_k] + R[k - x_k] \\
\vdots \\
B[k-1] + R[1] \\
B[k] + R[0].
\end{aligned}$$

The row corresponding to x_k represents the minimum, so all its other values are larger (or equal). In particular, all rows before x_k are larger than or equal to $R[x_k] + B[k - x_k]$.

Now, consider what happens at $k + 1$. We want to show that the minimum value of

$$B[x] + R[k + 1 - x]$$

lies at x_k or beyond. Enumerating these values, we get

$$\begin{aligned}
B[0] + R[k+1] \\
B[1] + R[k] \\
B[2] + R[k-1] \\
\vdots \\
B[x_k] + R[k - x_k + 1] \\
\vdots \\
B[k] + R[1] \\
B[k+1] + R[0].
\end{aligned}$$

However, recall that $R[i+1] = R[i] + r_{i+1}$. Substituting this, we get

$$\begin{aligned}
&(B[0] + R[k]) && + r_{k+1} \\
&(B[1] + R[k-1]) && + r_k \\
&(B[2] + R[k]) && + r_{k-1} \\
&\vdots \\
&(B[x_k] + R[k - x_k]) && + r_{k-x_k+1} \\
&\vdots \\
&(B[k] + R[0]) && + r_1 \\
&B[k+1] + R[0] && .
\end{aligned}$$

But now, note that these are basically the same values as before (for k), just with

$$r_{k+1}, r_k, r_{k-1}, \ldots$$

added in! Furthermore, this sequence is decreasing, so later values get increased by smaller amounts. Since $B[x_k] + R[k - x_k]$ is smaller than the values before it, it stays smaller after the increase, which means the optimal splitting point cannot be before x_k. This proves that $x_k \le x_{k+1}$. □

The point of this alternate proof is so that you may see that there may be multiple ways of proving it, and it's not impossible for you to concoct one during contest.

By the way, these proofs have a bit of sloppiness: there may be several optimal splitting points, and we didn't check that x_k is the *largest* one among them. However, I didn't include it because it will just clutter the proof. Feel free to make this more rigorous if you want.

3.2 The divide and conquer optimization, generalized

Let's summarize and generalize what we've learned.

Suppose we have a function $A[y]$ of the following form:

$$A[y] = \operatorname*{opt}_{0 \le x \le y} (f(x) + C(x, y)) \tag{4}$$

where f and C are any functions that can be "evaluated quickly" (say, $\mathcal{O}(1)$) and "opt" is just either "min" or "max".

Let x_y be the argument x which optimizes $A[y]$. **If x_y is nondecreasing, then we can evaluate $A[y]$ for all y from 0 to n in just $\mathcal{O}(n \lg n)$.**

The technique is exactly the same as the one detailed above. There, $f(x) = B[x]$ and $C(x, y) = R[y - x]$. And since B and R are just arbitrary arrays given in the input, you can see that this technique should work for any functions f and C. This completes the proof!

Actually, not so fast. We do have one requirement—we must show that x_y is nondecreasing. Typically, this part is where we would start invoking problem-specific properties of the functions f and C in order to complete the proof.

But actually, the proof that x_y is nondecreasing can be generalized as well. If you follow the steps in our proof in Alienmons, you'll note that it boiled down to showing the following property, and didn't invoke any specific properties of any function until here:

$$R[k' - x_y] - R[k - x_y] \le R[k' - x] - R[k - x].$$

In our case, we proved it by appealing to the fact that r was nondecreasing, which made R a convex function. (*Side note:* If you know the calculus definition of convexity, you can see how r being nondecreasing is a discrete analog to R having a nonnegative second derivative everywhere.)

3.3 The quadrangle inequality

Let's express this in terms of our general functions—actually, only C was important. Suppose we have any $L \le \ell \le r \le R$. Then, if the following inequality holds,

$$C(\ell, R) - C(\ell, r) \preceq C(L, R) - C(L, r)$$

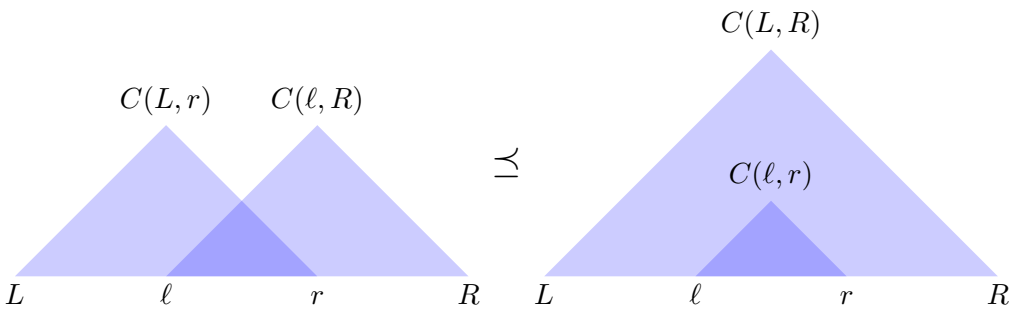
then the splitting points x_y is nondecreasing, and we can apply our Divide and Conquer Optimization. Here, $\prec$ should be read as "is more optimal than", and is just either less than or greater than, depending on if we are minimizing or maximizing. Actually, we can rearrange this property into the following form:

$$C(L, r) + C(\ell, R) \preceq C(L, R) + C(\ell, r).$$

Then, we have the well-known **quadrangle inequality**. Keep in mind that $L \le \ell \le r \le R$.

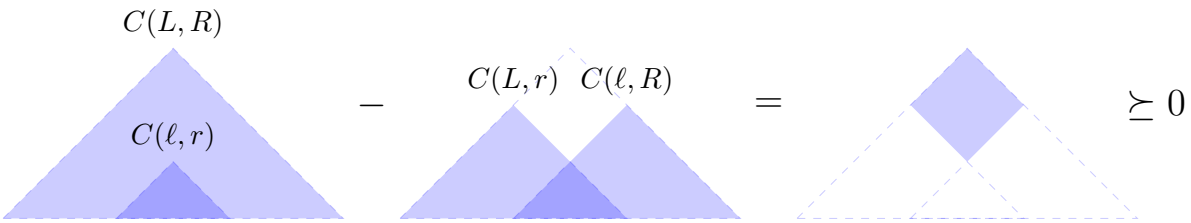
Theorem 3.4. If C satisfies the quadrangle inequality, then the x_ys will be nondecreasing.

You can visualize the quadrangle inequality as follows:



Using this picture, it could already be intuitive to you why the quadrangle inequality implies some good things about the optimal partition.

From another perspective, you could also view the quadrangle inequality as saying that the cost of every quadrangle is nonnegative, since it is equivalent to:



The quadrangle inequality is a sufficient but not necessary condition for x_y to be nondecreasing. So, in some cases, the quadrangle inequality serves as useful algebra plug-and-chug in order to mechanically verify that x_y is nondecreasing. Or maybe you may just find it more intuitive to recreate the proof from scratch. It's up to you!

Here's one way to intuitively feel why the quadrangle inequality implies that the optimal cutting points x_k are nondecreasing. (The picture above helps.) Notice that for $L \le \ell \le r \le R$, it's better to take the partitions $\{[L, r], [\ell, R]\}$ than $\{[L, R], [\ell, r]\}$. So if we consider partitioning the intervals $[1, r]$ and $[1, R]$ optimally, the quadrangle inequality biases us to choose the cutting point for $[1, R]$ later than for $[1, r]$. This isn't a proof of course, but I hope it helps.

Of course, if we're maximizing instead of minimizing, you'll have to reverse all your intuitions above. (Alternatively, negate everything so we're minimizing again.)

3.4 Divide and conquer optimization and DP

Now, I promised you DP, so let's do some DP! You'll see that it was worth the effort to frontload learning the Divide and Conquer trick in isolation. Its application to DP is so straightforward!

Consider Guardians of the Lunatics, from NOI.PH 2014 Finals. Here's an abridged version:

Problem 3.2. You are in charge of assigning guards to a prison. There are n prison cells numbered from 1 to n. Cell i houses exactly one lunatic whose *craziness level* is C_i.

Each lunatic should have exactly one guard watching over him/her. Each guard should be assigned a set of consecutive-numbered cells. You only have g guards to assign.

For the lunatic in cell i, define R_i to be the product of C_i and the number of lunatics the guard assigned to him/her is watching over. Define R to be the sum of R_is for i from 1 to n. You have to assign which lunatics each guard should watch over in order to minimize R. What is the minimum possible value of R?

Constraints

- $1 \le n \le 8000$
- $1 \le g \le 800$

Again, you should try it on your own first. You should be able to get a $\mathcal{O}(gn^2)$ DP solution.

The DP recurrence for this problem is straightforward. Let $opt(d, y)$ be the minimum R if we have d guards, and only considering the first y prisoners. Let's consider what we want to do with the rightmost guard. They monitor some contiguous segment of prisoners, and for sure we know that they have to be in charge of monitoring prisoner y (because they are the rightmost guard). So, let's consider all possible left endpoints x, so that this guard monitors prisoners $x + 1$ to y (of which there are $y - x$ of them), contributing some amount to R, and then the remaining $d - 1$ guards will have to deal with the prisoners up to x. This gives the following DP recurrence, where $P_t = R_1 + R_2 + \cdots + R_t$:

$$opt(d, y) = \min_{0 \le x \le y} \left(opt(d - 1, x) + (y - x)(P_y - P_x) \right).$$

Doing this DP takes $\mathcal{O}(gn^2)$ time, which is too slow.

One thing we notice with this recurrence is that d barely changes. Indeed, we can compute opt in increasing order of d—i.e., row by row—and discover that each row only depends on the previous one. To capture the fact that d barely changes, we can change notations a bit. Let's write $opt_d(y)$ instead of $opt(d, y)$, so our recurrence becomes

$$opt_d(y) = \min_{0 \le x \le y} \left(opt_{d-1}(x) + (y - x)(P_y - P_x) \right)$$

Hey, we've seen this form before! Can we apply the Divide and Conquer Optimization?

Indeed. Just set

$$\begin{aligned} A &= opt_d, \\ f &= opt_{d-1}, \\ C(x,y) &= (y-x)(P_y - P_x), \\ \text{opt} &= \min, \end{aligned}$$

and then we can apply (4) to compute each row in $\mathcal{O}(n \lg n)$ time!

The only thing left is to show that x_y is nondecreasing, which you can verify yourself using the quadrangle inequality on C. (It's a little bit of algebra, but it's not too bad.) All we have to do to solve this problem is apply the Divide and Conquer Optimization g times. Therefore, the overall running time is $\mathcal{O}(gn \lg n)$, which passes!

We leave the implementation of the solution as an exercise, to emphasize the fact it really isn't that different at all from Alienmons. We really are doing the exact same thing. The only difference is that your subroutine will probably be `solve_range(d, l, r, xl, xr)`—it needs to include d in the signature so that it knows which layer to populate.

3.5 A more general case

A more general version of Problem 3.2 is the following, which is a variation of our earlier general partitioning problem (Problem 2.2), but this time, the number of partitions is now bounded:

Problem 3.3. Given a cost function C and integers n and g, partition $[0, n]$ into g intervals

$$[i_0, i_1], [i_1, i_2], \ldots, [i_{g-1}, i_g]$$

such that:

- $0 = i_0 \le i_1 \le \ldots \le i_g = n$.
- The following total cost is minimized: $C(i_0, i_1) + C(i_1, i_2) + \ldots + C(i_{g-1}, i_g)$.

The DP recurrence is clearly

$$opt_d(y) = \min_{0 \le x \le y} \left(opt_{d-1}(x) + C(x, y) \right).$$

For simplicity, we assume that C can be "evaluated quickly", i.e., in $\mathcal{O}(1)$. (Again, you should adjust the running times otherwise.)

Again, using this recurrence gives a $\mathcal{O}(gn^2)$ straightforward DP. However, *if the optimal splitting points in each row are nondecreasing*, then we can repeatedly use the divide and conquer optimization in order to compute each row opt_d in $\mathcal{O}(n \log n)$ time, and thus the whole DP table in $\mathcal{O}(gn \log n)$ time.

More formally, let x_y be the argument x which optimizes $opt_d(y)$. Then we want x_y to be nondecreasing for the above technique to work. And as we've seen above, this is automatically true if the cost function C satisfies the quadrangle inequality

$$C(L, r) + C(\ell, R) \le C(L, R) + C(\ell, r)$$

for $L \le \ell \le r \le R$. (It can be confusing to remember which L or ℓ appears where, so I suggest simply remembering our visualization above involving triangles/quadrangles.)

If you forgot about the quadrangle inequality during a contest, you can still do a few things to prove that x_y is nondecreasing:

- You can rederive the proof I gave. (Just remember the key steps.)
- You can just try proving it directly, not necessarily with a similar proof as above. Quadrangle inequality is merely a shortcut—a convenient one of course—but if it's not available, you could just take the long way.
- Try a bunch of random cases, print out the x_ys, and see if they're indeed nondecreasing. (Don't just do this by eye—use `assert`s!) This isn't really a proof, but it should give you confidence that it's true; because many times, if something is false, there's usually a small counterexample where it fails.

  For example, I implemented the $\mathcal{O}(gn^2)$ DP for Guardians of the Lunatics (Problem 3.2), ran it on a couple of randomly generated cases with $g \le 10$ and $n \le 10$, and printed the x_ys in a 2D table. Here are the first three tables I got:

```
0 0 0 0 0 0 0 0 0 0 0
0 1 1 2 2 3 3 3 4 4 5
0 1 2 2 3 4 4 4 5 5 6
0 1 2 3 3 4 4 5 6 6 7
0 1 2 3 4 4 5 5 6 7 8
0 1 2 3 4 5 5 5 7 7 8
0 1 2 3 4 5 6 6 7 8 8
0 1 2 3 4 5 6 7 7 8 8
0 1 2 3 4 5 6 7 8 8 9
```

```
0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 2 2 3 3 3 4 4
0 1 2 2 2 3 4 4 4 4 6
0 1 2 3 3 3 4 4 6 6 6
0 1 2 3 4 4 4 4 6 6 6
0 1 2 3 4 5 5 6 6 6 6
0 1 2 3 4 5 6 6 6 6 9
0 1 2 3 4 5 6 7 7 7 9
0 1 2 3 4 5 6 7 8 8 9
```

```
0 0 0 0 0 0 0 0 0
0 1 1 2 2 3 3 3 4
0 1 2 2 2 4 4 4 5
0 1 2 3 3 4 4 5 6
0 1 2 3 4 4 4 5 7
0 1 2 3 4 5 5 5 7
0 1 2 3 4 5 6 6 7
0 1 2 3 4 5 6 7 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```

  As you can see, the rows do indeed seem to be monotonic. And in fact, I added `assert` statements to verify this for millions of random cases (literally!), and they're still monotonic, so if there's a counterexample, then it couldn't be found this way—maybe it's very large, or maybe you really have to think hard to come up with one, or maybe it simply doesn't exist.

3.6 Knuth optimization

There's something curious about the x_y tables we printed above. Remember that we printed them to check that the *rows* are nondecreasing. However, curiously, it seems that the *columns* are nondecreasing too! And if I put in **assert** statements in my code to check this (in addition to the rows), then sure enough, they also pass millions of test cases! This strongly suggests that it may be true—the columns are nondecreasing as well.

There should be two questions that we ask next:

1. **Why is that?**
2. **Can we exploit it?**

The second question is in line with the theme of the module, of course.

We'll try to answer both questions, starting with the second, but for now, let's answer the first question partially, and see if we can intuitively get a feel for why the columns are monotonic.

Remember that x_y represents the optimal splitting point when partitioning $[1, 2, \ldots, y]$ into d groups. Actually, we should write it as $x_{d,y}$ now, since we're now considering multiple rows at a time. Anyway, the fact that the columns are nondecreasing can be written as

$$x_{d-1,y} \leq x_{d,y}.$$

Intuitively, this is saying that if you allow yourself to have more groups, then the *optimal* rightmost group cannot suddenly increase in size! I mean, that's kinda intuitive, right? However, intuition is not proof, so we'll prove this rigorously later.

Anyway, back to the second question. Assuming that $x_{d,y}$ is really monotonic in both rows and columns, let's try to exploit it. Recall that for the Divide and Conquer optimization, we used the monotonicity of rows to give lower and upper bounds to each $x_{d,y}$ to reduce the search space to just $[x_{d,y-1}, x_{d,y+1}]$. Coupled with the fact that we can compute the elements of each row in any order, this allowed us to compute the answer in $\mathcal{O}(gn \log n)$ time.

However, if the columns are nondecreasing as well, then we have a new lower bound for $x_{d,y}$: $x_{d-1,y}$. This means we can search for $x_{d,y}$ in the interval $[x_{d-1,y}, x_{d,y+1}]$ instead. Now of course, we can only do this if we already have the values $x_{d-1,y}$ and $x_{d,y+1}$ available; that is, we've already computed $opt(d-1, y)$ and $opt(d, y+1)$. Luckily, we can make that happen:

- Since we have to compute the rows from top to bottom (i.e., increasing d), $x_{d-1,y}$ should already available.
- To make $x_{d,y+1}$ available, we must compute each row in *decreasing* order of y. This is possible since we can compute the elements of a row in any order.

This gives us the following algorithm (in pseudocode), where the 2D array `x` contains the $x_{d,y}$ values:

```
for d from 1 to g:
    for y from n to 0:  # note: decreasing!
        xl = (if d = 1 then 0 else x[d-1][y])
        xr = (if y = n then n else x[d][y+1])

        # compute opt(d,y) and x(d,y)
        opt[d][y] = min(opt[d-1][x] + C(x, y) for x = xl..xr)
```

```
8           x[d][y] = (the largest x minimizing the previous line)
```

If you don't like the if-then-else expressions for `xl` and `xr`, you can get rid of them by padding the table `x` with the value `0` on row 0 and `n` on column $n+1$.

So what's the running time of this? It's not even clear that it improves over the default $\mathcal{O}(gn^2)$ DP. After all, you might be thinking: the first row can already trigger an $\mathcal{O}(n^2)$ running time, especially if $x_{1,y}$ stays large even as y gets small—exactly the same problem we had when we were computing each row in increasing order of y.

However, it turns out that it runs faster than $\mathcal{O}(gn^2)$! Note that we only search for $x_{d,y}$ along the interval $[x_{d-1,y},\ x_{d,y+1}]$, which means that there are lots of different $x_{d,y}$ whose search intervals don't overlap at all. For example, let's note the location of $x_{4,7}$ in the table:

$$\begin{array}{cccc} \ddots & \vdots & & \\ \cdots & x_{3,7} & & \\ \cdots & x_{4,7} & x_{4,8} & \\ & & x_{5,8} & x_{5,9} & \cdots \\ & & \vdots & \vdots & \ddots \end{array}$$

Because of monotonicity in rows and columns, we have the inequalities:

$$x_{3,7} \le x_{4,7} \le x_{4,8} \le x_{5,8} \le x_{5,9}.$$

In other words, the search intervals for $x_{4,7}$ and $x_{5,8}$ are $[x_{3,7},\ x_{4,8}]$ and $[x_{4,8},\ x_{5,9}]$, and we notice that they don't overlap at all (except at endpoints)!

Generalizing this observation, we now see that the $x_{d,y}$s that lie along a diagonal of the DP table, such as

$$\begin{array}{ccccccccc} \ddots & & & & & & & & \\ & x_{4,7} & & & & & & & \\ & & x_{5,8} & & & & & & \\ & & & x_{6,9} & & & & & \\ & & & & x_{7,10} & & & & \\ & & & & & x_{8,11} & & & \\ & & & & & & \ddots & & \end{array}$$

have search intervals that don't overlap, so their combined search space is only $\mathcal{O}(n)$ in size. And since there are only $\mathcal{O}(g+n)$ diagonals in a $(g+1) \times (n+1)$ table, the algorithm only runs in $\mathcal{O}((g+n)n)$ time, indeed faster than $\mathcal{O}(gn^2)$! It passes the time limit as well.

So our earlier concern that the first row can already trigger $\mathcal{O}(n^2)$ behavior doesn't matter after all—if that happens, that just means there's less search space for the later rows, and overall work done will still be $\mathcal{O}(n^2)$.

This is called **Knuth's optimization**. It may be used when the optimal splitting point locations are monotonic in two directions. Historically, the Knuth optimization was used for a different kind of DP problem, not one of the form Problem 3.3. You'll encounter that form in the problems.

Exercise 3.5. Explain how to improve $\mathcal{O}((g+n)n)$ to just $\mathcal{O}(n^2)$ for our particular problem (Problem 3.2).

3.6.1 Proving column monotonicity

Actually, we're not done yet. We still have to prove that the columns are indeed nondecreasing.

I find this usually harder to prove than the corresponding theorem for the divide and conquer optimization. However, for partitioning problems like Guardians of the Lunatics (Problem 3.2), the proof generally goes the same way every time. We'll show this proof below. The only assumption we need is that C satisfies the quadrangle inequality.

We can state the theorem as follows:

Theorem 3.6. For any $d > 0$ and y, $x_{d-1,y} \le x_{d,y}$. Equivalently,

$$x_{0,y} \le x_{1,y} \le \ldots \le x_{g,y}.$$

Proof. Again, first let's do some scratchwork. Fix d and y.

We want to show that $x_{d,y} \ge x_{d-1,y}$, i.e., the optimal splitting point for $opt_d(y)$ is at least $x_{d-1,y}$. It is sufficient to show that $x_{d-1,y}$ is a better splitting point than any x less than it. The expression we're minimizing is "$opt_{d-1}(x) + C(x, y)$", so **here's we want to show**: for any $x < x_{d-1,y}$,

$$opt_{d-1}(x) + C(x, y) \ge opt_{d-1}(x_{d-1,y}) + C(x_{d-1,y}, y). \tag{5}$$

On the other hand, by definition, $x_{d-1,y}$ is the optimal splitting point for $opt_{d-1}(y)$, so here's **what we know**:

$$opt_{d-2}(x) + C(x, y) \ge opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y). \tag{6}$$

So if we want to deduce (5) from (6), it **would be nice** if we can prove the following ("subtract the inequalities"):

$$opt_{d-1}(x) - opt_{d-2}(x) \ge opt_{d-1}(x_{d-1,y}) - opt_{d-2}(x_{d-1,y}). \tag{7}$$

This is a sort of quadrangle inequality, but for *opt* instead of C.

Now, here's a funny trick. Notice that (7) only refers to rows $d-2$ and $d-1$. However, we're currently dealing rows $d-1$ and d. Thus, we may assume that it's already true *by induction*! (Formally, we're doing induction on d.)

So we're done! Adding (6) and (7), we get (5):

$$\begin{aligned} opt_{d-2}(x) + C(x, y) &\ge opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y) \\ + \left[opt_{d-1}(x) - opt_{d-2}(x)\right. &\ge \left. opt_{d-1}(x_{d-1,y}) - opt_{d-2}(x_{d-1,y})\right] \\ &\downarrow \\ opt_{d-1}(x) + C(x, y) &\ge opt_{d-1}(x_{d-1,y}) + C(x_{d-1,y}, y) \end{aligned}$$

for $x < x_{d-1,y}$.

...Actually, we're not done yet. We magically invoked induction, so we need to actually complete the induction! We still have to prove the quadrangle inequality on *opt*. We want to show that for $y' \le y$,

$$opt_d(y') - opt_{d-1}(y') \ge opt_d(y) - opt_{d-1}(y). \tag{8}$$

To make things easier to think about, let's make all terms have a '+' sign by transposing. So equivalently, we want to show that

$$opt_d(y') + opt_{d-1}(y) \ge opt_d(y) + opt_{d-1}(y'). \tag{9}$$

We'll probably want to use again induction somehow, which means we'll want to replace all these *opt*s with the recurrence formula to make them "one step smaller". Since the optimal splitting points are $x_{d,y'}$, $x_{d-1,y'}$, $x_{d,y}$, $x_{d-1,y}$ by definition, we have

$$\begin{aligned}
opt_d(y') &= opt_{d-1}(x_{d,y'}) &+ C(x_{d,y'}, y') \\
opt_{d-1}(y') &= opt_{d-2}(x_{d-1,y'}) &+ C(x_{d-1,y'}, y') \\
opt_d(y) &= opt_{d-1}(x_{d,y}) &+ C(x_{d,y}, y) \\
opt_{d-1}(y) &= opt_{d-2}(x_{d-1,y}) &+ C(x_{d-1,y}, y).
\end{aligned}$$

Substituting these to (9), this is now what we want to show:

$$\begin{aligned}
& opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y') + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y) \\
\geq\ & opt_{d-1}(x_{d,y}) + C(x_{d,y}, y) + opt_{d-2}(x_{d-1,y'}) + C(x_{d-1,y'}, y').
\end{aligned} \tag{10}$$

This is now a bit unwieldy. However, if we recall what each of these terms represent, this expanded form now somewhat informs us what we need to do. Recall that the right-hand side is $opt_d(y) + opt_{d-1}(y')$, because $x_{d,y}$ and $x_{d-1,y'}$ are the optimal splitting points. Thus, we want to prove that the left-hand side contains *suboptimal* splittings for $opt_d(y)$ and $opt_{d-1}(y')$, respectively. So we can simplify it a bit to:

$$opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y') + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y) \geq opt_d(y) + opt_{d-1}(y'). \tag{11}$$

This is now what we **want to show**.

So what are the **things that we know**? Consider the right-hand side. Since $opt_d(y)$ and $opt_{d-1}(y')$ represent the optimal splittings, we have the following inequalities for any x:

$$\begin{aligned}
opt_{d-1}(x) + C(x, y) &\geq opt_d(y) && \text{for } x \leq y, \\
opt_{d-2}(x) + C(x, y') &\geq opt_{d-1}(y') && \text{for } x \leq y'.
\end{aligned}$$

In particular, substituting the points $x_{d,y'}$ and $x_{d-1,y}$, we have the following inequalities:

$$\begin{aligned}
opt_{d-1}(x_{d,y'}) &+ C(x_{d,y'}, y) &&\geq opt_d(y) && (12) \\
opt_{d-2}(x_{d,y'}) &+ C(x_{d,y'}, y') &&\geq opt_{d-1}(y') && (13) \\
opt_{d-1}(x_{d-1,y}) &+ C(x_{d-1,y}, y) &&\geq opt_d(y) && (14) \\
opt_{d-2}(x_{d-1,y}) &+ C(x_{d-1,y}, y') &&\geq opt_{d-1}(y'). && (15)
\end{aligned}$$

Actually, by adding pairs of these together, we can get pretty close to the inequality that we want, (11)! For example, adding (13) and (14) together, we get an equation that's very similar to what we want, but not quite the same:

$$opt_{d-2}(x_{d,y'}) + C(x_{d,y'}, y') + opt_{d-1}(x_{d-1,y}) + C(x_{d-1,y}, y) \geq opt_d(y) + opt_{d-1}(y'). \tag{16}$$

Similarly, adding (12) and (15), we get another very similar equation:

$$opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y) + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y') \geq opt_d(y) + opt_{d-1}(y'). \tag{17}$$

Unfortunately, they're not exactly the same as (11).

However, you should know the trick by now. We have something that we *want to show* (11), and we have something that *we know* (16), so by subtracting them, we get something that **would be nice** to prove:

$$opt_{d-1}(x_{d,y'}) + opt_{d-2}(x_{d-1,y}) - opt_{d-1}(x_{d-1,y}) - opt_{d-2}(x_{d,y'}) \geq 0.$$

If we can prove this, then we're done; we can add it to (16) to get (11).

However, this is basically just the quadrangle inequality on *opt*! In other words, it's basically what we're trying to prove, but one step lower. So it's true by induction, as long we we verify that the following inequality holds: $x_{d,y'} \leq x_{d-1,y}$.

Sadly, this isn't always true. But at least we've solved the cases where it's true. So all we have to do is handle the remaining cases. So in what follows, **we'll assume that** $x_{d-1,y} \leq x_{d,y'}$.

Luckily, we still have an inequality we haven't used yet: (17). So by "subtracting" inequalities again, we get another thing that **would be nice** to prove:

$$C(x_{d,y'}, y') + C(x_{d-1,y}, y) - C(x_{d,y'}, y) - C(x_{d-1,y}, y') \geq 0.$$

Again, if we can prove this, then we're done; we can add it to (17) to get (11).

But now, this is just the quadrangle inequality, but on C! Thus, this is true as long as we verify the following inequalities:

$$x_{d-1,y} \leq x_{d,y'} \leq y' \leq y.$$

The last one is true by assumption. The middle one is true because $x_{d,y'}$ is a splitting point in the range $\{1, \ldots, y'\}$, so naturally it's at most y'. Finally, the first one is true because we're currently in that case!

So we've finished the proof. □

The previous proof went a bit long because it included our scratchwork and exploration. Here's a cleaned-up version if you want to read it:

Proof. We will prove the following two things by induction on d:

1. $x_{d,y} \geq x_{d-1,y}$
2. $opt_d(y') + opt_{d-1}(y) \geq opt_d(y) + opt_{d-1}(y')$ for any $y' \leq y$.

First part. We'll prove that for any $x < x_{d-1,y}$, x is a suboptimal splitting point for $opt_d(y)$. This proves that the optimal splitting point, $x_{d,y}$, is at least $x_{d-1,y}$, which proves the first part.

Because $x_{d-1,y}$ is the optimal splitting point for $opt_{d-1}(y)$, we have:

$$opt_{d-2}(x) + C(x, y) \geq opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y). \tag{18}$$

On the other hand, by induction,

$$opt_{d-1}(x) + opt_{d-2}(x_{d-1,y}) \geq opt_{d-1}(x_{d-1,y}) + opt_{d-2}(x), \tag{19}$$

using the fact that $x < x_{d-1,y}$. Adding these two together and simplifying, we get:

$$opt_{d-1}(x) + C(x, y) \geq opt_{d-1}(x_{d-1,y}) + C(x_{d-1,y}, y).$$

This says that $x_{d-1,y}$ is a better splitting point than x for $opt_d(y)$, which is what we wanted to show.

Second part. Next, we will show that for $y' \leq y$,

$$opt_d(y') + opt_{d-1}(y) \geq opt_d(y) + opt_{d-1}(y'). \tag{20}$$

By definition,

$$opt_d(y) = opt_{d-1}(x_{d,y}) + C(x_{d,y}, y)$$
$$opt_{d-1}(y') = opt_{d-2}(x_{d-1,y'}) + C(x_{d-1,y'}, y').$$

Also, the following is true for any $x \leq y$ and $x' \leq y'$:

$$opt_d(y) \leq opt_{d-1}(x) + C(x, y)$$
$$opt_{d-1}(y') \leq opt_{d-2}(x') + C(x', y').$$

Adding these, we get:

$$opt_{d-1}(x) + C(x, y) + opt_{d-2}(x') + C(x', y') \geq opt_d(y) + opt_{d-1}(y'). \tag{21}$$

We will now handle two cases.

- **The case $x_{d,y'} \leq x_{d-1,y}$.** Substituting $x := x_{d-1,y}$ and $x' = x_{d,y'}$ to (21), we get

  $$opt_{d-1}(x_{d-1,y}) + C(x_{d-1,y}, y) + opt_{d-2}(x_{d,y'}) + C(x_{d,y'}, y') \geq opt_d(y) + opt_{d-1}(y').$$

  (Note that this is valid because $x = x_{d-1,y} \leq y$ and $x' = x_{d,y'} \leq y'$.)
  By induction (using $x_{d,y'} \leq x_{d-1,y}$)

  $$opt_{d-1}(x_{d,y'}) + opt_{d-2}(x_{d-1,y}) \geq opt_{d-1}(x_{d-1,y}) + opt_{d-2}(x_{d,y'}).$$

  Adding these together and simplifying, we get

  $$opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y') + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y) \geq opt_d(y) + opt_{d-1}(y').$$

  However, by definition of $x_{d,y'}$ and $x_{d-1,y}$, the left-hand side is just $opt_d(y') + opt_{d-1}(y)$, so the inequality becomes

  $$opt_d(y') + opt_{d-1}(y) \geq opt_d(y) + opt_{d-1}(y'),$$

  which is what we wanted to show.

- **The case $x_{d-1,y} \leq x_{d,y'}$.** Substituting $x = x_{d,y'}$ and $x' = x_{d-1,y}$ to (21), we get

  $$opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y) + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y') \geq opt_d(y) + opt_{d-1}(y').$$

  (Note that this is valid because $x = x_{d,y'} \leq y' \leq y$ and $x' = x_{d-1,y} \leq x_{d,y'} \leq y'$.)
  We have $x_{d-1,y} \leq x_{d,y'} \leq y' \leq y$, so by the quadrangle inequality,

  $$C(x_{d-1,y}, y) + C(x_{d,y'}, y') \geq C(x_{d-1,y}, y') + C(x_{d,y'}, y).$$

Adding these together and simplifying, we get

$$opt_{d-1}(x_{d,y'}) + C(x_{d,y'}, y') + opt_{d-2}(x_{d-1,y}) + C(x_{d-1,y}, y) \geq opt_d(y) + opt_{d-1}(y').$$

Just like before, this is equivalent to

$$opt_d(y') + opt_{d-1}(y) \geq opt_d(y) + opt_{d-1}(y')$$

which is what we wanted to show.

This finishes the proof.

□

Actually, this proof is still somewhat sloppy: just like before, there may be several optimal splitting points, and we didn't check that $x_{d,y}$ is the *largest* one among them. Also, we didn't prove the base case of the induction!

Exercise 3.7. Complete the proof by proving the base case, $d = 1$.

4 Problems

4.1 Non-coding problems

No need to be very formal in your answers!

N1 [20★] Prove Theorem 3.4.

Hint: You more-or-less just mimic our proof of Theorem 3.2.

N2 [20★] In the proofs we had for the Divide and Conquer optimization and the Knuth optimization, we mentioned that there was a bit of sloppiness because there may be several optimal splitting points, and we didn't check that $x_{d,y}$ is the *largest* one among them.

One way to fix the proofs would be to meticulously keep track of which inequalities are strict and which ones are not, but that may be tedious.

Show that you can also fix the proofs by perturbing the cost function to force uniqueness of optimal splitting points, say via

$$C_{\text{new}}(x, y) = C_{\text{old}}(x, y) - 2^x \varepsilon$$

where ε is an infinitesimally small number. (We don't have to do this in code; this is just for the proof.)

Don't forget to verify the quadrangle inequality for this updated cost function!

N3 [10★] One of my suggestions for handling the indices $n < k \leq 2n$ in Alienmons (Problem 3.1) is to pad the array R and B with ∞ values. One issue with this is that the array R won't be convex anymore.

Explain how to fix this by *padding r instead*, or equivalently, padding R with the values "$[\infty, 2\infty, 3\infty, \ldots]$".

N4 [30★] Let $opt(d, y)$ be as we defined them while solving Guardians of the Lunatics (Problem 3.2). Show that $opt(d, y)$ is *convex* in columns, i.e.,

$$opt(d, y) - opt(d-1, y) \leq opt(d-1, y) - opt(d-2, y).$$

Note: Keeping up with the module's theme, this property may sometimes be exploited to improve the solution even further, faster than when using just the Divide and Conquer optimization or Knuth's optimization.

N5 [15★] Consider the DP solution for Problem 2.2, which runs in $\mathcal{O}(n^2)$. Show that if C satisfies the quadrangle inequality, then the optimal splitting points are nondecreasing.

N6 Consider the following problem.

> *Given a cost function C and an integer n, partition $[0, n]$ into some number of intervals*
>
> $$[i_0, i_1], [i_1, i_2], \ldots, [i_{k-1}, i_k]$$
>
> *such that:*
>
> - $0 = i_0 \leq i_1 \leq \ldots \leq i_k = n$.
> - *The following total cost is minimized:* $C(i_0, i_1) + C(i_1, i_2) + \ldots + C(i_{k-1}, i_k)$.

You may choose any $k > 0$.

- [10★] Find a DP recurrence for this problem that can be used to solve it in $\mathcal{O}(n^2)$.
- [10★] If C satisfies the quadrangle inequality, show that the optimal splitting points are nondecreasing.
- [10★] If C doesn't necessarily satisfy the quadrangle inequality, show that the optimal splitting points may not be nondecreasing.

N7 This problem attempts to demonstrate why the convex hull trick is called that, by actually relating it with convex hulls!

- [20★] Describe a data structure that can dynamically maintain the convex hull of a set of points on the plane. The two operations are:
  - **Update.** Add a new point (x, y).
  - **Query.** Given a vector v, find the "first point along the direction v".

  Hint: Exploit the fact that as you go around the polygon counterclockwise, the angle vectors are also rotating counterclockwise.
- [20★] Implement the convex hull trick by representing each line $L(x) = a + bx$ as a point (a, b), and maintaining their convex hull as above.
  Hint: Notice that $L(x)$ is the dot product of $\langle a, b\rangle$ and $\langle 1, x\rangle$.
- [10★] When implementing the convex hull trick this way, explain why we don't have to maintain the full convex hull, but only "half of it".
- [5★] (Optional) Explain how the previous question is related to the upper and lower envelopes.
- [10★] (Optional) Now, suppose the lines are inserted in increasing order of slope. Recall that we can simply use a stack to construct the envelope in this case.
  Explain how this is related to computing the convex hull via Andrew's monotone chain algorithm.
- [10★] (Optional) Explain how to use a stack to maintain the dynamic envelope assuming that all lines have $a < 0$, and the lines will be inserted in increasing order of b/a.

N8 Consider the following problem. (It's a classical problem, just discretized.)

> *Given n, you wish to construct a* static *binary search tree (BST) with n nodes, with the keys*
>
> $$1, 3, 5, \ldots, 2n - 1.$$
>
> *Without any additional info, your first instinct is to construct a balanced BST. However, if you know that some keys will be accessed more than others, then it may make sense to imbalance the tree a bit so that the total access time is minimized.*
> *Suppose your tree will be queried on the keys*
>
> $$0, 1, 2, \ldots, 2n,$$
>
> *and suppose that key k will be queried c_k times for each k from 0 to $2n$. The cost of a query is equal to the number of nodes visited while looking for that key. We want to minimize the* total *cost across all queries.*
> *What is the minimum total cost, if you construct your BST optimally?*

- [20★] Find an $\mathcal{O}(n^3)$ DP solution for this problem. (This is normal DP—no techniques from this module are needed.)
- [10★] Improve this DP solution to $\mathcal{O}(n^2)$.[5]
- [30★] Prove that your solution is correct.
- [20★] Generalize this $\mathcal{O}(n^3) \to \mathcal{O}(n^2)$ optimization to arbitrary cost functions. (You should be able to know what a "cost function" is after doing the above, but if not, feel free to ask in Discord.)
- [15★] Show that the running time is still $\mathcal{O}(n^2)$ regardless of the order in which we fill in the DP table (as long as they're a valid topological order of the states, of course).

By the way, we mention that in the case where $c_1 = c_3 = \ldots = c_{2n-1} = 0$, there's a solution that runs in $\mathcal{O}(n \log n)$ time. It's called the Garsia–Wachs algorithm.

4.2 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[6] Making the implementation self-contained in a `class` or `struct` makes it very easy for you to reuse, which is handy for your future contests!

C1 [10★] Implement the convex hull trick by maintaining the convex hull, as detailed in Problem **N7**.

C2 [10★] Implement the poor man's convex hull.

S1 [50★] **Guardians of the Lunatics:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/guardians-lunatics

S2 [50★] **Branch Assignment:** https://open.kattis.com/problems/branch (ICPC World Finals 2016)

S3 [50★] **Ciel and Gondolas:** https://codeforces.com/problemset/problem/321/E

S4 [50★] **Cutting Sticks:** (UVa) Online Judge 10003 (Solve it in $\mathcal{O}(n^2)$.)

S5 [50★] **Gain Battle Power:** (UVa) Online Judge 12836

S6 [50★] **Magic Value:** https://www.hackerrank.com/contests/university-codesprint-4/challenges/magic-value

S7 [50★] **Optimal Bus Stops:** https://www.hackerrank.com/contests/101hack53/challenges/optimal-bus-stops

S8 [50★] **Pair Sums:** https://www.hackerrank.com/challenges/pair-sums. Then see the editorial for a description of simpler implementations in some cases, including a "divide and conquer" implementation.

S9 [50★] **Two Buildings:** https://codeforces.com/gym/102920/problem/L

Acknowledgment

Thanks to Cisco for writing a nice blog post introducing the divide and conquer optimization! I incorporated a lot of it in section 3. He should post it in the NOI.PH blog already!

[5] I think this is historically the first real use of the Knuth optimization, by Knuth himself.

[6] See also: https://en.wikipedia.org/wiki/Modular_programming.