香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Dynamic Programming (III)

Nicholas Ko {__jk__}
2026-03-21

Prerequisites and Overview

- This lecture is about dynamic programming (DP) optimisation, so obviously you should know what DP is, and how to write DP properly, before learning how to optimise it.

- Four main perspectives to optimise DP transitions will be discussed:
  - with data structures,
  - with monotonic queues,
  - with geometric reasoning, and
  - with divide and conquer.

Problems on DP Optimisation (Challenging)

- If you have attended this lecture in previous years and believe you have fully understood the content, I invite you to try the following problems.
- **Warning**: all of the practice problems below are challenging. Please do not feel (too) bad if you cannot complete all, or even *any*, of them. (More approachable practice problems are the end of the slides.)
- (ICPC Asia East Continent Final 2025 Problem E) Efficient Express
- (ICPC Asia Pacific Championship 2025 Problem K) Book Sorting
- (Korea TST 2023 Day 1 Problem 3) Taxi
- (Korea TST 2024 Day 1 Problem 1) Maximum Average (difficult)

Why DP optimisation?

- Suppose you have come up with a DP transition
  - State definition:
    dp[i][j] represents ...
  - State transition:
    dp[i][j] = (some formula in terms of other dp states)
  - Base case(s)
    dp[0][0] = ...

  ... but the time complexity is still too high.
- Aim: modify the DP state definition and/or the transition formula in a form that you can optimise it.

Example 1 — (A310) DP Optimization: Partial Sum

- Input gives you $0 \le a[1], a[2], ..., a[N] \le K \le 10^5$.
- Output the number of distinct arrays A[1], A[2], ..., A[N] so that
  - $0 \le A[x] \le a[x]$ for every index x (from 1 to N), and
  - A[1] + A[2] + ... + A[N] is exactly K.

- Consider the following possible approach:

  Let dp[i][j] be the number of distinct arrays A[1], A[2], ... A[i], so that
  - $0 \le A[x] \le a[x]$ for every index x (from 1 to i), and
  - A[1] + A[2] + ... + A[i] is exactly j.

Example 1 — (A310) DP Optimization: Partial Sum

- Let dp[i][j] be the number of distinct arrays A[1], A[2], ... A[i], so that
  - $0 \le A[x] \le a[x]$ for every index x (from 1 to i), and
  - A[1] + A[2] + ... + A[i] is exactly j.

- Then the base case is dp[0][0] = 1, and the transition formula for $i \ge 1$ is

  dp[i][j] = sum {from k = 0 to min(a[i], j)} of dp[i - 1][j - k]

  and the output is dp[N][K].
- There are O(NK) states, and each state takes up to O(max A) = O(K) time to compute. The total time complexity is therefore $O(NK^2)$.

Example 1 — (A310) DP Optimization: Partial Sum

- The transition formula for i ≥ 1 is

  dp[i][j] = sum {from k = 0 to min(a[i], j)} of dp[i - 1][j - k]

- What if we store the partial sum array of dp[i - 1]? Would it be helpful?
- Consider the partial sum array of dp[i - 1], and denote it as

  ps[i][j] = sum {from k = 0 to j} of dp[i][k]

- The transition formula above can then be rewritten in terms ps[i][j]:

  dp[i][j] = ps[i - 1][j] - ps[i - 1][j - min(a[i], j) - 1]

  (if j ≤ a[i], then the second term doesn't exist and we treat it as 0.)

Example 1 — (A310) DP Optimization: Partial Sum

- The transition formula above can then be rewritten in terms ps[i][j]:

  $$dp[i][j] = ps[i - 1][j] - ps[i - 1][j - \min(a[i], j) - 1]$$

  (if $j \le a[i]$, then the second term doesn't exist and we treat it as 0.)

- Here is a possible optimised solution:
  - Initialise the dp and ps arrays
  - Iterate in increasing order of i (from 1 to N)
    - Calculate dp[i][j] for each $0 \le j \le K$
    - Using the calculated values of ps[i][j], update ps[i][j] for each $0 \le j \le K$ accordingly
- Each DP state now takes O(1) time only; updating the ps array is O(K) for each i, and we do it N times, so the total time complexity is O(NK).

What have we learnt from Example 1?

- We wrote out the transition formula for dp[i][j], and realised that its computation can be sped up if we stored the partial sum array of dp[i - 1].
- In retrospect, what properties of the dp approach have we utilised?
- The fact that each dp[i][j] is the sum of some previous contiguous dp states enabled us to use partial sum.
  (Off-topic question: what if the states are not contiguous? For example, if you are given a set S in input, and dp[i] = sum over {s ≤ i in S} of (dp[i - s]), is it still possible to compute this DP?)
- More specifically, the part that we optimised is querying the sum of a subarray, which we all know is achieved using a partial sum array which helped us optimise from O(K) to O(1) time per “query”.

What have we learnt from Example 1?

- The fact that each dp[i][j] is the sum of some previous contiguous dp states enabled us to use partial sum.

- Generally, whenever you see that your dp transition formula is of the form

  dp[i] = sum {from j = l[i] to r[i]} of a[j]

  where l[], r[], and a[] are arrays that are already known, then it might be suitable to consider using partial sum optimisation.

What have we learnt from Example 1?

- In fact, this technique we just discussed can generalise even if the transition formula involves something other than sums!
- Say, for example, that we encounter something of the form

$$dp[i] = \textbf{max} \{\text{from } j = l[i] \text{ to } r[i]\} \text{ of } a[j]$$

- We can no longer use a partial sum array
- ... but we can just change the partial sum portion to a sparse table or segment tree that can maintain and query maximum in a subarray, and the idea works exactly the same as before!

What have we learnt from Example 1?

- What if we encounter something of the form

  $$dp[i] = \mathbf{op} \{\text{from } j = l[i] \text{ to } r[i]\} \text{ of } a[j]$$

  where **op** is some binary operation $\mathbf{op} : S \times S \to S$ on some set S?

- Referring to the lecture *Variants of Segment Tree*, as long as (S, **op**) is a monoid, then you can consider using a segment tree to maintain the DP array, so that you can compute each subsequent state (by making a query on the segment tree) in logarithmic time.

What have we learnt from Example 1?

- As long as (S, **op**) is a monoid, you can consider using a segment tree to maintain the DP array, so that you can compute each subsequent state in logarithmic time.
- This idea also generalises to higher dimensions. There is nothing stopping you to use 2D segment trees, or k-d trees to maintain information about the processed DP states (of course, the time complexity would increase).

- In particular, the observant reader may notice that the convex hull trick, discussed later in this lecture, is (very nearly) a weak version of this technique when you use a Li-Chao segment tree.

Example 2 — (A311) DP Optimization: Monotone Queue

- Input gives you $-10^9 \le a[1], a[2], ..., a[N] \le 10^9$ and an integer M.
- The elements at the two ends, a[1] and a[N] are always picked.
  You can choose (independently) to pick a[2], ..., a[N-1] yourself or not, but the indices of neighbouring chosen indices must have a difference of ≤ M
- Output the maximum possible sum of values picked.

- If you use a segment tree to maintain the DP array, just like what we have discussed, you can obtain a O(N log N) solution. (Exercise: implement it!)
- This time, our aim is to come up with a O(N) solution.

Example 2 — (A311) DP Optimization: Monotone Queue

- The very first step of approaching DP problems is always to write out the state definitions, transitions, and base case(s).
- We let dp[i] be the maximum sum of our picked subsequence if we consider the prefix a[1], a[2], ..., a[i] only, and that index i is picked. Then

  $$dp[i] = \max \{\text{from } j = \max(1, i - M) \text{ to } i - 1\} \text{ of } dp[j] + a[i]$$

  for every $2 \le i \le N$. The base case is $dp[1] = a[1]$, and our answer is $dp[N]$.
- From a data structure perspective, we essentially want to support:
  - appending a new element,
  - removing the currently oldest element,
  - querying the current maximum element.

Example 2 — (A311) DP Optimization: Monotone Queue

- We have dp[i] = max {from j = max(1, i - M) to i - 1} of dp[j] + a[i].
- From a data structure perspective, we essentially want to support:
  - appending a new element,
  - removing the currently oldest element,
  - querying the current maximum element.

- Instead of storing all active elements, we only store *"useful" candidates* that might be the current maximum element.
- The main idea is that we maintain a std::deque (double-ended queue), in which the dp values are (strictly) decreasing.

Example 2 — (A311) DP Optimization: Monotone Queue

- We have dp[i] = max {from j = max(1, i - M) to i - 1} of dp[j] + a[i].
- The main idea is that we maintain a std::deque (double-ended queue), in which the dp values are (strictly) decreasing.
- What does this mean? Why would this be helpful?

- Suppose that we are calculating the dp values from left to right, and have just found the value of dp[i].
- Then every index $j < i$ with $dp[j] \leq dp[i]$ will never be useful again.
- This is because the index i is newer (and hence stays longer in the "window" of being a possible transition) than indices < i, and brings at least as much value as index j.

Example 2 — (A311) DP Optimization: Monotone Queue

- So, after calculating dp[i], we can remove all values in the deque which has value ≤ dp[i] because they are representing indices older than i.
- Simply put, the important invariants of the deque are that
  - it maintains a list of increasing indices between max(1, i - M) and i - 1, and
  - an index j in [max(1, i - M), i - 1] is in the deque while calculating dp[i] if and only if a[j] > a[k] for every k in [j + 1, i - 1].
- The first condition says that the indices are still in the possible “transition window” of i, and the second condition says that index j has not been removed due to the insertion of subsequent newer elements.

Example 2 — (A311) DP Optimization: Monotone Queue

- So for every $2 \le i \le n$ in increasing order, after removing the old indices, the best transition point is simply the index which is currently at the front of the deque, so we can just set dp[i] = dp[deque.front] + a[i].
- ... and afterwards, we remove smaller values at the back of the deque so that the dp values in the deque remains decreasing.

- Therefore, a possible solution could look like this:
  - Initialise dp[1] = a[1], and deque = {1}.
  - For $2 \le i \le n$ in increasing order,
    - While deque.front < i - M, pop the front of the deque.
    - Set dp[i] = dp[deque.front] + a[i].
    - While deque is not empty and dp[deque.back] ≤ dp[i], pop the back of the deque.
    - Push the index i into the back of the deque.

Example 2 — (A311) DP Optimization: Monotone Queue

- Initialise dp[1] = a[1], and deque = {1}.
- For 2 ≤ i ≤ n in increasing order,
  - While deque.front < i - M, pop the front of the deque.
  - Set dp[i] = dp[deque.front] + a[i].
  - While deque is not empty and dp[deque.back] ≤ dp[i], pop the back of the deque.
  - Push the index i into the back of the deque.

- What is the overall time complexity? Each index gets pushed into the deque once (and thus gets popped out at most once), so we have a O(N) complexity solution now.

What have we learnt from Example 2?

- We realised that once a index j has a smaller dp value than a later index i, then j becomes useless forever, and we neglect it from index i onwards.
- In retrospect, what properties of the dp approach have we utilised?
- Our solutions revolves around the fact that once an element is removed from the deque, it can never be inserted back again. In terms of the transition formula, this is a sufficient condition for when
  - the starting index for the windows (i.e. max(1, i - M) in Example 2) are nondecreasing, and
  - if index j_0 is a better transition point than index j_1 (where $j_0 > j_1$ are both in the window) at index i, then j_0 continues to be a better transition point than index j_1 at all indices larger than i, and hence j_1 can be discarded permanently.

What have we learnt from Example 2?

- Our conditions are
  - the starting index for the windows (i.e. max(1, i - M) in Example 2) are nondecreasing, and
  - if index j_0 is a better transition point than index j_1 (where $j_0 > j_1$ are both in the window) at index i, then j_0 continues to be a better transition point than index j_1 at all indices larger than i, and hence j_1 can be discarded permanently

- Just like what we did in Example 1, we try to generalise as widely as possible. Suppose we have a transition formula

  dp[i] = **op** {j = l[i] to r[i]} over dp[j] + a[i]

- If both arrays l and r are increasing and **op** = max or min, then we can use the monotonic queue optimisation.

What have we learnt from Example 2?

- Suppose we have a transition formula

$$dp[i] = \textbf{op}\ \{j = l[i] \text{ to } r[i]\} \text{ over } dp[j] + a[i]$$

- Unfortunately, if **op** is not max or min (± constant), it is quite rare that we could use the monotonic queue optimisation directly, because it is precisely the key property of “permanent dominance” that allows the deque we are maintaining to be monotonic (hence the name of the technique).

- As you might expect, there is something called a monotonic stack optimisation as well. It behaves exactly like how you would expect it to.

Aside: if f(i) is not monotonic

- If you read previous years' DP (III) slides, there is a short section that says even if l[i] is not monotonic in dp[i] = max {j = l[i] to r[i]} over dp[j] + a[i], you can still use a priority queue instead of a deque, as I quote:
  "Instead of popping elements in the front that "expired", binary search on first element that hasn't "expired". As the value in the monotone queue is decreasing, it is the largest possible value."

- Exercise: suggest a reason why I might think this is useless.

Aside: if f(i) is not monotonic

- *"Instead of popping elements in the front that "expired", binary search on first element that hasn't "expired". As the value in the monotone queue is decreasing, it is the largest possible value."*

- Answer: since you are using a priority queue, its time complexity is O(N log N) already. You might as well directly maintain the DP with a segment tree (like in Example 1). In each segment tree node, you store the max / min value of all the active nodes in that interval. This is also O(N log N) and is more versatile, albeit with a larger constant factor.
  (If evil problemsetters force you to do constant optimisation, then that's another story ~~and it is then the problemsetters' fault, not yours.~~)

A Step towards Generalisation

- We have been dealing with transitions of the form

  dp[i] = (**op** {j = l[i] to r[i]} over f(j)) + a(i)

  where **op** is either max or min.

- What if we wanted to deal with cross terms, say for example,

  dp[i] = (**op** {j = 0 to i - 1} over f(i) * g(j) + a(i) + b(j))

  where f[], g[], a[], b[] are arrays that are already known, and **op** is either max or min?

Example 3 — (A312) DP Optimization: Convex Hull Trick

- Input gives you two arrays A[1], A[2], ..., A[N] and B[1], B[2], ..., B[N], where A is strictly increasing and B is strictly decreasing.

  Find a set of indices $\{1, N\} \subseteq S$ such that the sum$(A[S_{i+1}] * B[S_i])$ is minimised.
- What is the DP state definition and transition formula?
- We let dp[i] be the maximum desired sum among all prefix of length i, and that index i is in the set S.
- Then, the base case is dp[1], our answer is dp[N], and for all $2 \leq i \leq N$,

$$dp[i] = \min \{j = 1 \text{ to } i - 1\} \text{ of } (dp[j] + B[j] * A[i])$$

Example 3 — (A312) DP Optimization: Convex Hull Trick

- dp[i] = min {j =1 to i - 1} of (dp[j] + B[j] * A[i]). A increasing, B decreasing.

- **Claim:** this is equivalent to the following process:
  - We maintain a set S of lines on the Cartesian coordinate plane.
  - Initially, we set dp[1] = 0, and S has only the line {y = B[1] * x}.
  - For every 2 ≤ i ≤ N increasing,
    - Set dp[i] = minimum value of y over all lines in S when we put x = A[i].
    - Add the line {y = B[i] * x + dp[i]} into the set S.

  (Exercise: convince yourself and/or prove that both procedures do the same thing, before continuing on to the next slide.)

Aside: Push and Pull DP

- From the "Introduction to DP" page from USACO:
  "There are two main DP approaches:
  - ***Push DP**, where we calculate future states based on the current state*
  - ***Pull DP**, where we calculate the current state based on past states"*

  (or in previous HKOI Training terminology, bottom-up DP and top-down DP)
- In our first example today, we saw that Pull DP is rather compatible with quick lookups and queries with the help of data structures.
- We have just started with our third example problem, and what I did just now is to encourage you to think in terms of Push DP instead of Pull DP.
- It is frequently the case that only one of the two approaches can help us do DP optimisation. Be flexible!

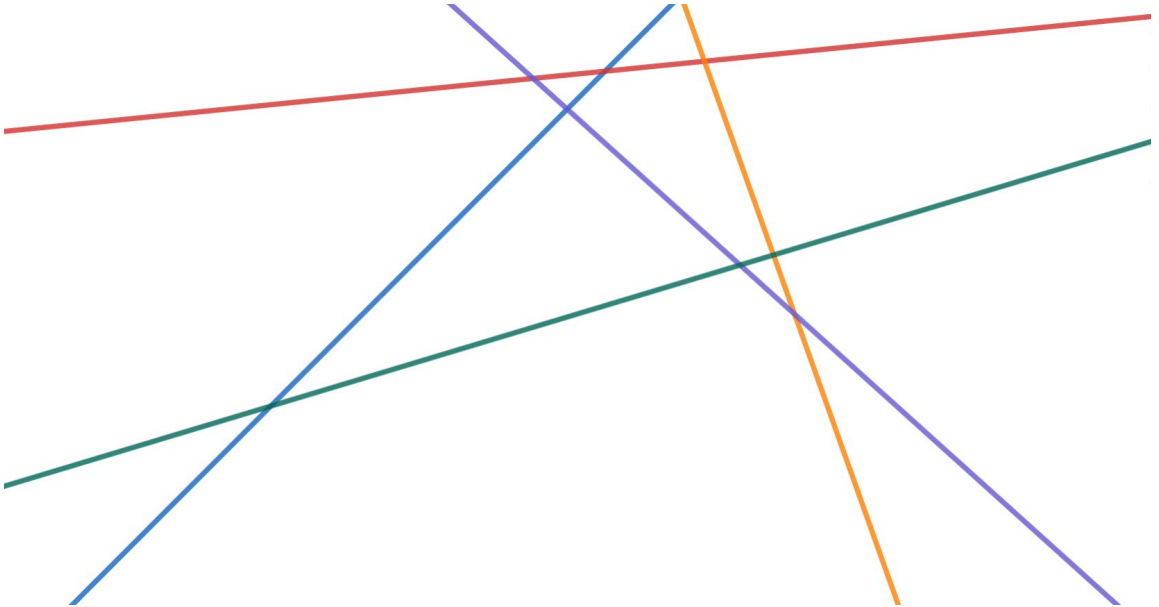
Example 3 — (A312) DP Optimization: Convex Hull Trick

- Initially, we set dp[1] = 0, and S has only the line {y = B[1] * x}.
- For every 2 ≤ i ≤ N increasing,
  - Set dp[i] = minimum value of y over all lines in S when we put x = A[i].
  - Add the line {y = B[i] * x + dp[i]} into the set S.

- Since we are only interested in the minimum value when calculating the DP values, it suffices for us to maintain the lower hull of S. (Informally, "lower hull" is the subset of line (segment)s that give the minimum y value for some x.)
- Naturally, when we add a new line L into the original hull of S, we shall maintain it by replacing the sections (exercise: prove that "the sections" is either empty or an interval) that is above L.

Example 3 — (A312) DP Optimization: Convex Hull Trick

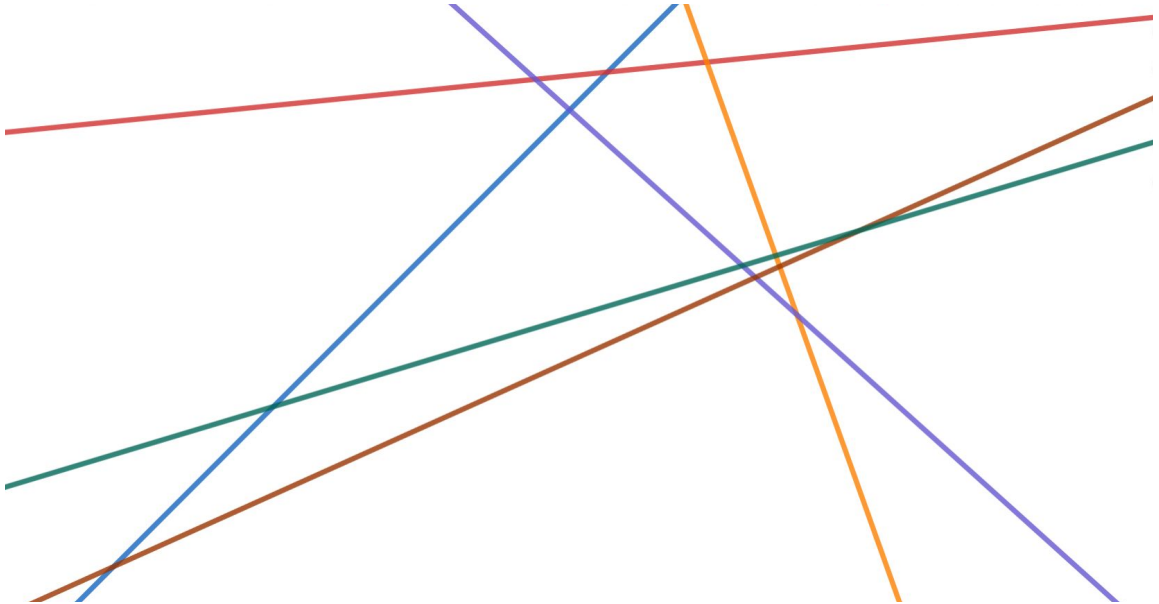
- Naturally, when we add a new line L into the original hull of S, we shall maintain it by replacing the sections that is above L.
- Consider the example on the right:
- The lower hull is currently in the order of blue-green-purple-gold, from left to right.



Example 3 — (A312) DP Optimization: Convex Hull Trick

- Naturally, when we add a new line L into the original hull of S, we shall maintain it by replacing the sections that is above L.
- The lower hull was blue-green-purple-gold.
- Now that the brown line L is newly inserted,
  - Blue is shortened
  - Green disappeared
  - Purple is shortened

  ⇒ “the sections” replaced by brown



Example 3 — (A312) DP Optimization: Convex Hull Trick

- Initially, we set dp[1] = 0, and S has only the line {y = B[1] * x}.
- For every 2 ≤ i ≤ N increasing,
  - Set dp[i] = minimum value of y over all lines in S when we put x = A[i].
  - Insert the line {y = B[i] * x + dp[i]} into the set S.

- We can use a std::set to maintain the set of lines on the lower hull in increasing order of x-coordinates. For every line in the set, we also store the range (interval) of x-coordinates for which it has the smallest value.

Example 3 — (A312) DP Optimization: Convex Hull Trick

- Every line gets inserted (and hence deleted at most once). Along with every line insertion, at most two lines will have their range shortened but not completely removed. (Why?)
- Hence, there are at most O(N) modifications to the elements in the std::set. Since find, insert, and delete operations are all O(log N) each, we have a O(N log N) working solution.
- Although the folklore implementation is only 25 lines long, it is notoriously difficult to implement correctly as it is extremely concise. I *strongly recommend* you to try coding this once yourself, instead of blindly copying and/or memorising the following few slides.

Example 3 — (A312) DP Optimization: Convex Hull Trick

- We declare a set of “Line”s.
  This stores the set of lines on the min hull.
- The lines are sorted according to their “dominating interval” on the hull, from left to right. However, in practice, we often write the custom comparator by sorting the slopes of the lines instead (from large to small).
  (Exercise: why are these two orders equivalent?)
- For each line in the hull, we store it in the form y = mx + c. Additionally, we maintain the right endpoint of its dominating interval (and so the left endpoint is then previous line’s right endpoint + 1).

```
struct Line {
    int m, c;  // y = mx + c
    mutable int ep;  // right endpoint
    bool operator<(const Line& o) const {
        return m > o.m;  // sort by slope
    }
};
set <Line> s;
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- The “dominate” function takes in two neighbouring lines.
  It recalculates the endpoint of the left line segment, and hence determines the left segment entirely dominates the right segment (meaning the right segment becomes useless).
- Exercise: find at least 3 bugs in the code on the right.

```
bool dominate(set <Line>::iterator itl,
              set <Line>::iterator itr) {
    itl->ep = (itr->c - itl->c) / (itl->m - itr->m);
    return itl->ep >= itr->ep;
}
void add_line(int m, int c) {
    auto cur = s.insert({m, c}).first;
    auto prv = prev(cur), nxt = next(cur);
    while (dominate(cur, nxt)) nxt = s.erase(nxt);
    if (dominate(prv, cur)) {
        dominate(prv, s.erase(cur));
    } else {
        while (prev(prv)->ep >= prv->ep) {
            cur = prv;
            prv = prev(cur);
            dominate(prv, s.erase(cur));
        }
    }
}
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- The equation in “dominate” comes from solving

  $l.m * x + l.c \le r.m * x + r.c$

  $\Leftrightarrow (l.m - r.m) * x \le r.c - l.c$

  so $x \le (r.c - l.c) \div (l.m - r.m)$ since we assumed slopes are sorted in decreasing order, and line l is to the right of line r in the hull.
- Exercise: find at least 3 bugs in the code on the right.

```
bool dominate(set <Line>::iterator itl,
              set <Line>::iterator itr) {
    itl->ep = (itr->c - itl->c) / (itl->m - itr->m);
    return itl->ep >= itr->ep;
}
void add_line(int m, int c) {
    auto cur = s.insert({m, c}).first;
    auto prv = prev(cur), nxt = next(cur);
    while (dominate(cur, nxt)) nxt = s.erase(nxt);
    if (dominate(prv, cur)) {
        dominate(prv, s.erase(cur));
    } else {
        while (prev(prv)->ep >= prv->ep) {
            cur = prv;
            prv = prev(cur);
            dominate(prv, s.erase(cur));
        }
    }
}
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- To add a line into the hull, erase the line segments on its right of which the line dominates over.
- Similarly, erase the lines on its left of which it is being dominated by the new line.
- It might also be the case that the current line is being dominated and so we shall delete it.
- Exercise: find at least 3 bugs in the code on the right.

```
bool dominate(set <Line>::iterator itl,
              set <Line>::iterator itr) {
    itl->ep = (itr->c - itl->c) / (itl->m - itr->m);
    return itl->ep >= itr->ep;
}
void add_line(int m, int c) {
    auto cur = s.insert({m, c}).first;
    auto prv = prev(cur), nxt = next(cur);
    while (dominate(cur, nxt)) nxt = s.erase(nxt);
    if (dominate(prv, cur)) {
        dominate(prv, s.erase(cur));
    } else {
        while (prev(prv)->ep >= prv->ep) {
            cur = prv;
            prv = prev(cur);
            dominate(prv, s.erase(cur));
        }
    }
}
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- Querying the minimum at a point is straightforward: just find the line for which its dominating interval covers x.

- Because of this, it is a good idea to introduce another custom comparator so we can use std::set::lower_bound() directly.

```
struct Line {
    int m, c;  // y = mx + c
    mutable int ep;  // right endpoint
    bool operator<(const Line& o) const {
        return m > o.m;  // sort by slope
    }
    bool operator<(int x) const {
        return ep < x; // to handle queries
    }
};
set <Line, less<> > s;
int query(int x) {
    auto it = *s.lower_bound(x);
    return it.m * x + it.c;
}
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- Here are a few bugs in the previous code snippet for adding a new line. Think carefully how to suggest a fix for each of them.

- We have to handle the boundary cases where the iterators are begin() or end() of a set to make sure that we are assessing legitimate elements.
- Division by zero in the formula in “dominate” if two lines have the same slope (although this won’t happen to us in this problem, why?).
- If we have to deal with negative x-coordinates (again, not us here), then integer division in C++ truncates towards 0 instead of rounding down.

Example 3 — (A312) DP Optimization: Convex Hull Trick

- My entire implementation that allows lines with equal slopes:

```
multiset <Line, less<> > s;
bool dominate(set <Line>::iterator itl,
      set <Line>::iterator itr) {
  if (itr == s.end()) {
    itl->ep = INF;
    return false;
  }
  if (itl->m == itr->m) {
    if (itl->c < itr->c) itl->ep = INF;
    else itl->ep = -INF;
  } else {
    itl->ep = div(itr->c - itl->c, itl->m
       - itr->m);  // integer division
  }
  return itl->ep >= itr->ep;
}
```

```
void add_line(int m, int c) {
  auto cur = s.insert({m, c});
  auto nxt = next(cur);
  while (dominate(cur, nxt)) nxt = s.erase(nxt);
  if (cur == s.begin()) return;
  auto prv = prev(cur);
  if (dominate(prv, cur)) dominate(prv, s.erase(cur));
  else {
    while (prv != s.begin() && prev(prv)->ep >=
          prv->ep) {
      cur = prv;
      prv = prev(cur);
      dominate(prv, s.erase(cur));
    }
  }
}
int query(int x) {
    auto it = *s.lower_bound(x);
    return it.m * x + it.c;
}
```

Example 3 — (A312) DP Optimization: Convex Hull Trick

- However, we can still do better here; there is a O(N) solution.
- Recall that the problem guarantees that the array A is increasing and B is decreasing. How could we utilise this additional constraint?

- We translate this condition back in terms of lines — the weapon that we have been using: the slopes of the lines inserted is decreasing over time, and the x-coordinates that we are querying at is increasing over time.
- One can observe that the "section" for which a newly inserted line updates the lower hull always forms a nonempty interval that is unbounded above. (Why?)

Example 3 — (A312) DP Optimization: Convex Hull Trick

- One can observe that the “section” for which a newly inserted line updates the lower hull always forms a nonempty interval that is unbounded above.
- So, with a similar spirit as Example 2, one can use a monotonic queue to keep track of the lines instead of a std::set. That way, the insertions and deletions are O(1) per operation instead of O(log N).
- To handle queries efficiently as well, because “x-coordinates that we are querying at is increasing over time”, we can also achieve amortised O(1) time per query instead of O(log N). We have this solved the problem in linear time.

Summary on Example 3

- We changed perspective from pull DP to a push DP. This allowed us to model the transition formula in terms of lines in the 2D coordinate plane.
- We used a std::set to maintain the lower hull of the lines, by storing the actual lines and the interval for which the line is attaining the minimum value. Thanks to this structure, we can now achieve O(log N) query.

- We further optimised the solution by using a monotonic queue instead of a set because the problem gave us additional constraints that preserves monotonicity. This made us able to achieve amortised O(1) per update and query.

What have we learnt from Example 3?

- In retrospect, what properties of the dp approach have we utilised?
- The fact that the dp formula has the form (thing) * f(i) + (thing), where (thing)s are anything independent of i. This allowed us to illustrate each transition as a line.

- More generally, if your transition is of the form

  dp[i] = **op** {j = 1 to i - 1} of (c[j] + f[i] * g[j])

  where **op** is either max or min, then consider using convex hull trick. (Correspondingly, if **op** is max, then you maintain the upper hull instead.)

What have we learnt from Example 3?

- Even more generally, if your transition is of the form

  $$dp[i] = \mathbf{op}\ \{j = l[i] \text{ to } r[i]\} \text{ of } (c[j] + f[i] * g[j])$$

  where **op** is either max or min, we can also use a data structure (with a similar spirit to segment tree optimisation in Example 1) to speed up.
- A possible candidate would be an (extended) Li-Chao Tree. You can do range chmax line, and query range maximum.
- Unfortunately this lecture is about DP, not data structures. You may refer to the implementation details at https://codeforces.com/blog/entry/86731. Be aware that this incurs an extra log factor, so updates and queries are $O(\log^2 N)$ each.

What have we learnt from Example 3?

- In fact, this method works even if the transitions cannot be expressed as equations of straight lines. The only condition that we actually require is that it can be maintained effectively by an std::set. A necessary condition is that "the dominating section" of every transition has to be either empty or an interval.
- A sufficient condition is that for every pair of indices $i \neq j$, if we denote their transitions formula in terms of a dp state x to be $f_i(x)$ and $f_j(x)$, then:
  - $f_i(x) \geq f_j(x)$ for all subsequent x (or vice versa, i.e. $f_i(x) \leq f_j(x)$ for all subsequent x), or
  - There exists a constant c so that $f_i(x) \geq f_j(x)$ for all subsequent $x \leq c$, and $f_j(x) \geq f_i(x)$ for all subsequent $x > c$ (or vice versa).
    (Why?)

  (Exercise / hint: deduce why this trick is named "*convex* hull")

Example 3 Revisited — (A312) DP Optimization: Convex Hull Trick

- We do this problem again, using another perspective. This time, instead of doing it geometrically, we do it algebraically.

$$dp[i] = \min \{j = 1 \text{ to } i - 1\} \text{ of } (dp[j] + B[j] * A[i])$$

- A very intuitive question to ask is the following:
  Suppose we have two possible candidates of indices j to transition from; let's call them $j_0 < j_1$. Should we choose to transition from j_0 or j_1 to i?

- If we choose j_0, then dp[i] is $dp[j_0] + B[j_0] * A[i]$.
- If we choose j_1, then dp[i] is $dp[j_1] + B[j_1] * A[i]$.
- We then try to solve the inequality.

Example 3 Revisited — (A312) DP Optimization: Convex Hull Trick

- We then try to solve the inequality, where $j_0 < j_1$ are distinct indices $< i$:

$$\begin{aligned}
& \text{index } j_1 \text{ is not worse than than index } j_0 \\
\Leftrightarrow\ & dp[j_1] + B[j_1] * A[i] \le dp[j_0] + B[j_0] * A[i] \\
\Leftrightarrow\ & dp[j_1] - dp[j_0] \le B[j_0] * A[i] - B[j_1] * A[i] \\
\Leftrightarrow\ & dp[j_1] - dp[j_0] \le -(B[j_1] - B[j_0]) * A[i] \\
\Leftrightarrow\ & (dp[j_1] - dp[j_0]) \div (B[j_1] - B[j_0]) \ge -A[i]
\end{aligned}$$

  in which the last step is correct because we are given that the array B[] is strictly decreasing and we assumed that $j_0 < j_1$.
- Alright, but what does this mean?

Example 3 Revisited — (A312) DP Optimization: Convex Hull Trick

- Index j_1 is not worse than index j_0 ⇔ $(dp[j_1] - dp[j_0]) \div (B[j_1] - B[j_0]) \geq -A[i]$
- If we let point P[j] to have coordinates (B[j], dp[j]), then the above formula says that the slope of the line between the two points $P[j_0]$ and $P[j_1]$ has to be ≥ -A[i].
- Recall the array A[] is given to be strictly increasing, so in other words -A[] is strictly decreasing.
  ⇒ Once j_1 is better (or not worse) than index j_0, it will remain to be better than index j_0 forever in the future.
- Does this remind you of any other DP optimisation techniques?

Example 3 Revisited — (A312) DP Optimization: Convex Hull Trick

- "*Once j_1 is better (or not worse) than index j_0, it will remain to be better than index j_0 forever in the future.*"
  This is precisely the condition needed for a monotonic queue!

- How do we implement this?

- Question: explain why convex hull trick optimisation is sometimes also known as the "slope trick".

Example 3 Revisited — (A312) DP Optimization: Convex Hull Trick

- Initialise dp[1] = 0 and a deque = {1}.
- For every 2 ≤ i ≤ N,
  - While the slope formed by the two frontmost elements in the deque forms a slope ≥ -A[i], pop the frontmost element.
  - Set dp[i] = dp[deque.front] + B[deque.front] + A[i].
  - While the slope formed by the two backmost elements in the deque is larger than the slope formed by the backmost element and the point (B[i], dp[i]), pop the backmost element (to maintain convexity).
  - Push i into the back of the deque.
- The same working principle as our previously discussed monotonic queue.

A Short Reflection: Two Approaches to Example 3

- Both the geometric approach and the algebraic approach eventually simplify to using a monotonic queue to maintain the list of possible candidates for optimal transition points and work in O(N) time.
- The points in the algebraic approach forms a “convex hull” too! (Why?) However, note that the two constructed hulls in general are **not** the same.

- Question: we saw earlier that the geometric approach works even when A is not monotonic. Can we modify our algebraic approach to make this work for A that is not monotonic as well? If yes, how? If no, why not?
- Answer the same question again, but we are now interested in situations where the array B is not monotonic, instead of A.

Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- There is a rectangular grid of size N * M. You want to walk from cell (1, 1) to cell (N, M) by going right and down only with the shortest time:
  - It takes A[i] seconds to go rightwards from cell (i, j) to (i, j + 1).
  - It takes B[j] seconds to go downwards from cell (i, j) to (i + 1, j).

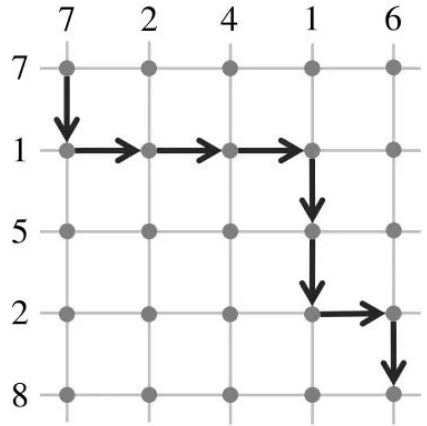
- Immediately we can come up with a O(NM) solution like this:
- Let dp[i][j] be the shortest possible time to walk from cell (1, 1) to (i, j). Then dp[i][j] = min(dp[i - 1][j] + B[j], dp[i][j - 1] + A[i]).

- Obviously, we now ask: how can we optimise this approach?

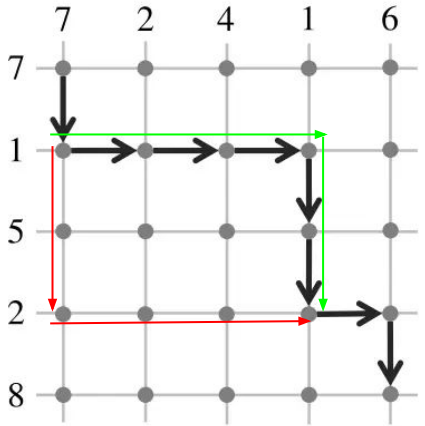
Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- It doesn't seem clear that a geometric approach can be used immediately, what about an algebraic approach?
- We take a closer look at the local structure of an optimal solution. What does an optimal solution look / behave like?

- In any path from (1, 1) to (N, M), we can break the path down into blocks of maximal length of contiguous moves.
- For example, on the sample on the right, we can break it down into [1 down, 3 rights, 2 downs, 1 right, 1 down].



Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto



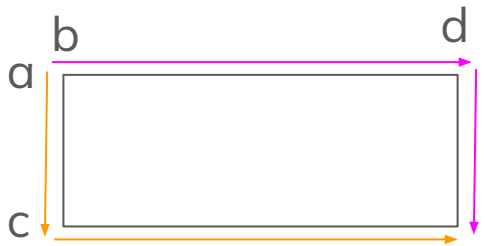
- Consider any two consecutive blocks of maximal length, e.g. [3 rights, 2 downs] from (2, 1) to (4, 4).
- Clearly one must be right and the other must be down.
- The fact that we took this path implies that the route [3 rights, 2 downs] is not worse than the route [2 downs, 3 rights]. Algebraically, this means

$$3 * A[2] + 2 * B[4] \le 2 * B[1] + 3 * A[4]$$
$$\Leftrightarrow \quad 2 * (B[4] - B[1]) \le 3 * (A[4] - A[2])$$
$$\Leftrightarrow (B[4] - B[1]) \div (4 - 1) \le (A[4] - A[2]) \div (4 - 2)$$

- Does this ring a bell?

Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- Before directly moving on, let's formally verify what we did just now. We shall make sure this makes sense for the general case.
- Suppose we are at cell (a, b) and after two maximal block moves, we arrive at (c, d). Here a < c and b < d.



- If we use the orange path, the cost is (c - a) * B[b] + (d - b) * A[c]. If we use the purple path, the cost is (d - b) * A[a] + (c - a) * B[d]. So the orange path is better than the purple path if and only if

$$(c - a) * B[b] + (d - b) * A[c] < (d - b) * A[a] + (c - a) * B[d]$$
$$\Leftrightarrow (d - b) * (A[c] - A[a]) < (c - a) * (B[d] - B[b])$$
$$\Leftrightarrow (A[c] - A[a]) \div (c - a) < (B[d] - B[b]) \div (d - b)$$

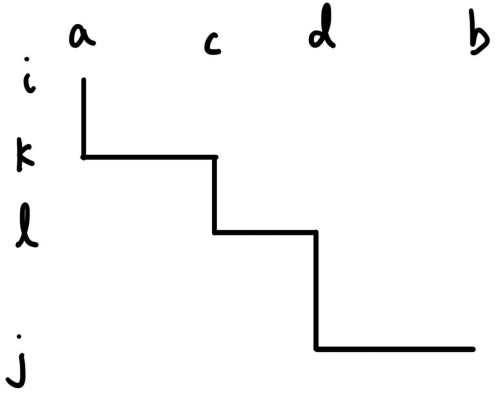
Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- ... ⇔ (A[c] - A[a]) ÷ (c - a) < (B[d] - B[b]) ÷ (d - b)
- This form looks similar to what we did in the algebraic version of convex hull trick technique, when we constructed points with given coordinates so that the slope between points resemble the inequality condition.
- For convenience, we denote this assertion as P(a, b, c, d).

- The above form (almost literally) screams at us to build two convex hulls on the points (i, A[i]) and (i, B[i]) respectively. But why? ... and what next?
- Perhaps we do a bit more algebra to find out.

Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

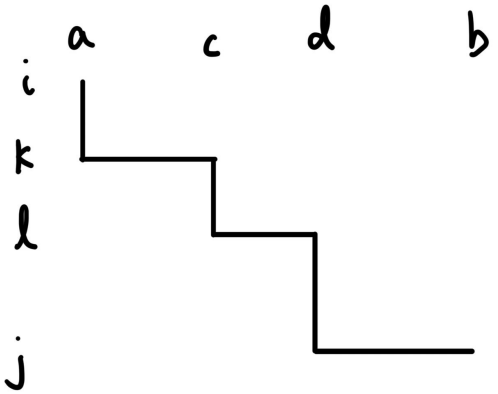
- Consider two rows i < j.
- **Claim:** There exists a path for which we never need to move rightwards on all rows k satisfying i < k < j, and that the point (k, A[k]) lies on or above the segment connecting the points (i, A[i]) and (j, A[j]).
- Proof: suppose the contrary that such a path exists.
- By the assertion P(i, a, k, c), we know that $(A[k] - A[i]) \div (k - i) \le (B[c] - B[a]) \div (c - a)$.
- By the assertion P(k, a, l, c), we know that $(B[c] - B[a]) \div (c - a) < (A[l] - A[k]) \div (l - k)$.
- ... and so on, obtaining a chain of inequalities.



Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- In particular, if we focus on the terms related to A, we see that $(A[k] - A[i]) \div (k - i) < (A[l] - A[k]) \div (l - k) < (A[j] - A[l]) \div (l - j)$.
- However, this contradicts our assumption that point (k, A[k]) lies on or above the segment between (i, A[i]) and (j, A[j]). So the claim is proved.

- Symmetrically, we can obtain an identical result about columns.
- Therefore, there exists an optimal path whose rows and columns of its "turning points" are all on the lower hull formed by the points (i, A[i]) and (i, B[i]) respectively. (Why?)



Example 4 — (JOISC 2022 Day 1 Problem 2) Sightseeing in Kyoto

- Recall (and/or note) that the slopes of the line segments on a lower hull are strictly increasing from left to right (as x-coordinate increases).
- By an easy exchange argument, we can greedily do the following:
  - If moving to the next point in the hull of (*, A[*]) gives a smaller slope, then move downwards to the row corresponding to the next point of the convex hull.
  - Otherwise, move rightwards to the column corresponding to the next the next point of the convex hull of (*, B[*]).
- Since constructing the lower hulls in this problem can be done in linear time with a monotonic stack, we have solved this problem with O(N + M) time complexity.

What have we learnt from Example 4?

- If a DP transition looks unappealing to be optimised at first, try exploring local properties using algebraic approaches first.
- Once you developed an expression which resembles the slope of two points, try to relate it with convexity and fully explore its properties.
- In fact, what we did is actually equivalent to computing the lower hull of the Minkowski sum of the two hulls formed by (i, A[i]) and (i, B[i]). It is a well-known result that the (Minkowski) sum of two convex regions is also convex.
- Question: how does our solution to this example problem relate to the algorithm that finds the Minkowski sum of two convex regions?

Example 5 — (A313) DP Optimization: Divide and Conquer

- Partition {1, 2, ..., N} into K non-empty contiguous subarrays. For every pair of $1 \le i < j \le N$, if i and j are in the same subarray, then add A[i][j] to the total cost. Find the minimum possible cost over all partitions.

- Let dp[i][j] be the minimum answer if we partition the first j people using i subarrays. Then dp[0][0] = 0, our answer is dp[k][n], and transition is:

  dp[i][j] = min {k = 0 over j - 1} of (dp[i - 1][k] + sumA[k + 1][j])

- We have O(NK) states, and each state takes O(N) time.
- What are some possible ways to motivate an optimisation here?

Example 5 — (A313) DP Optimization: Divide and Conquer

- dp[i][j] = min {k = 0 over j - 1} of (dp[i - 1][k] + sumA[k + 1][j])
- We denote $k_{i,j}^{opt}$ as the optimal index k that minimises the expression above (in general, we tiebreak by taking the smaller index).
- **Claim**: We always have $k_{i,j}^{opt} \leq k_{i,j+1}^{opt}$.
  Proof by Alex Tung
- Alright, but is there a better intuition without tons of algebra?
- Increasing j means extending the current segment to the right. However, the earlier the current segment started, the more it is being penalised. Visually, this is because the area of the square increases by something proportional to its current side length.

Example 5 — (A313) DP Optimization: Divide and Conquer

- We always have $k_{i, j}^{opt} \le k_{i, j+1}^{opt}$.
- With this in mind, we want to compute each layer (i is fixed) of values of dp[i][j] for all j efficiently. How can we achieve this?
- Divide and conquer works well:
  - Compute dp[i][N / 2] naively in O(N) time first.
  - Then, we can compute dp[i][N / 4] and dp[i][3 * N / 4] in O(N) time altogether, since $k_{i, n/4} \le k_{i, n/2} \le k_{i, 3*n/4}$.
  - Similarly, compute dp[i][N / 8], dp[i][3 * N / 8], dp[i][5 * N / 8], and dp[i][7 * N / 8] in O(N) time altogether, as the optimal k-s are nondecreasing.
  - ... and so on.

Example 5 — (A313) DP Optimization: Divide and Conquer

- Divide and conquer works well.
- Each round takes O(N) time, and there are log N rounds in total, so the complexity for calculating one layer of dp[i][*] is O(N log N). There are K layers, so the total time complexity is O(K * N log N).
- Very easy to write! Remember to handle the base cases dp[0][*] correctly.

```
void dnc(int i, int l, int r, int koptl, int koptr) {
  if (l > r) return;  // nothing to compute
  int mid = (l + r) / 2, best = INF, kopt;
  for (int k = koptl; k <= koptr && k < mid; k++) {
    int cost = dp[i - 1][k] + sumA[k + 1][mid];
    if (cost < best) {
      best = cost;
      kopt = k;
    }
  }
  dp[i][mid] = best;
  dnc(i, l, mid - 1, koptl, kopt);
  dnc(i, mid + 1, r, kopt, koptr);
}

// call dnc(i, 1, n, 0, n - 1) for each i from 1 to k
```

What have we learnt from Example 5?

- The condition that $k_{i,j}^{opt} \le k_{i,j+1}^{opt}$ is surprisingly useful. It allowed us to find all the optimal transition points $k_{i,*}^{opt}$ in O(N log N) time instead of $O(N^2)$.
- In general, there are a few ways to come up this important conclusion:
  - Draw proportionally sensible diagrams on your rough paper
  - Write a brute force to verify
  - Come up with the algebra yourself during contest (do you have enough time for this?)
  - Guess and submit (???)
- Question: compare the divide and conquer optimisation with the parallel binary search algorithm (整體二分). What similarities do they share?

What have we learnt from Example 5?

- In retrospect, what properties of the dp approach have we utilised?
- Divide and conquer optimisation only applies if $k_{i,j}^{opt} \le k_{i,j+1}^{opt}$.
- More generally, if our transition formula is of the form

  $$\min \{k = 0 \text{ over } j - 1\} \text{ of } (dp[i - 1][k] + cost(k + 1, j))$$

  Then a sufficient (but not necessary) condition for the inequality $k_{i,j}^{opt} \le k_{i,j+1}^{opt}$ to hold is the following (quadrangle inequality):

  $$cost(l, r + 1) - cost(l, r) \le cost(l - 1, r + 1) - cost(l - 1, r) \text{ for all } l \le r$$

- Intuitively, this means whenever we increment the right endpoint of a range, decrementing the left endpoint is never cheaper.

Example 6 — (IOI 2013 Day 1 Problem 3) Wombats

- There is a rectangular grid of size R * C ($R \leq 5000$, $C \leq 200$).
  - It takes A[i][j] seconds to go between cells (i, j) to (i, j + 1).
  - It takes B[i][j] seconds to go downwards from cell (i, j) to (i + 1, j).

  Multiple queries of the form [X, Y]. Find the shortest time to walk from cell (0, X) to cell (R - 1, Y) by going left, right and down only.

- Between every two neighbouring rows, we can construct a matrix dist[C][C], where dist[x][y] denotes the shortest time to go from column x of the upper row to column y of the lower row.

Example 6 — (IOI 2013 Day 1 Problem 3) Wombats

- Between every two neighbouring rows, construct a matrix dist[C][C], where dist[x][y] denotes the shortest time to go from column x of the upper row to column y of the lower row.

- Now what if we want to find the distance from (r, x) to (r + 2, z)?
  The answer is simply min {y = 0 to C - 1} of ($dist_r[x][y] + dist_{r+1}[y][z]$).

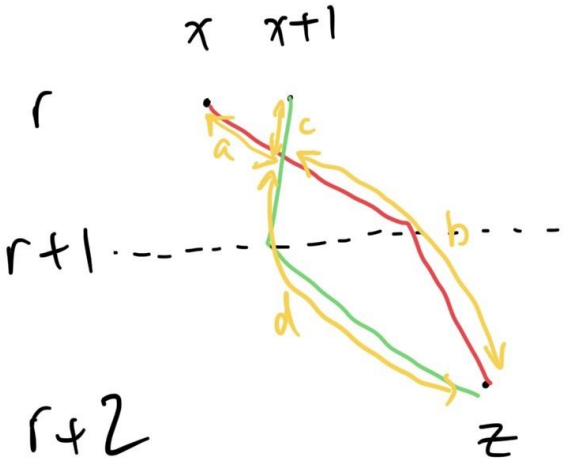
- Does this ring a bell? (Hint: recall that our target is to answer distances from cell (0, X) to cell (R - 1, Y).)

Example 6 — (IOI 2013 Day 1 Problem 3) Wombats

- $dist_{r\ to\ r+2}[x][z] = \min\ \{k = 0\ to\ C - 1\}\ of\ (dist_r[x][k] + dist_{r+1}[k][z])$
- This is generalised matrix multiplication (廣義矩陣乘法) using (min, +)!
- Consider a segment tree where the i-th leaf is the array $dist_i[][]$. This means we have to be able to combine two matrices to pushup.

  The naive approach is $O(C^3)$ which is definitely too bad for us.
- **Claim:** Fix z. As x increases, $k_{x,z}^{opt} \le k_{x+1,z}^{opt}$.
- Proof: Suppose the contrary. By the virtue of shortest paths, we have d ≤ b and b ≤ d.

  This quickly forces a contradiction.



Example 6 — (IOI 2013 Day 1 Problem 3) Wombats

- Therefore, we can do divide and conquer optimisation on this so that pushup is $O(C^2 \log C)$ each. Better, but still not good enough.
-
- By symmetry, one can similarly observe that if we fix x, and as z increases, $k_{x,z}^{opt} \le k_{x,z+1}^{opt}$.
- **Observation:** $k_{x,z}^{opt} \le k_{x,z+1}^{opt} \le k_{x+1,z+1}^{opt}$
- Therefore, if we iterate through the entries in increasing order of z - x, then we can perform achieve $O(C^2)$ time per pushup. So our segment tree works fast enough. Getting full score requires some tedious constant memory optimisation which is not of the interest of today's lecture.

What have we learnt from Example 6?

- Divide and conquer optimisation is not just compatible with DP; it can also be used to speed up other things like matrix (min, +) multiplication.
- Divide and conquer optimisation works not just for one dimension, but also the same for 2D (and has even better time complexity here)!

- Just like in Example 4, analysing the virtue of shortest paths and doing a bunch of exchange arguments to explore local properties often demonstrate to be useful.

Practice Problems

- (M1643 / APIO 2014 Problem 2) Split the Sequence
- (M1863) Osu!
- (M2343) The Chessmaster's Challenge
- (M2543) Bar Chart
- (NP2324) 天天愛打卡
- (T173) Secret Meeting
- (T231) Facility Location (up to 81 points, full solution requires some other optimisation techniques not covered in today's lecture)
- (IOI 2023 Day 2 Problem 2) Overtaking
- the challenging practice problems at the start of the slides
- … and, of course, all the problems discussed in today's lecture.