Dynamic Programming 2

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

DP Optimizations Overview

- Sometimes when you're doing DP your solution will be too slow but your recurrence is very structured.
- You've seen this before with ranges and range trees.
- This lecture will show a few more cases of structured recurrences where there are well known techniques for speeding them up.
- Again, this does not help if your state space is too large. This should not really change your overall approach to DP but you should keep an eye out for recurrences that can be sped up.

Table of Contents

- One of the easiest to apply and most useful optimizations.
- Handy when your recurrence is formed by linear functions.
- Though once you get good at spotting these, more things look like a linear function than you might expect!

- General setting is:
  - You are doing a 1D DP. Let's say you are calculating $dp[N]$ in order of increasing i.
  - You have an array $m[N]$. Think of these as gradients. The array $m[N]$ is in *decreasing order*.
  - You have an array $p[N]$. Think of these as the positions of the points for which you are calculating $dp[N]$.
- You have some base case value $dp[0]$ and the recurrence is:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Take a look at the recurrence:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Hopefully you see an $O(N^2)$ solution.
- What do those terms in the min look like?
- Equations for lines with $dp[j]$ as the y-intercept and $m[j]$ as the slope.

- Recurrence:

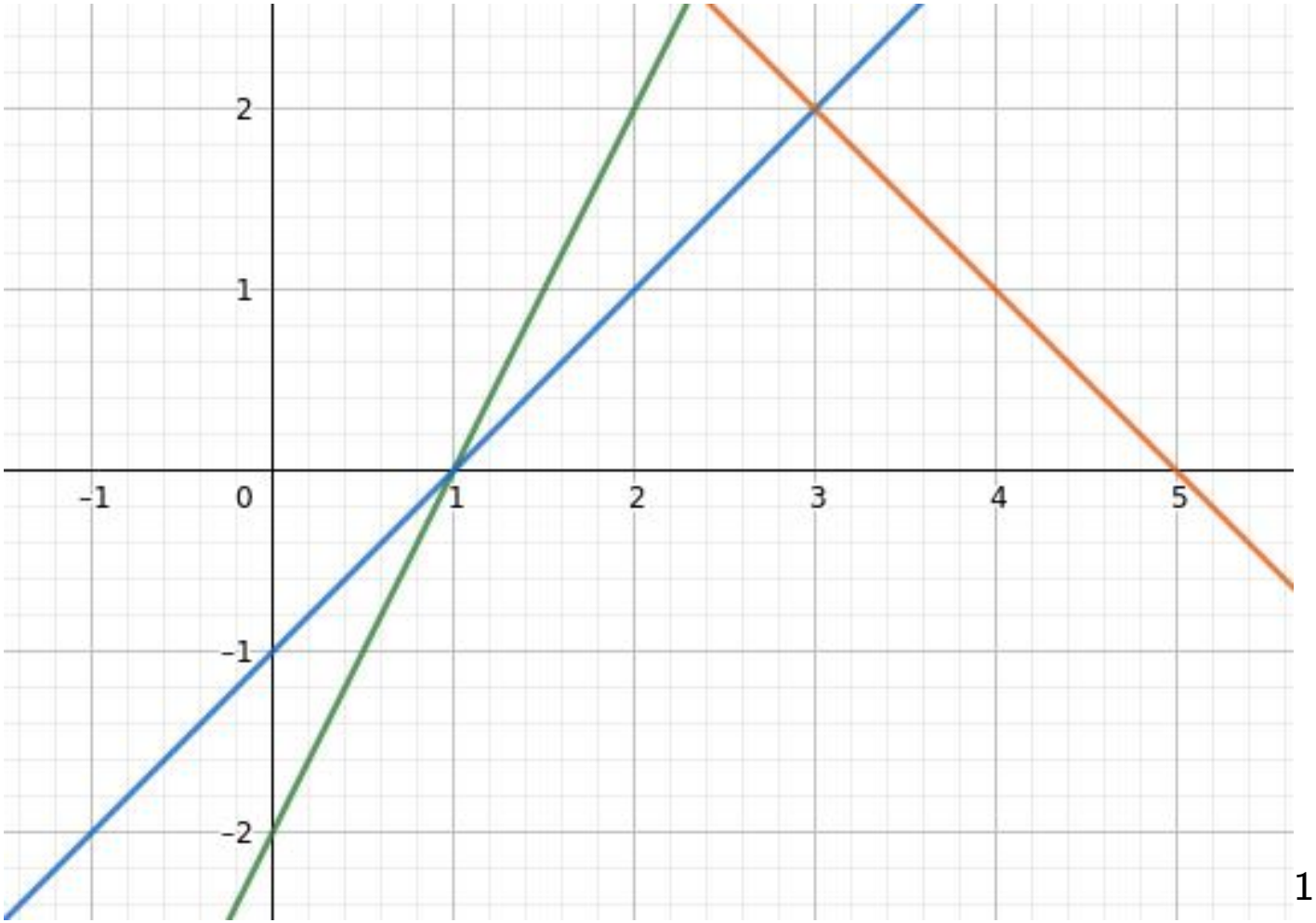
$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Suppose so far we have:
  - $dp[0] = -2, m[0] = 2$
  - $dp[1] = -1, m[1] = 1$
  - $dp[2] = 5, m[2] = -1$
- And we want to calculate $dp[3]$ (leave $p[3]$ unfixed for now).
- Then $dp[i]$ is the minimum of a set of lines at the x coordinate $p[3]$ where our lines are:
  - $y = m[0]x + dp[0] = 2x - 2$
  - $y = m[1]x + dp[1] = x - 1$
  - $y = m[2]x + dp[2] = -x + 5$
- So far we have done this in $O(N)$ for each $dp[i]$ calculation. We will see how to speed this up.

- To speed up this recurrence, we will build a data structure that exactly supports the operations we need.
- It will export 2 methods:
  - `void add(int m, int b)`: Add a line $l = mx + b$. **Requirement:** m is strictly less than the gradient of all lines in the data structure already.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return $\min_i(m_i q + b_i)$.
- We will have `add` run in $O(1)$ *amortized* and `query` run in $O(\log n)$ where n is the number of lines added so far.
- Afterwards we will apply this data struture as a black box to some DP problems.

- To get the right complexity, we need some observations first regarding the set of lines we are querying.
- For concreteness, let us return to our earlier example. Suppose we have:
  - $b[0] = -2, m[0] = 2$
  - $b[1] = -1, m[1] = 1$
  - $b[2] = 5, m[2] = -1$

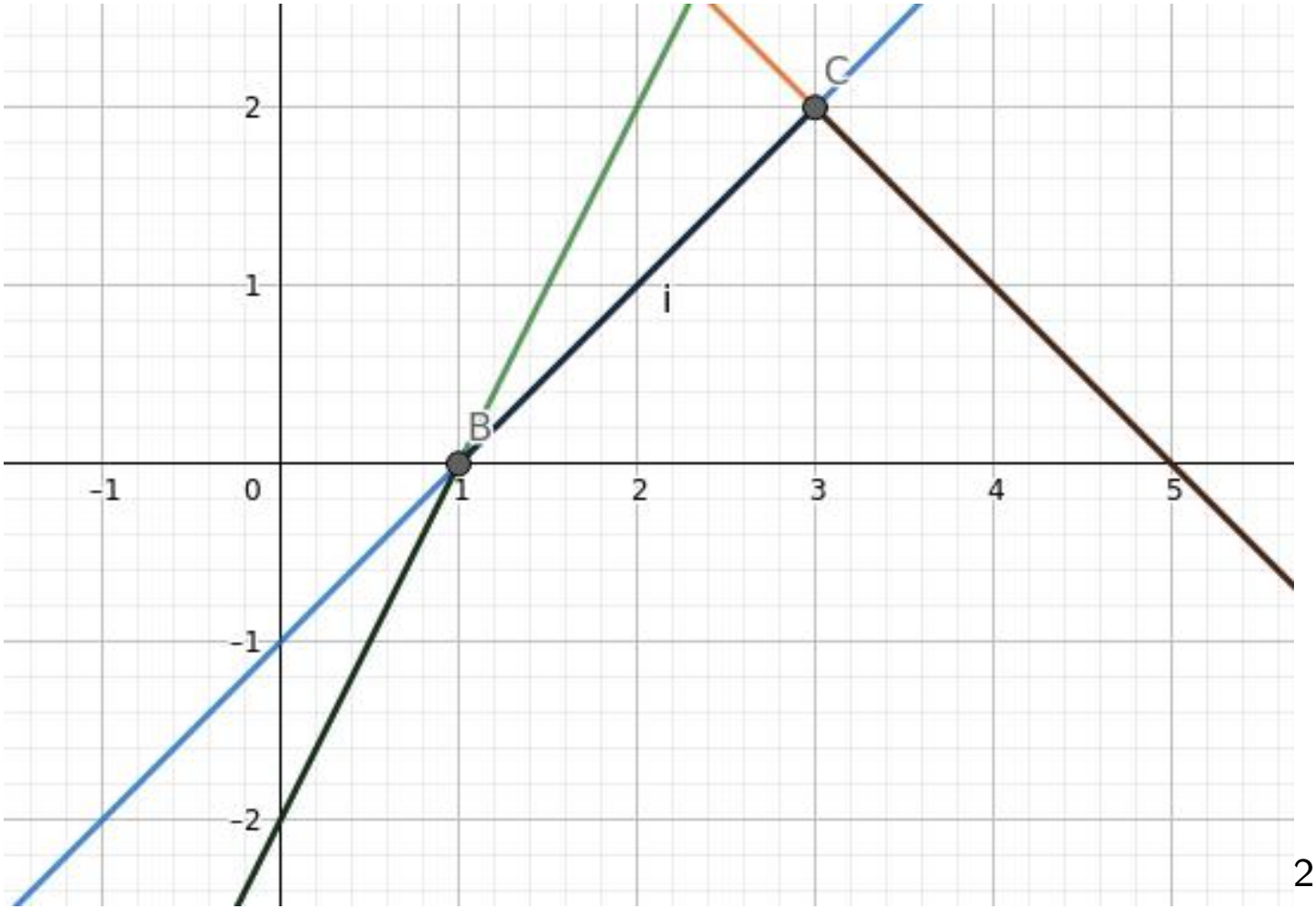
  (where $m[i]$ are the gradients and $b[i]$ are the y-intercepts)



[1]

[1] https://www.geogebra.org/graphing

- Remember our aim is to query the minimum over these 3 lines at the x coordinate q.
- What could this possibly be?
- Let's emphasize what the minimum value is at each x coordinate.



[2]

[2] https://www.geogebra.org/graphing

- **Key Observation:** This is the upper convex hull of the areas below the lines.
- This means each line is optimal for a contiguous range of x (possibly empty) and as we move left to right, the gradient of the optimal line never increases.
- Our goal is to maintain the convex hull by maintaining the set of line segments that the convex hull is made out of.
- We will say a line l is dominant at x-coordinate x if it is the line that forms the convex hull at x. This is the line that gives us our minimum in the equation

$$query(q) = \min_i(m_i \cdot q + b_i)$$

- For this, it suffices to store the lines that make up the convex hull in left to right order. (same as in decreasing gradient order).
- **Note:** Importantly, this does not contain all the lines. It omits any line that is never in the upper convex hull.
- Given this data, we can calculate the range of x at which each line *l* is dominant. The segment is the range between the intersection of *l* with the line before it, and after it, in the convex hull.
- We keep the lines in a vector.

CHT Data Structure Construction


```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;
/*
 * The intersection points are
 *  intersect(cht[0], cht[1]), intersect(cht[1], cht[2]), ...
 *  Line i is optimal in the range
 *      [intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]
 *  where for i = 0, the first point is -infinity,
 *  and for i = N-1, the last point is infinity.
 */
```

- We keep the lines in a vector.
- Recall the 2 methods our data structure is meant to support are:
  - What's the line on the upper convex hull at $x = q$? (Equivalently, what is $\min_i(m_i \cdot q + b_i)$ over all the lines)
  - Add the line $y = m[i] * x + dp[i]$.
- We handle the former with binary search.

- Our aim is to find the line that is dominant at $x = q$.
- Suppose our lines are ordered in decreasing gradient order. Then recall the range at which line l_i is dominant is

$$[intersect(l_{i-1}, l_i), intersect(l_i, l_{i+1})]$$

  where for $i = 0$, the left term is $-\infty$ and for the last line, the right term is $+\infty$.
- So to find the i for which the segment contains q, it suffices to find the minimum i such that

$$intersect(l_i, l_{i+1}) \geq q$$

CHT Query Implementation

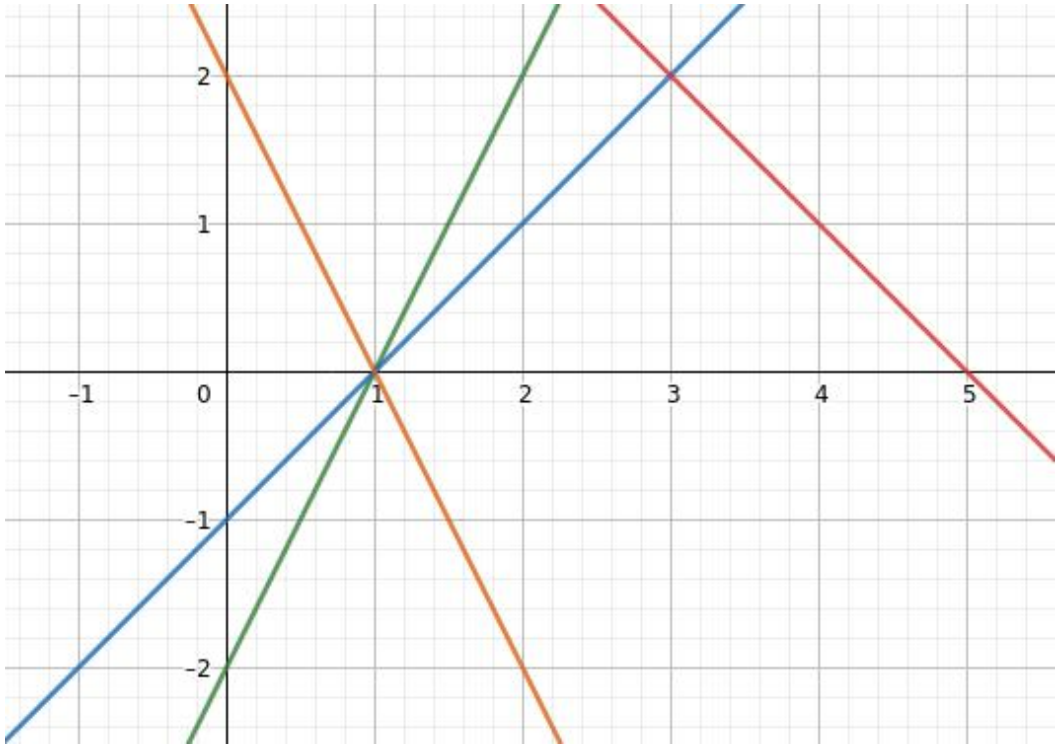

```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;

/* Recall that the range the ith line is dominant in is:
 * [intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]
 * We want to find the line that is dominant at x.
 * To do this, we note that the sequence (intersect(cht[i], cht[i+1]))
 * is monotonically increasing in i.
 * Hence we can binary search for the minimum i such that
 * intersect(cht[i], cht[i+1]) >= x
 */
long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    // Find largest idx such that x <= intersect(cht[idx], cht[idx+1])
    // If this doesn't exist then idx should be cht.size()-1.
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

CHT Add Implementation

- To add a line we crucially use the fact that $m[N]$ is in decreasing order.
- So the new line has to go on the end of our convex hull (recall the convex hull is sorted in non-increasing order of gradients).
- However, this may cause some lines to disappear from the convex hull.

CHT Add Implementation

- For example, consider adding the line $y = -2x + 2$ to the earlier example:
- 

- What is the new convex hull?

CHT Add Implementation

- So some of the lines may become useless and we need to remove them.
- When does a line l become useless? When the line we just added covers the entire range l is dominant in.
- **Observation:** The useless lines are always at the end of the convex hull.
- So we just need to keep popping the last line of the convex hull as long as it is useless.
- How do we check if the last line is useless?
- For this, it helps to draw pictures and move your new line l around.

CHT Add Implementation

- You will hopefully observe some variant of the following: Letting $cht[N-1]$ be the last line, it is useless if:

  $$intersect(cht[N-1], l) \leq intersect(cht[N-2], cht[N-1])$$

- Recall the range that $cht[N-1]$ is dominant in is $(intersect(cht[N-2], cht[N-1]), \infty)$.
- The above essentially says l is better than $cht[N-1]$ for this entire range.
- Another way of phrasing this is that the intersections $(intersect(cht[i], cht[i+1]))$ need to be kept in increasing order.

CHT Full Implementation


```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;

void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
           intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
}

long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    // Find largest idx such that x <= intersect(cht[idx], cht[idx+1])
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

- **Complexity?** $O(1)$ *amortized* per add. $O(\log n)$ per query.
- But do *remember*, we did assume the gradients are in decreasing order. It is non-trivial to remove this assumption.
- In certain special cases, we can actually get a better complexity!

CHT Data Structure Modifications

- One common case, each query we make is at least the previous query we make. (formally, say our code calls query with $query(q_1), query(q_2), \ldots, query(q_Q)$. Then $q_1 \leq q_2 \leq \ldots \leq q_Q$)
- In this case, the index of the dominant line for each q_i only increases.
- So we keep track of the index of the dominant line. Whenever we query, we check if the current dominant line is still the dominant line for the new query point.

- If cp is the current dominant line, this amounts to checking if

$$p[i] \leq intersect(cht[cp], cht[cp+1])$$

- While this does not hold, increase cp.
- The above had an omission. There is a special case. Since the number of lines in the convex hull can decrease, cp may be out of bounds. To fix this, after each update we just need to update cp to point to the last line if it is out of bounds.
- If you want to rigorously check this, the invariant you are maintaining is

$$intersect(cht[cp-1], cht[cp]) \leq p[i]$$

CHT Data Structure Modifications


```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;
int cp;

void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
           intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
    cp = min(cp, (int)cht.size()-1);
}

long long query(long long x) {
    while (cp+1 != cht.size() &&
           intersect(cht[cp], cht[cp+1]) < x) cp++;
    return cht[cp].m*x + cht[cp].b;
}
```

- **Complexity?** Now updates and queries are both $O(1)$ amortized.

More Modifications

- Currently query returns the minimum over all lines. We can also instead return the maximum. If so, our invariant is that our gradients should be increasing. We are now calculating the lower convex hull of the area above all lines.
- But we can actually use the exact same code! (literally no modifications needed)

More Modifications

- Not hard to adjust for gradients non-increasing (instead of decreasing). Just one extra special case.
- Can also have no conditions on the gradients. We still keep lines in sorted gradient order. However, we now need to keep lines in a set since insertions are arbitrary. Not that common mostly because it is tedious to get right.
- Alternatively we can also use a "Li Chao tree".

More Modifications

- Another useful modification is when everything (gradients and query points) is in integers.
- Currently we use doubles to compute the intersection points. This gives us the precise ranges at which we dominate (up to precision errors) $[intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]$.
- Doubles are not precise though even for the range of a long long. Usually not problematic for competitions since numbers generally only go up to 10^9 but you have to keep it in mind if you need to query higher.
- If we only care about integer coordinates of x then we can round these down and work entirely in integers:

$$(\lfloor intersect(cht[i-1], cht[i]) \rfloor, \lfloor intersect(cht[i], cht[i+1]) \rfloor]$$

More Modifications

- But if we only care about integer coordinates of x then we can round these down:

  $$(\lfloor intersect(cht[i-1], cht[i]) \rfloor, \lfloor intersect(cht[i], cht[i+1]) \rfloor]$$

- Just **be careful**. If you want to do this, **make sure** it is clear to you why the above is exclusive-inclusive.
- In integers, whether your inequalities are strict or not actually matters. When you compare with an intersection point, mentally check if it agrees with the above ranges.
- If we only care about *positive* integers, we can omit the floor and just use regular integer division. But if you care about negatives, **note that** integer division isn't floor, it rounds towards 0.

CHT without floats


```
struct line { long long m, b; };
long long floordiv(long long a, long long b) {
    return a / b - (a%b && ((a<0) ^ (b<0)));
}
long long intersect(line a, line b) {
    floordiv(b.b - a.b, a.m - b.m);
    // for POSITIVE ints: can do:
    // return (b.b - a.b) / (a.m - b.m);
}
vector<line> cht;
void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
            intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
}
long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        // NOTE: It is critical here that this is >= not >.
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

- The above construction gives us exactly the data structure we needed.
- As a reminder, it supports 2 methods:
  - `void add(int m, int b)`: Add a line $l = mx + b$.
    Requirement: m is strictly less than the gradient of all lines in the data structure already.
    Complexity: $O(1)$.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return

$$\min_i m_i q + b_i$$

    Complexity: $O(\log n)$ in general, we can make it $O(1)$ if queries are given in non-decreasing order.

- There are 2 knobs we can tweak for our data structure.
  - If queries are in non-decreasing order, we can replace our query code with an $O(1)$ routine.
  - If everything is in integers, we can use integer division.
- However, aside from these, we can essentially use it as a black box!
- So the construction is good to know, but application wise, you can just copy paste it in if need be.

- Let us return to our earlier DP problem. We had the recurrence:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

  where we assumed $m[j]$ is in decreasing order.
- Since $m[j]$ is decreasing, we fulfil the one requirement of our CHT data structure. Hence we can directly apply convex hull trick as a black box.

```
int main() {
    // Base case:
    dp[0] = baseCaseCost;
    line l;
    l.m = m[0];
    l.b = dp[0];
    add(l);

    for (int i = 1; i < N; i++) {
        dp[i] = query(p[i]);
        line l;
        l.m = m[i];
        l.b = dp[i];
        add(l);
    }
    return 0;
}
```

- **Complexity?** $O(N \log N)$.
- This is a significant improvement from $O(N^2)$!
- Note, we could even get $O(N)$ if our $p[i]$ are non-decreasing.

- DPs with recurrences in the form

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

  can be done in $O(N \log N)$.
- This is done by creating a data structure for CHT which supports:
  - `void add(int m, int b)`: Add a line $l = mx + b$. **Requirement:** m is strictly less than the gradient of all lines in the data structure already.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return $\min_i(m_i q + b_i)$.
- You can treat this data structure as a black box. However it is good to at least know how to tweak it using the 2 knobs mentioned above.

In Practice

- In practice, the difficulty of CHT comes from:
  - Recognizing it is useful. Often you just have to write out the recurrences. Be suspect whenever the recurrence is given by a formula.
  - Figuring out how to calculate the gradients and y intercepts. You will often have to juggle around terms in the recurrence to make it work. This comes mostly with practice.
  - Essentially, the recurrence

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

tells us that we need the gradient to be a function of only j and the query point to be a function of only i.

Example: Covered Walkway

- **Problem Statement:** I have N points on a walkway I need to cover. To cover the walkway from point x to point y inclusive costs $C + (x - y)^2$ where C is a fixed, given constant. Note that you can cover a single point with cost C. What is the minimum cost needed to cover all the points?
- **Input Format:** First line 2 integers, $N, C, 1 \leq N \leq 10^6, 1 \leq C \leq 10^9$. The next N lines contain the points in increasing order. All points are in the range $[1, 10^9]$.
- **Source:** 2012 University of Chicago Invitational Programming Contest.

- Hope you can see a $O(N^2)$ DP.
- Calculate $dp[N]$, where $dp[i]$ is the min cost to cover exactly the points up to the ith.
- But the recurrence here is a very nice formula. Let us unpack it.

$$dp[j] = \min_{i<j} dp[i-1] + C + (x[j] - x[i])^2$$
$$= \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- Focus on the last term. What does this look like?
- A linear function (if you squint hard enough)! So we should try to fit this into our CHT framework.

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- If you recall, we were earlier looking at recurrences of the form $\min_{i<j} dp[i] + m[i] * p[j]$.
- What should $m[i]$ and $p[j]$ be?
- We should have $m[i] = -2x[i]$, $p[j] = x[j]$ (since we need the gradient to be determined by i and the query point by j).
- Note that our gradients are decreasing which we need for our CHT!

Example: Covered Walkway

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- We also need to modify the y-intercept of the line, it isn't just $dp[i]$ anymore. What should it be?
- **Key:** It needs to include all terms dependent on i. So it needs to include $dp[i-1]$ and $x[i]^2$. **Why?**
- But it can't include $x[j]^2$ for obvious reasons. So we should add the $x[j]^2$ part when we calculate $dp[j]$.
- You can choose whether to add C when you calculate $dp[j]$ or whether to add C to the y-intercept.

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- So for i we will add a line where:
  - y-intercept is: $dp[i-1] + x[i]^2$.
  - Gradient is: $-2x[i]$.
- And to calculate $dp[j]$, we query our CHT with point $x[j]$. Define $r := query(x[j])$. Then $dp[j] = r + C + x[j]^2$.

Example: Covered Walkway


```
/* Insert CHT code here: You can use the version with only
 * positive integer queries and non-decreasing queries */
void add(line l);
long long query(long long x);

const int MAXN = 1000000;
int N;
long long x[MAXN+1];
long long C;

long long dp[MAXN+1];

int main() {
    scanf("%d %lld", &N, &C);
    for (int i = 0; i < N; i++) {
        scanf("%lld", &x[i]);
    }
    for (int i = 0; i < N; i++) {
        // Compare to formulas written in slides
        dp[i] = query(x[i]) + C + x[i]*x[i];
        line l;
        l.m = -2*x[i];
        // Base case is i == 0, dp[i-1] = 0.
        l.b = (i == 0 ? 0 : dp[i-1]) + x[i]*x[i];
        add(l);
    }
    // Again, calculating dp[N-1] using the same formula as above.
    printf("%lld\n", dp[N-1]);
}
```

- **Complexity?** We note our query points, $p[j] = x[j]$ are increasing in j. So we can use the $O(1)$ amortized CHT. Then the complexity is just $O(N)$.
- Since our query points are integers, we can also use the version of CHT with no doubles. Using just integer division doesn't change the complexity but in practice is a large speed up (I get a 3 times speedup locally).

- This is how many CHT problems go.
  - First you come up with a normal DP that is too slow. This step is the same as in the DP lecture.
  - Then you note the recurrence is a nice formula. Write it out.
  - Now, split up the recurrence as in this example. Figure out which part corresponds to the slope and query point and how to split up the constant into the y intercept and the part added when you calculate $dp[j]$.
  - Usually this is just breaking up the terms depending on if they depend on j or i.

- **Problem Statement:** I have N soldiers in a line. The ith soldier has effectiveness x_i. I want to partition the soldiers into squadrons. Each squadron consists of a subsegment of *contiguous* soldiers. Say these soldiers have total effectiveness S. Then the effectiveness of the squadron is $AS^2 + C$ where $A < 0$ and C are given constants. Maximize sum of the effectiveness of the squadrons.
- **Input Format:** First line 2 integers, N, A, C. $1 \leq N \leq 10^6, -5 < A < -1, |C| \leq 10^4$. Next line contains N integers, the effectiveness of the soldiers in order. Each effectiveness satisfies $1 \leq x_i \leq 100$.
- **Source:** APIO 2010.

Example: Mini Commando

- **Sample Input:**

  ```
  4
  -1 -20
  2 2 3 4
  ```

- **Sample Output:** -101
- **Explanation:** Split the soldiers into groups of $\{1, 2\}, \{3\}, \{4\}$, with effectiveness $\{-36, -29, -36\}$.

- Hope you can see a $O(N^2)$ DP.
- Calculate $dp[N]$, where $dp[i]$ is the min cost to partition exactly the first i soldiers.
- Where to go next?
- Look at the recursion!

$$dp[j] = \max_{i<j} dp[i-1] + C + A(x_i + x_{i+1} + \ldots + x_j)^2$$

- This is kind of messy but should remind you a bit of the last example.

- **Useful trick:** We want to clean up the sum in the square. It is often easier to work with cumulative arrays than sums.
- Define $S[N]$ to be the cumulative sum, $S[i] := (x_1 + \ldots + x_i)$.
- Then let's rewrite the recurrence.

$$\begin{aligned} dp[j] &= \max_{i<j} dp[i-1] + C + A(S[j] - S[i-1])^2 \\ &= \max_{i<j} dp[i-1] + C + A \cdot S[i-1]^2 + A \cdot S[j]^2 \\ &\quad - 2A \cdot S[i-1] \cdot S[j] \end{aligned}$$

- Much cleaner IMO.

- $$dp[j] = \max_{i<j} dp[i-1] + C + A \cdot S[i-1]^2 + A \cdot S[j]^2 - 2A \cdot S[i-1] \cdot S[j]$$
- What does this look like?
- The key is the last term. If you unpack the important parts, a line!

- $$dp[j] = \max_{i<j} dp[i-1] + C + A \cdot S[i-1]^2 + A \cdot S[j]^2 - 2A \cdot S[i-1] \cdot S[j]$$
- Again, main question is to convert this into $b[i] + m[i] * p[j]$ form. Let us do the latter part first.
- What is $m[i]$?
- $-2A \cdot S[i-1]$. So $p[j] = S[j]$. Gradient is increasing (since A is negative) which is correct for maximization problems.
- How about the constant? Which parts form $b[i]$ and which parts should we add directly to $dp[j]$?
- Again, we need $b[i]$ to include $dp[i-1] + A \cdot S[i-1]^2$. This time around we will also include C in $b[i]$.

- $$dp[j] = \max_{i<j} dp[i-1] + C + A \cdot S[i-1]^2 + A \cdot S[j]^2 - 2A \cdot S[i-1] \cdot S[j]$$
- So for i we will add a line where:
  - y-intercept is: $dp[i-1] + A \cdot S[i-1]^2 + C$.
  - Gradient is: $-2A \cdot S[i-1]$.
- And to calculate $dp[j]$, we query our CHT with point $S[j]$. Define $r := query(S[j])$. Then $dp[j] = r + A \cdot S[j]^2$.

Example: Mini Commando


```
/* Insert CHT code here: You can use the version with only
 * positive integer queries and non-decreasing queries */
void add(line l);
long long query(long long x);

const int MAXN = 1000000;
int N;
long long a, c;
long long S[MAXN+1];
long long dp[MAXN+1];

int main() {
    scanf("%d", &N);
    scanf("%lld %lld %lld", &a, &c);
    for (int i = 1; i <= N; i++) {
        long long x; scanf("%lld", &x);
        S[i] = S[i-1] + x;
    }
    for (int i = 1; i <= N; i++) {
        line l;
        l.m = -2*a*S[i-1];
        l.b = dp[i-1] + a*S[i-1]*S[i-1] + c;
        add(l);
        dp[i] = query(S[i]) + a*S[i]*S[i];
    }
    printf("%lld\n", dp[N]);
    return 0;
}
```

- This time around we want to maximize the value. However, our code for CHT works for both min (with decreasing gradients) and max (with increasing gradients) without modification.
- **Complexity?** We note our query points, $p[j] = S[j]$ are increasing in j. So we can use the $O(1)$ amortized CHT. Then the complexity is just $O(N)$.
- One should be a bit cautious here of overflows. Here $S[i]$ can go up to 10^8, but we take $S[i]^2$, and $dp[i]$ can go up to around 10^{13}. This is okay with long longs.
- Our y intercepts go up to around 10^{16}. So we're going to lose precision if we use doubles for our intercepts. I think this is okay given our query points only go up to 10^8 (it gets AC) but honestly I'm not sure.

Table of Contents

Divide and Conquer Optimisation

- Here the structure comes in the choices we make during the DP.
- The most common setting is a 2D DP.

- **Problem Statement:** You have N kids in a row. Each has a noisiness $s_i > 0$. You want to partition the kids into K rooms. Each room must contain an *contiguous* segment of the kids. The noisiness of a room containing kids $[i, j]$ is $(s_i + s_{i+1} + \ldots s_j)^2$ (because kids...).
  What is the minimum total noisiness that is attainable?
- **Input Format:** First line 2 integers, N, K.
  $1 \leq K \leq \min(100, N), 1 \leq N \leq 10^4$. Next line contains N integers, the noisiness of the kids in order. Each noisiness is in the range $[1, 1000]$.

- Hope you can see an $O(N^2K)$ DP.
- Our state will be $dp[k][i]$, the minimum noisiness to partition the first i kids into k groups.
- We will proceed in increasing k and increasing i order. Recurrence?
- $$dp[k][i] = \min_{j<i} dp[k-1][j] + (s_{j+1} + \ldots + s_i)^2$$
- Naively this recurrence is $O(N^2)$, there are $O(N)$ values of j and the sum takes $O(N)$ to calculate. How do we speed up this recurrence to $O(N)$?
- Cumulative sum!

Example Problem: Noisy Kids


```cpp
#include <bits/stdc++.h>
using namespace std;

const long long INF = 1e17;
const int MAXN = 10000;
const int MAXK = 100;
int N, K;
// 1-indexed. S[i] = sum(s_1 + ... + s_i)
long long S[MAXN+1];
long long dp[MAXK+1][MAXN+1];

int main() {
    scanf("%d %d", &N, &K);
    for (int i = 1; i <= N; i++) {
        long long s; scanf("%lld", &s);
        S[i] = S[i-1] + s;
    }
    for (int i = 1; i <= N; i++) dp[0][i] = INF;
    for (int k = 1; k <= K; k++) {
        for (int j = 1; j <= N; j++) {
            dp[k][j] = INF;
            for (int t = 0; t < j; t++) {
                dp[k][j] = min(dp[k][j],
                               dp[k-1][t] + (S[j]-S[t])*(S[j]-S[t]));
            }
        }
    }
    printf("%lld\n", dp[K][N]);
    return 0;
}
```

- Let us take a look at the recurrence again.

$$dp[k][i] = \min_{j<i} dp[k-1][j] + (s_{j+1} + \ldots + s_i)^2$$

- Let $opt[k][i]$ be the value of j at which the minimum is attained in the above recurrence. (if there's multiple, any will suffice)
- **Key Claim:**

$$opt[k][i] \leq opt[k][i+1]$$

- Proof later. Intuitively, one expects increasing i to move our groups to the right. If $opt[k][i+1] < opt[k][i]$ then the kth group for $i+1$ encompasses the kth group for i. This seems wrong because the cost of squaring increases quicker the larger the group gets.
- More formally, the key is x^2 is *convex*.

- This is the setting we work in. We have a 2D dp with recurrence

$$dp[k][i] = \min_{j<i} dp[k-1][j] + Cost(j+1, i)$$

- We define the array $opt[k][i]$ to be any value of j at which the above attains a minimum.
- And we know (through some voodoo or maths) that:

$$opt[k][i] \leq opt[k][i+1]$$

- We now see how to speed this up.

The structure of the following slides:

- In the next few slides, I assume the setting just mentioned. In particular, I assume that $opt[k][i] \leq opt[k][i+1]$ for all k, i.
- Under this setting, I will explain the general procedure for optimising the DP from $O(N^2K)$ to $O(NK \log N)$.
- I will demonstrate the procedure on our example problem.
- I will then talk about when $opt[k][i] \leq opt[k][i+1]$ holds. As a special case, we will show it holds for our example problem.

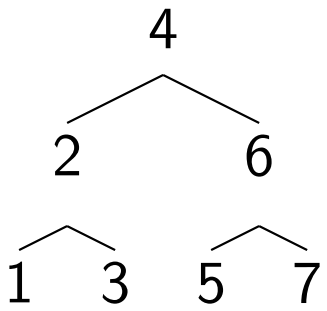
- Suppose $opt[k][i] \leq opt[k][i+1]$.
- The earlier recurrence for $dp[k][i]$ was $O(N)$ because we had to try every $j < i$.
- But if we know $prev := opt[k][i-1]$ then we only need to try every $j \in [prev, i)$.
- This doesn't lead to a speed up just yet. But for example, if we further know $nxt := opt[k][i+1]$ then we only have to try every $j \in [prev, nxt]$.
- This is how we will proceed. We will constrain the range we have to search on both sides.

Divide and Conquer Framework

- First we calculate $dp[k][N/2]$ and $opt[k][N/2]$. For convenience, define $o_{k,N/2} := opt[k][N/2]$.
- Now, for all $i < N/2$ we know $o_{k,i} < o_{k,N/2}$ and for all $i > N/2$ we know $o_{k,i} > o_{k,N/2}$. So on the left we have to search the range $[1, o_{k,N/2}]$ and on the right $[o_{k,N/2}, N]$.
- Note: this partitions the range between the 2 halves.
- We now divide and conquer by repeating this procedure in both halves, except only searching the above range in each of the halves.

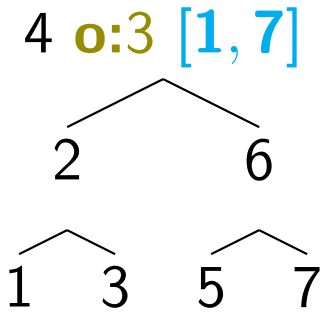
Divide and Conquer Framework

- Suppose we have k fixed and $N = 7$. Then we calculate $dp[k][n]$ in top to bottom order in the following tree.
- Each node will store its index i and $o_{k,i}$ in the form **o:**$o_{k,i}$.
- Each node will also the range in which it searched for its recurrence like **[x, y]**.



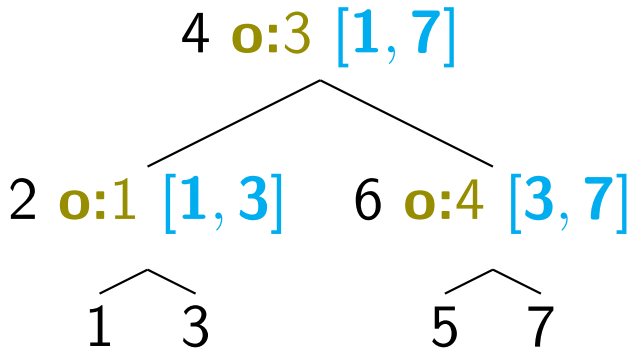
Divide and Conquer Framework

- First, we have to search the range $[1, 7]$ to find $dp[k][4]$ and $o_{k,4}$. Suppose $o_{k,4} = 3$:



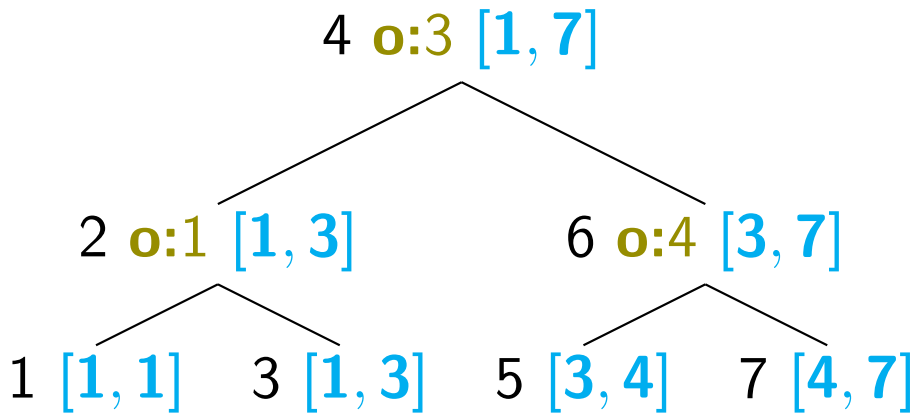
Divide and Conquer Framework

- Next we calculate $dp[k][i]$ for the 2 children. We should only search for j in the ranges that make sense. Suppose $o_{k,2} = 1$ and $o_{k,6} = 4$.



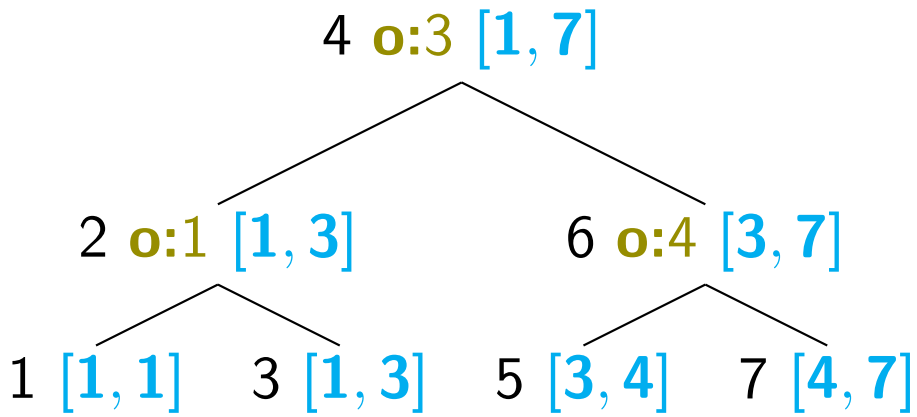
Divide and Conquer Framework

- Finally we calculate $dp[k][i]$ for the last layer, again only searching in the ranges that make sense. I am going to omit **o:**x for this layer because it does not matter.



Divide and Conquer Framework

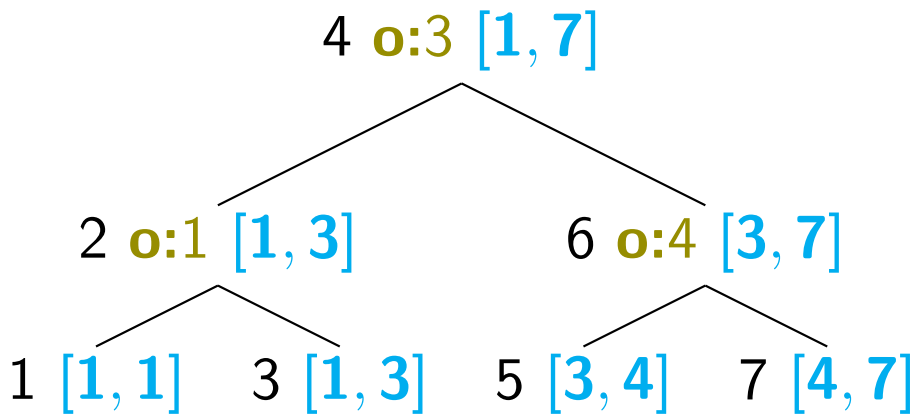
- What can you say about the search ranges of each layer of the tree?



- Besides overlaps at endpoints, they form a partition of $[1, 7]$.

Divide and Conquer Framework

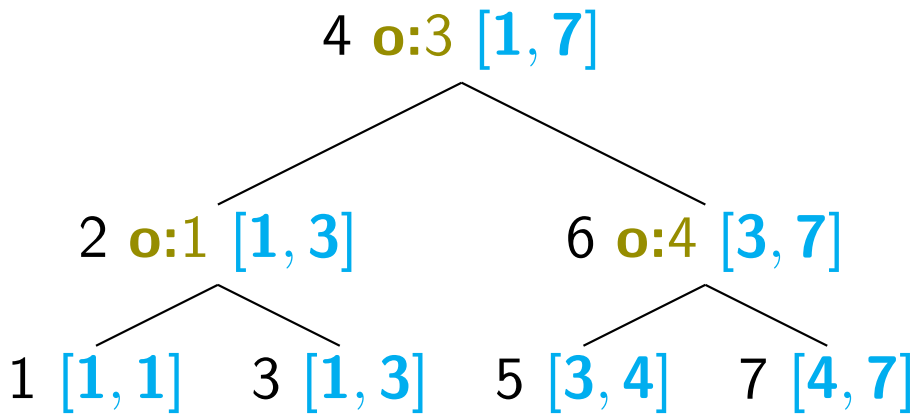
- Under this procedure, what is the cost to calculate $dp[k][2]$?



- The size of the range we have to search in for index 2. In this case it is $|[1, 3]| = 3$.

Divide and Conquer Framework

- What is the total cost to calculate all the $dp[k][i]$?



- Sum of the sizes of the ranges. The sum of the sizes for each layer is $O(N)$, there are $O(\log N)$ layers. Therefore $O(N \log N)$ overall.

- Hence by calculating our DP in this divide and conquer order, we get a speed up of $O(N^2K)$ to $O(NK \log N)$.
- Implementation wise, we DFS instead of doing it layer by layer.
- Our code is similar to range trees. During our DFS, we keep track of a range $[cL, cR)$ and our DFS will calculate $dp[k][i]$ for all $i \in [cL, cR)$. We also keep the search range (the range in the tree diagrams).

Divide and Conquer Framework


```
// Insert your cost function here:
long long Cost(int i, int j);

// Search range for our DP is in the range [qL, qR]
// In this branch, calculating dp[k][cL,..cR)
void dnc(int qL, int qR, int cL, int cR, int k) {
    if (cR <= cL) return;
    int bestpos(-1);
    int mid = (cL + cR) / 2;
    dp[k][mid] = INF; // assume this is a minimisation problem
    for (int i = qL; i <= min(qR, mid-1); i++) {
        // Depending on definition of Cost,
        //  add Cost(i+1, mid) or Cost(i, mid)
        long long newcost = dp[k-1][i] + Cost(i+1, mid);
        if (newcost < dp[k][mid]) {
            dp[k][mid] = newcost;
            bestpos = i;
        }
    }
    // Split our range in 2.
    // In the left, the search range is [qL, bestpos]
    //  and we calculate dp[k][cL,mid)
    // In the right, the search range is [bestpos, qR]
    //  and we calculate dp[k][mid+1,cR)
    dnc(qL, bestpos, cL, mid, k);
    dnc(bestpos, qR, mid+1, cR, k);
}
```

Example Problem: Noisy Kids

- Let us return to our original example problem.
- What is the cost function here? Letting $S[N]$ be the cumulative array, $Cost(j+1, i) = (S[i] - S[j])^2$.
- We can substitute this directly into our divide and conquer template.

Example Problem: Noisy Kids


```cpp
#include <bits/stdc++.h>
using namespace std;

const long long INF = 1e17;
const int MAXN = 10000;
const int MAXK = 100;
int N, K;
long long S[MAXN+1]; // cumulative array
long long dp[MAXK+1][MAXN+1];

// Cost of segment (i, j]
long long Cost(int i, int j) {
    return (S[j]-S[i])*(S[j]-S[i]);
}
// Search range: [qL, qR], calculating dp[k][cL..cR)
void dnc(int qL, int qR, int cL, int cR, int k) {
    if (cR <= cL) return;
    int bestpos;
    int mid = (cL + cR) / 2;
    dp[k][mid] = INF;
    for (int i = qL; i <= min(qR, mid-1); i++) {
        // Cost expects (], so we use Cost(i, mid).
        long long newcost = dp[k-1][i] + Cost(i, mid);
        if (newcost < dp[k][mid]) {
            dp[k][mid] = newcost;
            bestpos = i;
        }
    }
    dnc(qL, bestpos, cL, mid, k);
    dnc(bestpos, qR, mid+1, cR, k);
}
```

Example Problem: Noisy Kids


```
int main() {
    scanf("%d %d", &N, &K);
    for (int i = 1; i <= N; i++) {
        long long c; scanf("%lld", &c);
        S[i] = S[i-1] + c;
    }
    // For K = 0, dp[0][0] = 0 is the base case.
    for (int i = 1; i <= N; i++) dp[0][i] = INF;
    // Just call dnc for k from 1 to N
    // Make sure you get the initial [qL,qR] and [cL, cR) correct.
    for (int k = 1; k <= K; k++) dnc(0, N, 1, N+1, k);
    printf("%lld\n", dp[K][N]);
    return 0;
}
```

- **Complexity?** $O(NK \log N)$. Here $N = 10^4, K = 100$ we get $\approx 14 \cdot 10^6$ which is fine for a second.
- But we left out a detail, how do we know that $opt[k][i]$ is non-decreasing for a fixed k?
- This is a special case of a more general phenomenon.

Proof of Monotonicty of opt

- This part is quite technical. At the end I've included a key takeaways slide.
- The more important part to get is the intuition. Feel free to skip verifying any of the maths equations.

- For concreteness sakes, let $Cost(i, j) := (S[j] - S[i])^2$.
- I claim $Cost(i, j)$ satisfies the key feature, if $a < b < c < d$:

$$Cost(a, d) - Cost(b, d) \geq Cost(a, c) - Cost(b, c)$$

- What does this say? Suppose we started with looking at the ranges $[b, c]$ and $[b, d]$ and we define $A := Cost(b, c)$ and $B := Cost(b, d)$.
- Then it says that adding an extra item to the interval $[b, d]$ increases the cost more than adding the same item to the interval $[b, c]$.
- But when our function is squaring, this is immediate. This just says that $(B + c)^2 - B^2 \geq (A + c)^2 - A^2$ if $B > A \geq 0$.
- This follows from x^2 being convex (its derivative is an increasing function).

- Now, let *Cost* can be any function that satisfies:

$$Cost(a, d) - Cost(b, d) \geq Cost(a, c) - Cost(b, c)$$

  for all $a < b < c < d$.
- To prove $opt[k][i]$ is increasing in i, we proceed with the standard swapping argument.
- Suppose for a contradiction that $opt[k][i + 1] < opt[k][i]$.

- Then on the one hand by definition of $o_{k,i+1}$:

$$dp[k-1][o_{k,i+1}] + Cost(o_{k,i+1}, i+1) < dp[k-1][o_{k,i}] + Cost(o_{k,i}, i+1)$$

or equivalently:

$$Cost(o_{k,i+1}, i+1) - Cost(o_{k,i}, i+1) < dp[k-1][o_{k,i}] - dp[k-1][o_{k,i+1}]$$

- But the opposite situation holds for $o_{k,i}$:

$$dp[k-1][o_{k,i}] + Cost(o_{k,i}, i) < dp[k-1][o_{k,i+1}] + Cost(o_{k,i+1}, i)$$

or equivalently:

$$dp[k-1][o_{k,i}] - dp[k-1][o_{k,i+1}] < Cost(o_{k,i+1}, i) - Cost(o_{k,i}, i)$$

- Composing these two inequalities, we get

$$Cost(o_{k,i+1}, i+1) - Cost(o_{k,i}, i+1) < Cost(o_{k,i+1}, i) - Cost(o_{k,i}, i)$$

exactly contradicting our inequality on $Cost$ as $o_{k,i+1} < o_{k,i} < i < i+1$. □

- This inequality

$$Cost(a, d) - Cost(b, d) \geq Cost(a, c) - Cost(b, c)$$

  for all $a < b < c < d$ is called the quadrangle inequality.
- You should think of it as saying the larger your segment already is, the more costly it is to put items into it.
- So intuitively, one might expect this to imply monotonicity of *opt* since moving the right endpoint from i to $i + 1$ should disincentivize the last segment growing any larger to the left.
- Which is what the above proof showed. Hence satisfying the quadrangle inequality is sufficient for *opt* to be monotonic, and hence to apply D&C Optimisation.

Some examples:

- Our example involved $Cost(i,j) = F(s_{i+1} + \ldots + s_j)$ where $F(x) := x^2$. If all our $\{s_i\}$ are positive then $Cost$ satisfies the quadrangle inequality whenever F is convex. So $F(x) = x^3, x^{1.5}, x \log x$ all work.
- Also a lot of CHT! ($Cost(i,j) = b_i + m_i \cdot p_j$ where m_i is non-increasing and p_j non-decreasing). Conversely, a decent number of D&C problems can be solved with CHT after clever rewriting.
- By induction, it suffices that we can prove it for one step:

  $$Cost(b-1,c+1) - Cost(b,c+1) \geq Cost(b-1,c) - Cost(b,c)$$

  Often $Cost(i,j)$ is defined as some function operating on the multiset of $\{s_i, s_{i+1}, \ldots, s_j\}$. For these, proving a single step is often easy.
  - E.g: $Cost(i,j) = |\{(a,b) \mid s_a + s_b = 0 \wedge a,b \in [i,j]\}|$.

Key Takeaways

Key Takeaways:

- To show opt is monotone, we almost always instead show *Cost* satisfies the quadrangle inequality.
- You don't need to know why *Cost* satisfying the quadrangle inequality implies monotonicity of opt. But it is worth knowing roughly how to check the inequality holds.
- Intuitively, it just says the larger a range already is, the more *Cost* increases more upon adding another item.
- Many *Cost* functions satisfy the quadrangle inequality, in particular convex functions applied to the sum of the range.
- If *Cost* satisfies the quadrangle inequality or we suspect it does, then we can apply the earlier Divide and Conquer framework directly just by modifying the *Cost* function.

- For maximisation problems, we want the opposite quadrangle inequality

$$Cost(a, d) - Cost(b, d) \leq Cost(a, c) - Cost(b, c)$$

that is, it is harder to increase the cost of larger segments. Some examples:
  - Concave functions (e.g: $F(x) := x^{0.5}, F(x) = \log x$).
  - Again, it suffices to prove it for a single step.
- Often the annoying part is calculating $Cost(i, j)$ quickly. E.g: when $Cost(i, j) = |\{(i, j) \mid s_i + s_j = 0\}|$. For D&C it is too slow to calculate it in $O(j - i)$.

- The quadrangle inequality further gives that

$$opt[k-1][i] \leq opt[k][i] \leq opt[k][i+1]$$

Using this and doing the DP in increasing k order, then decreasing i order, gives a running time of $O((N+K)N)$. This is sometimes called "Knuth's Optimisation". (though I feel this term is overloaded)

Example: Arranging Heaps

- **Problem Statement:** There are N mines on a river, the ith is at position X_i and produces W_i units of coal.
  To collect coal, K facilities will be built on the river. The river only flows downstream so coal from position i can be delivered to position j iff $j > i$. To deliver w units of coal from position X to Y costs $w \cdot (Y - X)$ units of fuel.
  If we optimally position the facilities, what is the minimum amount of fuel necessary to collect all the coal?
- **Source:** 2012 Latin America Regionals.

Example: Arranging Heaps

- **Input Format:** First line, 2 integers N, K. $1 \leq K < N \leq 2000$. Next N lines describe the mines. Each line contains 2 integers X_i, W_i with $1 \leq X_i, W_i \leq 10^6$. The mines are in strictly increasing order of X_i.
- **Sample Input:**

```
3 2
11 3
12 2
14 1
```

- **Sample Output:** 3

Example: Arranging Heaps

- As always, start simple. What order to process in and what state?
- The standard ones for 2D DP. State is $dp[k][i]$, minimum cost to collect coal exactly from the first i mines using k facilities.
- Order is increasing k then increasing i. Increasing i then k works too.
- Recurrence? Has to be something about building facilities.
- **Easy Observation:** We should only build facilities at mines. By definition, in the state $dp[k][i]$, we must build a facility at mine i. So what is the choice?
- Choice is where the previous facility was built.

Example: Arranging Heaps

- Recurrence:

$$dp[k][i] = \min_{j<i} dp[k-1][j] + Cost(j, i)$$

  What is $Cost(j, i)$ here? (in words)
- It is the total cost to deliver coal from mines $j+1, \ldots, i$ to a facility at mine i.
- Formula?
-

$$\sum_{t=j+1}^{i} W_t \cdot (X_i - X_t)$$

- Exercise: Make this run in $O(N^2K)$ (Hint: Either rewrite the equation or do a backwards sweep to calculate the above min).
- $O(N^2K) \approx 8bil$. Too slow (at least without extreme micro optimization).

Example: Arranging Heaps

- How to speed this up? Note this is a standard 2D DP where we are making a choice per state, but making this choice is introducing an $O(N)$ cost. And we are blowing out due to this $O(N)$ cost not the state space.
- So we should look at whether this choice has a nice monotone structure. I.e: is $opt[k][i] \leq opt[k][i+1]$.
- To check this, we should check the quadrangle inequality, or the one step version:

$$Cost(b-1, c+1) - Cost(b, c+1) \geq Cost(b-1, c) - Cost(b, c)$$

- **Important:** What does your intuition tell you?
- My intuition tells me this seems likely. Adding another mine to a larger segment should incur more cost relative to adding it to a smaller segment. In the former, the coal will have to be shipped further.

- We can also give a quick proof this holds:

  $$Cost(b-1, c+1) - Cost(b, c+1) \geq Cost(b-1, c) - Cost(b, c)$$

- Recall

  $$Cost(j, i) := \sum_{t=j+1}^{i} W_t \cdot (X_i - X_t)$$

- Since most terms cancel, LHS is just $W_b \cdot (X[c+1] - X[b])$ while RHS is just $W_b \cdot (X[c] - X[b])$.
- Therefore quadrangle inequality is satisfied and we can apply D&C optimization.

- The final ingredient is we need to calculate $Cost(j, i)$ in $O(1)$ where

$$Cost(j, i) := \sum_{t=j+1}^{i} W_t \cdot (X_i - X_t)$$

- The difficulty is that X_i varies so we can't precompute this.
- Standard trick: rewrite this to isolate out the X_i part:

$$Cost(j, i) := \left(\sum_{t=j+1}^{i} W_t \right) X_i - \left(\sum_{t=j+1}^{i} W_t X_t \right)$$

- How do we calculate this in $O(1)$ now?
- Cumulative sums over the array $W[N]$ and the array $\{W_t X_t\}_{t=1}^{N}$.

Example: Arranging Heaps


```
#include <bits/stdc++.h>
using namespace std;
const long long INF = 1e18;
const int MAXN = 1000, MAXK = 1000;
int N, K;
long long X[MAXN+1], W[MAXN+1], sXW[MAXN+1], sW[MAXN+1];
long long dp[MAXK+1][MAXN+1];

// Cost of moving mines (i,j] to mine j.
long long Cost(int i, int j) {
    return (sW[j]-sW[i])*X[j] - (sXW[j]-sXW[i]);
}
// Best choice for our DP is in the range [qL, qR]
// Calculating dp[k][cL,..cR)
void dnc(int qL, int qR, int cL, int cR, int k) {
    if (cR <= cL) return;
    int bestpos(-1);
    int mid = (cL + cR) / 2;
    dp[k][mid] = INF;
    for (int i = qL; i <= min(qR, mid-1); i++) {
        // We use Cost(i, mid) since Cost(i,j) is the cost of (i,j]
        long long newcost = dp[k-1][i] + Cost(i, mid);
        if (newcost < dp[k][mid]) {
            dp[k][mid] = newcost;
            bestpos = i;
        }
    }
    dnc(qL, bestpos, cL, mid, k);
    dnc(bestpos, qR, mid+1, cR, k);
}
```

Example: Arranging Heaps


```
int main() {
    scanf("%d %d", &N, &K);
    for (int i = 1; i <= N; i++) {
        scanf("%lld %lld", &X[i], &W[i]);
        sXW[i] = sXW[i-1] + X[i] * W[i];
        sW[i] = sW[i-1] + W[i];
    }
    for (int i = 1; i <= N; i++) dp[0][i] = INF;
    for (int k = 1; k <= K; k++) dnc(0, N, 1, N+1, k);
    printf("%lld\n", dp[K][N]);
    return 0;
}
```

- **Complexity?** $O(NK \log N) \approx 44mil$. Likely fine for a second on a relatively modern machine.
- Common in many problems of this kind.
  - Start with the obvious DP but the recurrence is too slow.
  - Look at the structure there. Since you are making choices, check if these choices might be monotone. You want your intuition to play a strong role here.
  - To apply D&C, want *Cost* to be around $O(1)$ amortized. For simple examples, this usually just involves rewriting the equation until you can apply basic data structures. For more complicated examples, you may need to be more careful with how you persist state in your D&C.
- Aside: with some rewriting, not hard to do this with CHT either.