Convex Hull Optimization

Arnar Bjarni Arnarson

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Convex Hull Optimization

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.
- Each time a tree is cut its height is reduced by 1.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.
- Each time a tree is cut its height is reduced by 1.
- We use the chainsaw charging coefficient of the tree with the highest id which has been cut completely.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.
- Each time a tree is cut its height is reduced by 1.
- We use the chainsaw charging coefficient of the tree with the highest id which has been cut completely.
- If no tree has been cut completely, then it is impossible to charge the chainsaw.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.
- Each time a tree is cut its height is reduced by 1.
- We use the chainsaw charging coefficient of the tree with the highest id which has been cut completely.
- If no tree has been cut completely, then it is impossible to charge the chainsaw.

Kalila and Dimna in the Logging Industry

- Link to problem statement which you should read first.
- We have n trees.
- Each tree i has height a_i, given in strictly ascending order.
- Each tree i also has a chainsaw charging coefficient b_i, given in strictly descending order.
- Each time a tree is cut its height is reduced by 1.
- We use the chainsaw charging coefficient of the tree with the highest id which has been cut completely.
- If no tree has been cut completely, then it is impossible to charge the chainsaw.
- We want to minimize the total charge cost to cut all trees.

Analyzing the problem

- We are given $a_1 = 1$, so initially we must cut the first tree.

Analyzing the problem

- We are given $a_1 = 1$, so initially we must cut the first tree.
- Since $b_n = 0$ we must only cut the largest tree, at that point all cuts are free.

Analyzing the problem

- We are given $a_1 = 1$, so initially we must cut the first tree.
- Since $b_n = 0$ we must only cut the largest tree, at that point all cuts are free.
- We want to minimize the cost required to cut the largest tree.

Analyzing the problem

- We are given $a_1 = 1$, so initially we must cut the first tree.
- Since $b_n = 0$ we must only cut the largest tree, at that point all cuts are free.
- We want to minimize the cost required to cut the largest tree.
- It is also quite clear that once we start cutting a tree, we should finish cutting it before starting to cut others.

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.
- $$c_i = \min_{0 \leq j < i} (c_j + b_j \cdot a_i)$$

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.
- $$c_i = \min_{0 \leq j < i} (c_j + b_j \cdot a_i)$$
- This directly translates to an $\mathcal{O}\left(N^2\right)$ dynamic programming solution.

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.
- $$c_i = \min_{0 \leq j < i} (c_j + b_j \cdot a_i)$$
- This directly translates to an $\mathcal{O}\left(N^2\right)$ dynamic programming solution.
- But N can be 10^5 which means this is not fast enough.

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.
- $$c_i = \min_{0 \leq j < i} (c_j + b_j \cdot a_i)$$
- This directly translates to an $\mathcal{O}\left(N^2\right)$ dynamic programming solution.
- But N can be 10^5 which means this is not fast enough.
- Now substitute c_j for y_0, b_j for m and a_i for x. What do you get?

Constructing a solution

- Let c_i be the minimum cost of cutting tree i.
- We can compute each c_i by considering all possible trees $j < i$ that were the last tree to be cut.
- $$c_i = \min_{0 \le j < i} (c_j + b_j \cdot a_i)$$
- This directly translates to an $\mathcal{O}\left(N^2\right)$ dynamic programming solution.
- But N can be 10^5 which means this is not fast enough.
- Now substitute c_j for y_0, b_j for m and a_i for x. What do you get?
- A line!

Enter the Convex Hull Trick

- We want to maintain a set of lines.

Enter the Convex Hull Trick

- We want to maintain a set of lines.
- We need to be able to add lines to the set, at the very least in descending order by slope.

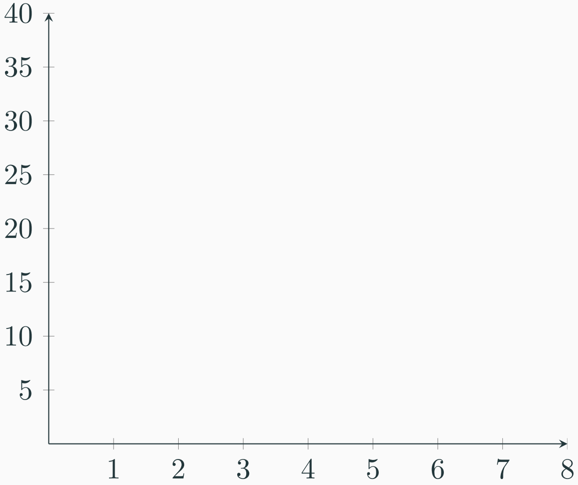
Enter the Convex Hull Trick

- We want to maintain a set of lines.
- We need to be able to add lines to the set, at the very least in descending order by slope.
- We need to be able to provide an x value and find the line with the minimum y value for that x.

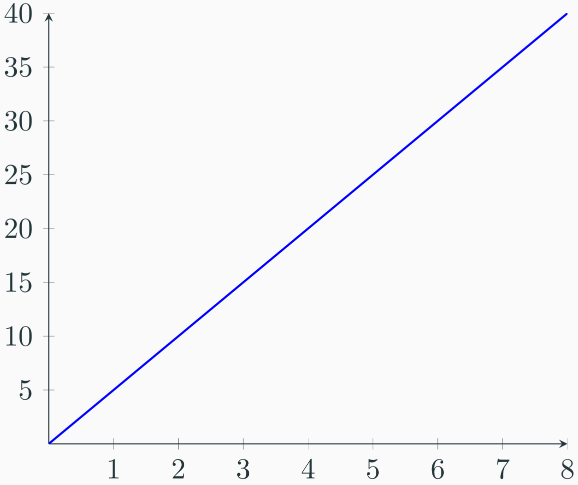
Enter the Convex Hull Trick

- We want to maintain a set of lines.
- We need to be able to add lines to the set, at the very least in descending order by slope.
- We need to be able to provide an x value and find the line with the minimum y value for that x.
- We need both operations to be sub-linear in time complexity.

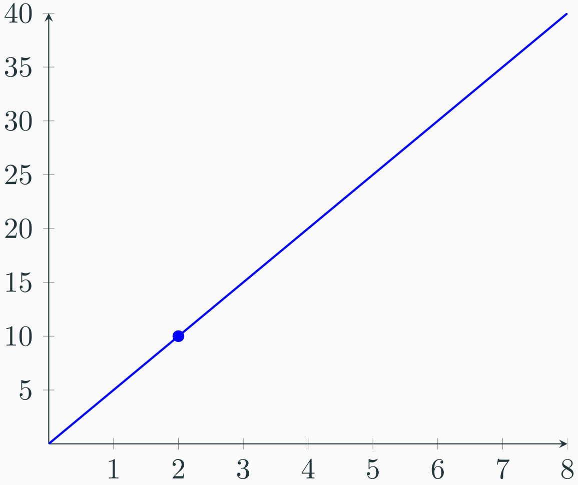
Sample 1 - Illustrated



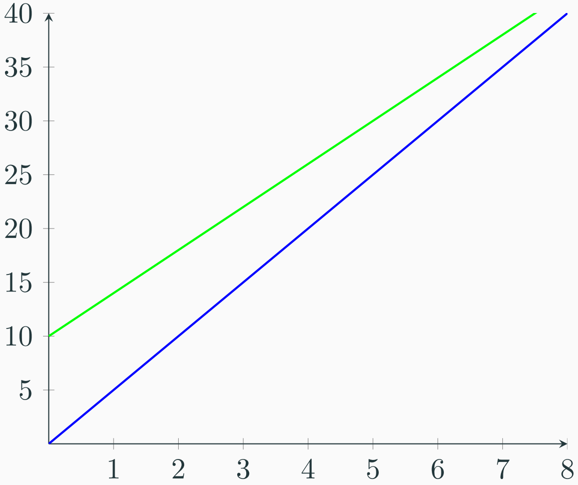
Sample 1 - Illustrated



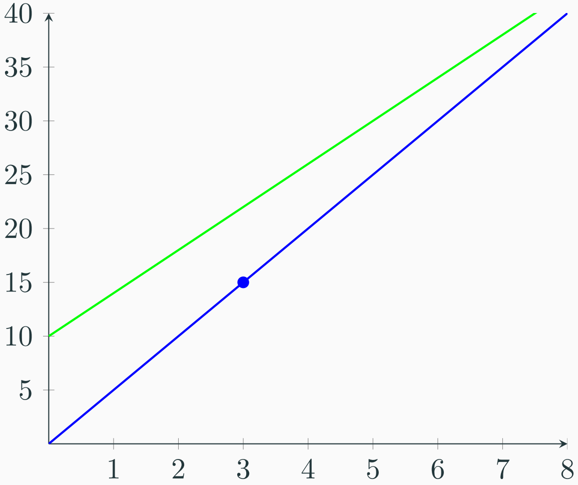
Sample 1 - Illustrated



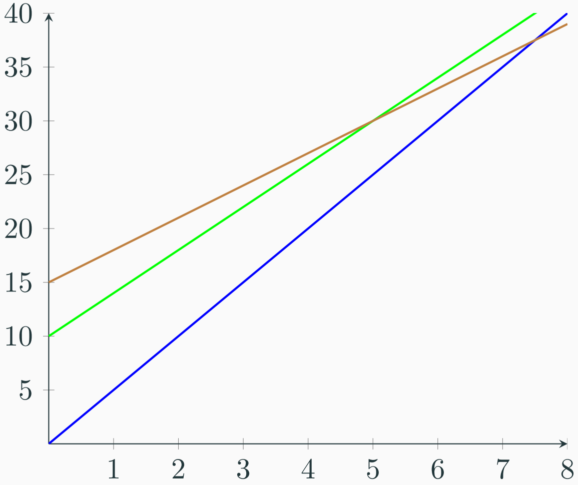
Sample 1 - Illustrated



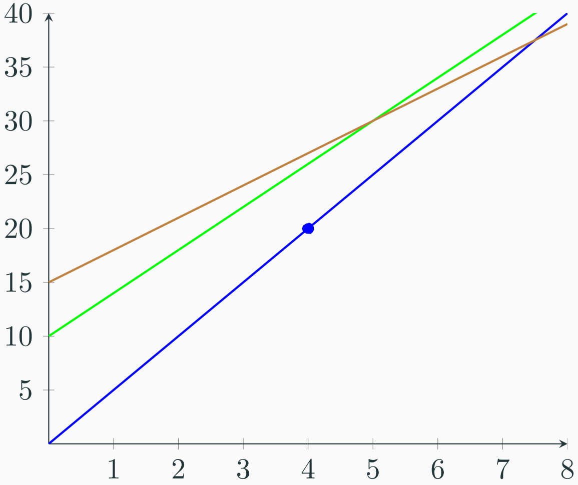
Sample 1 - Illustrated



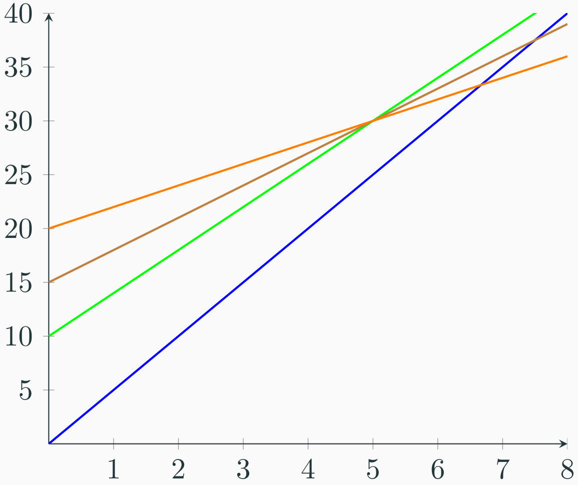
Sample 1 - Illustrated



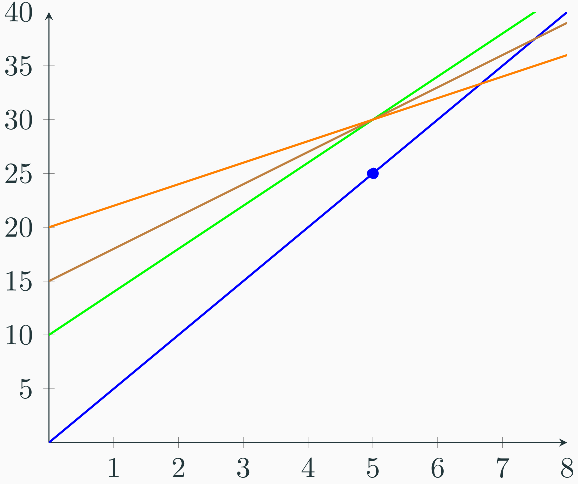
Sample 1 - Illustrated



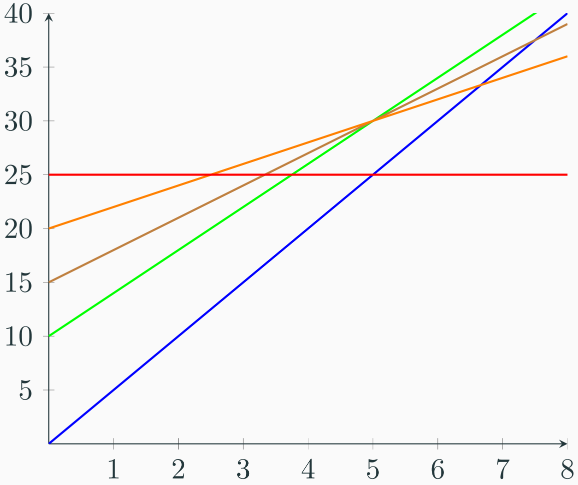
Sample 1 - Illustrated



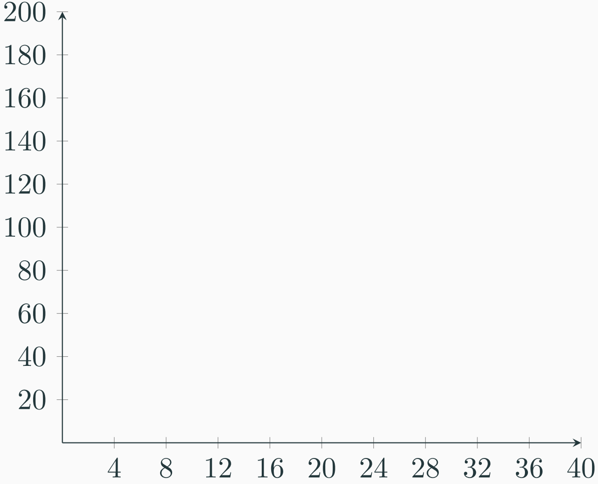
Sample 1 - Illustrated



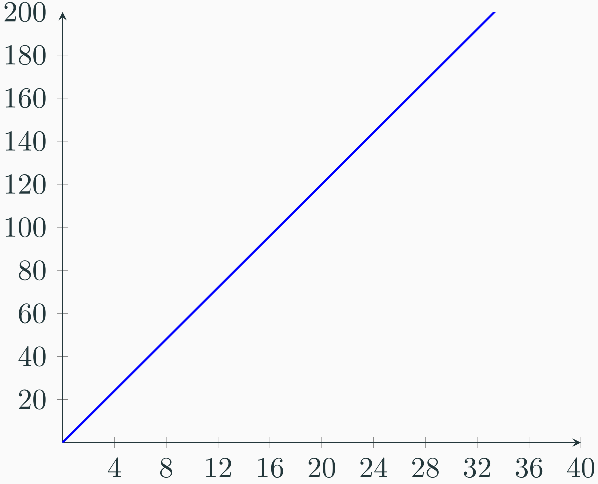
Sample 1 - Illustrated



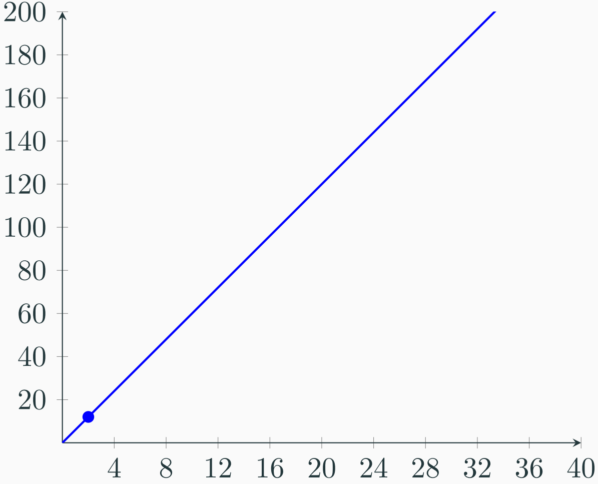
Sample 2 - Illustrated



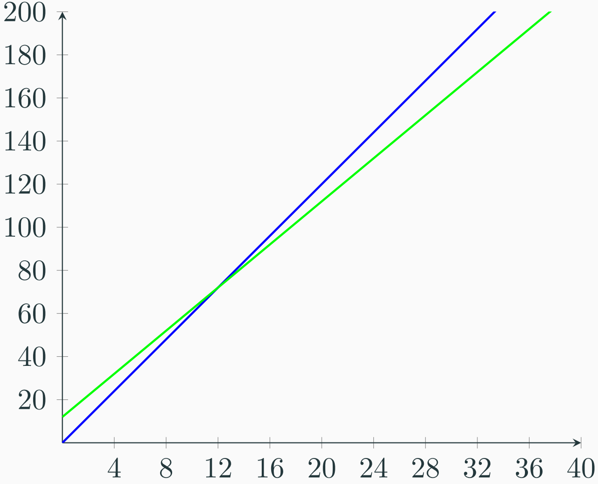
Sample 2 - Illustrated



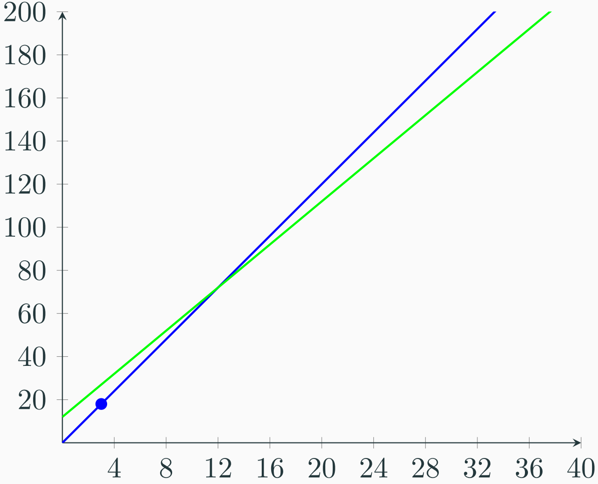
Sample 2 - Illustrated



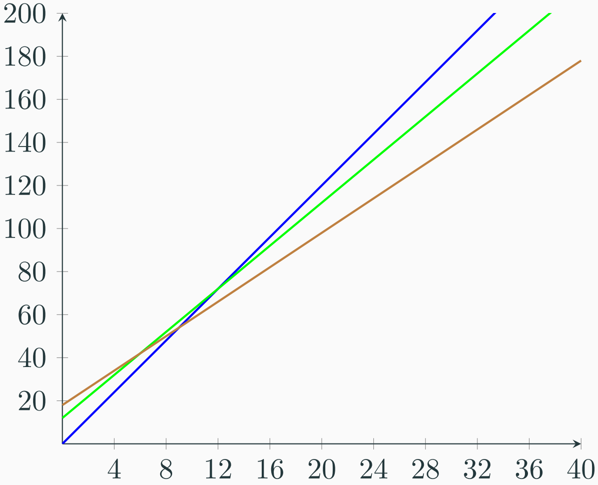
Sample 2 - Illustrated



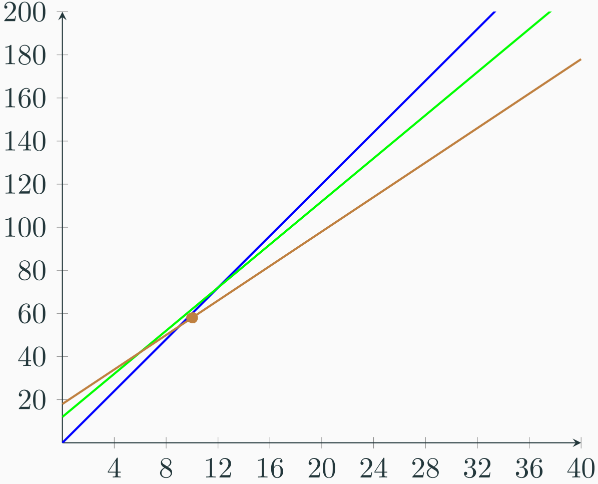
Sample 2 - Illustrated



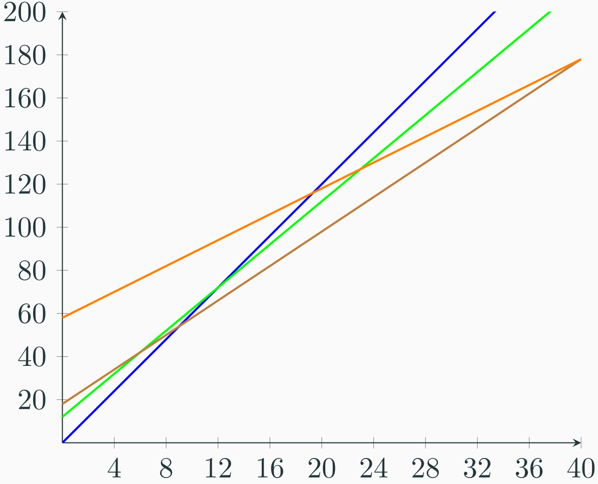
Sample 2 - Illustrated



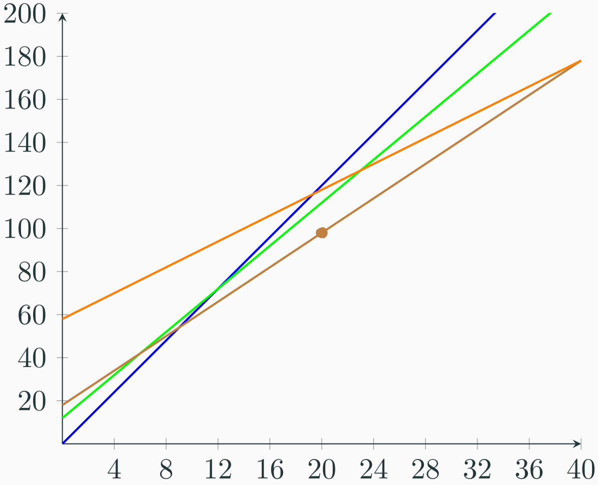
Sample 2 - Illustrated



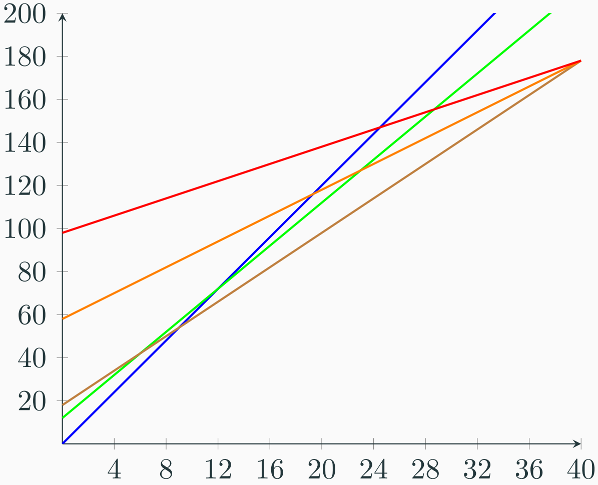
Sample 2 - Illustrated



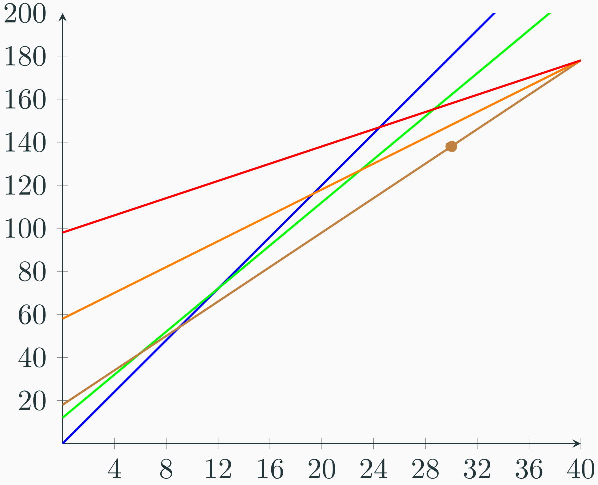
Sample 2 - Illustrated



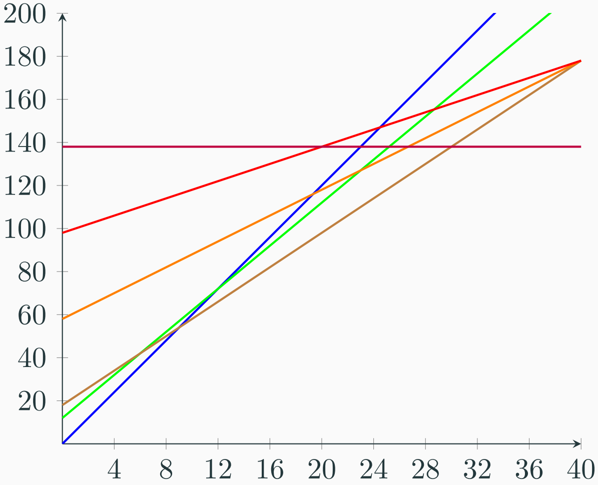
Sample 2 - Illustrated



Sample 2 - Illustrated



Sample 2 - Illustrated



Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.

Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.
- Suppose we are adding a line to our data structure.

Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.
- Suppose we are adding a line to our data structure.
- Let a be the intersection point of the new line and the second to last line in the data structure.

Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.
- Suppose we are adding a line to our data structure.
- Let a be the intersection point of the new line and the second to last line in the data structure.
- Let b be the intersection point of the last line and the second to last line in the data structure.

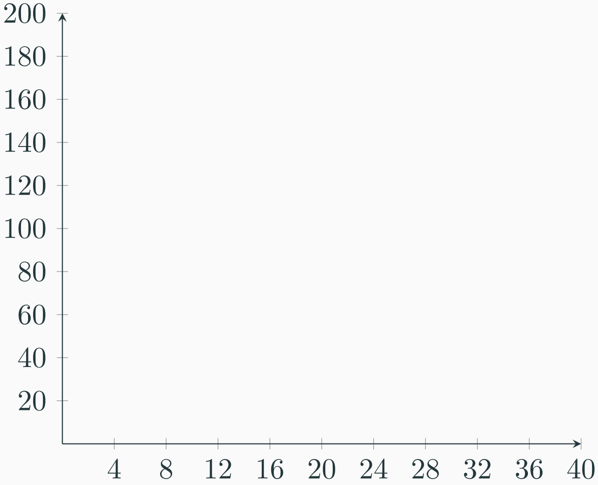
Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.
- Suppose we are adding a line to our data structure.
- Let a be the intersection point of the new line and the second to last line in the data structure.
- Let b be the intersection point of the last line and the second to last line in the data structure.
- If the x-coordinate of a is less than (or equal to) that of b, then the last line is redundant.

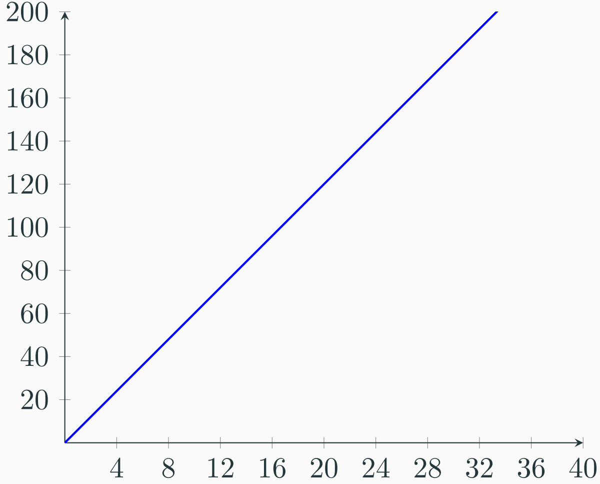
Remove useless lines

- We can see some lines can be discarded, since they do not contribute to the convex hull.
- Suppose we are adding a line to our data structure.
- Let a be the intersection point of the new line and the second to last line in the data structure.
- Let b be the intersection point of the last line and the second to last line in the data structure.
- If the x-coordinate of a is less than (or equal to) that of b, then the last line is redundant.
- We can therefore iteratively pop redundant lines from the back before adding a line.

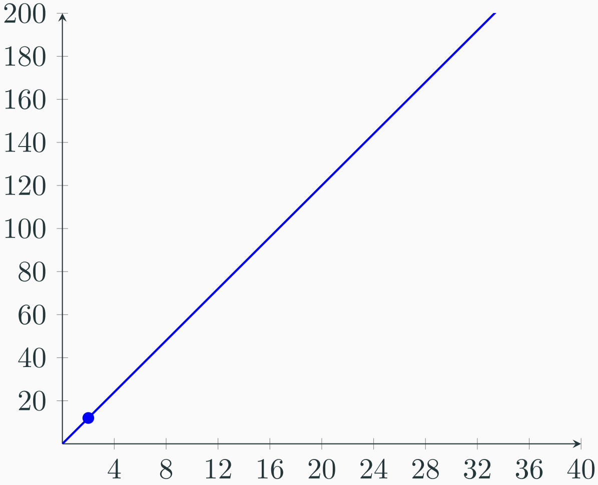
Sample 2 - Remove useless lines



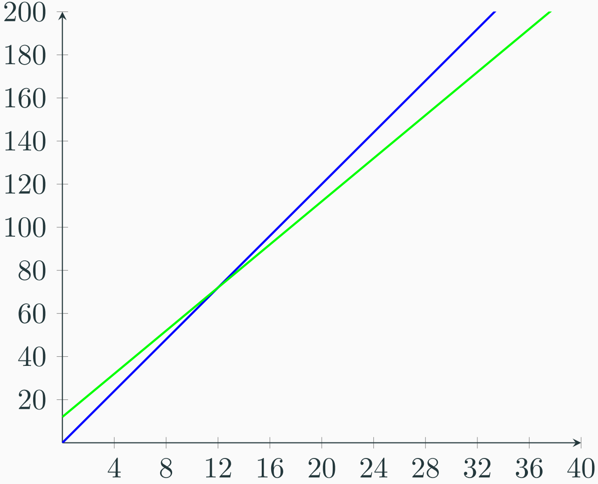
Sample 2 - Remove useless lines



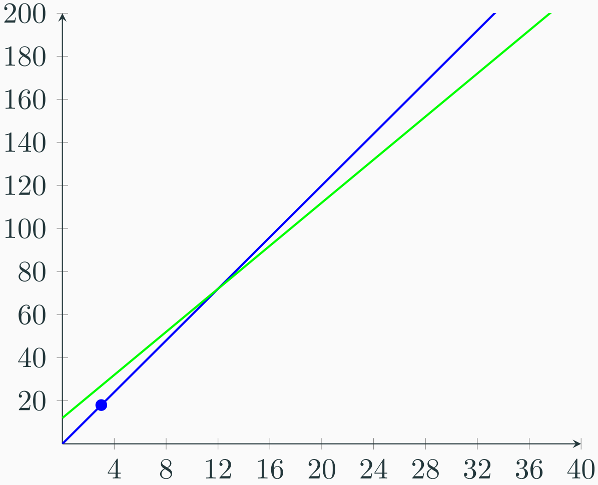
Sample 2 - Remove useless lines



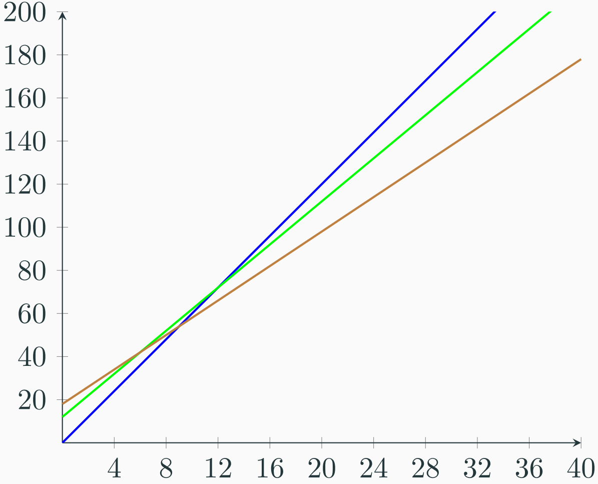
Sample 2 - Remove useless lines



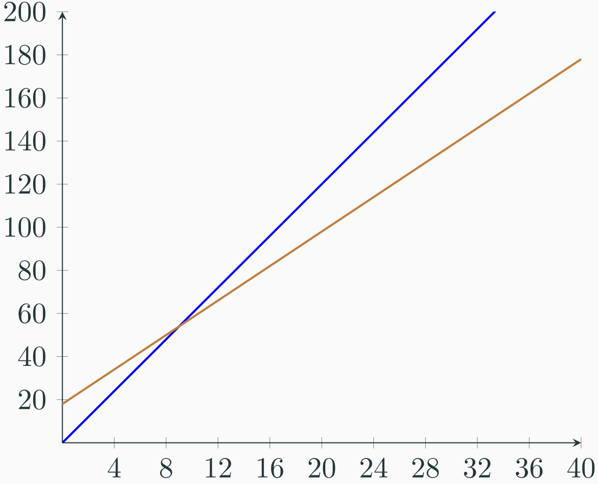
Sample 2 - Remove useless lines



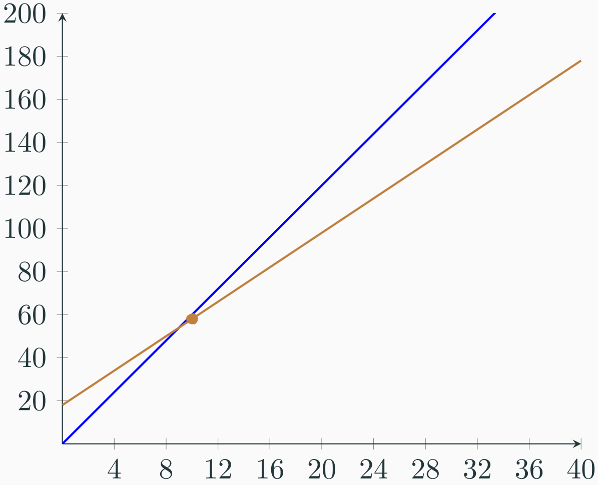
Sample 2 - Remove useless lines



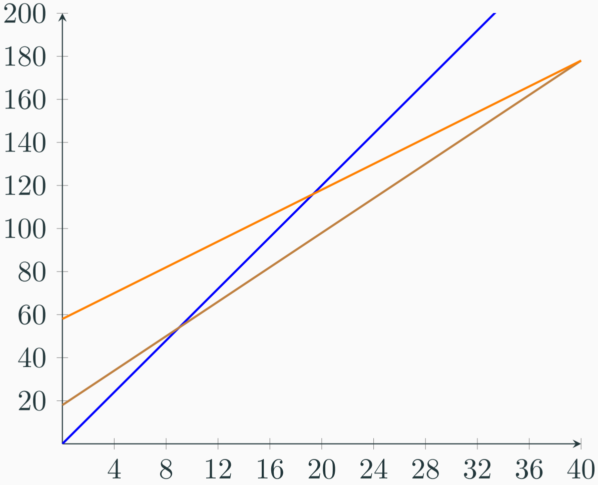
Sample 2 - Remove useless lines



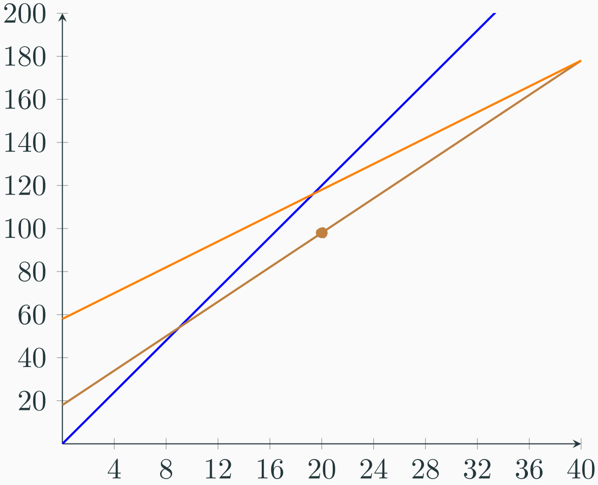
Sample 2 - Remove useless lines



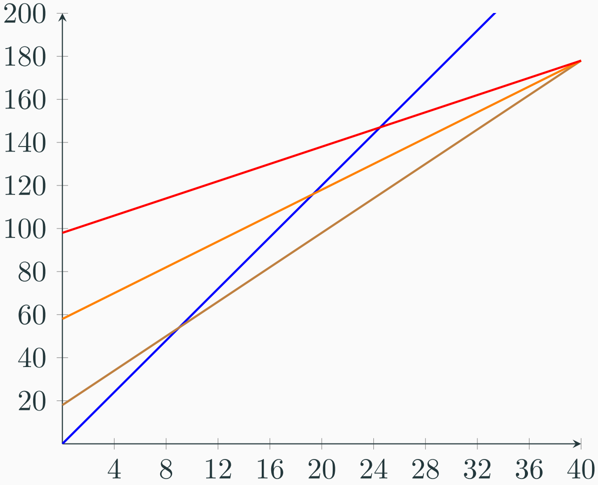
Sample 2 - Remove useless lines



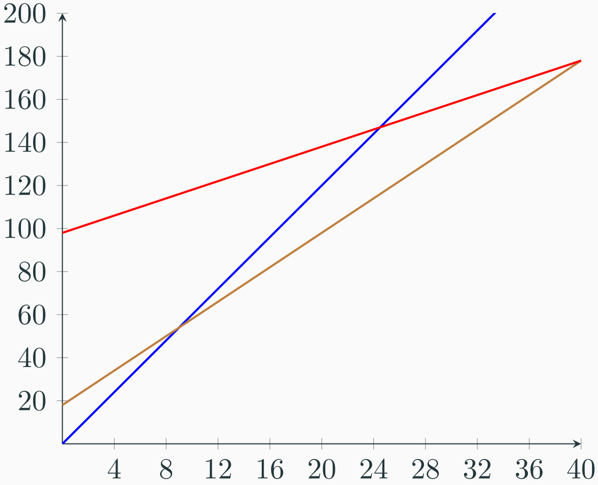
Sample 2 - Remove useless lines



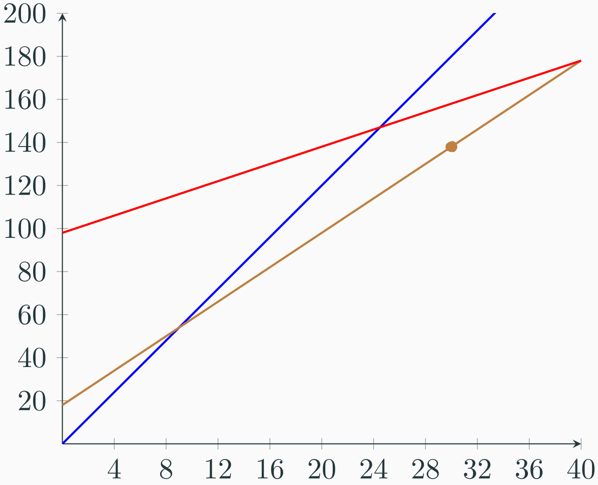
Sample 2 - Remove useless lines



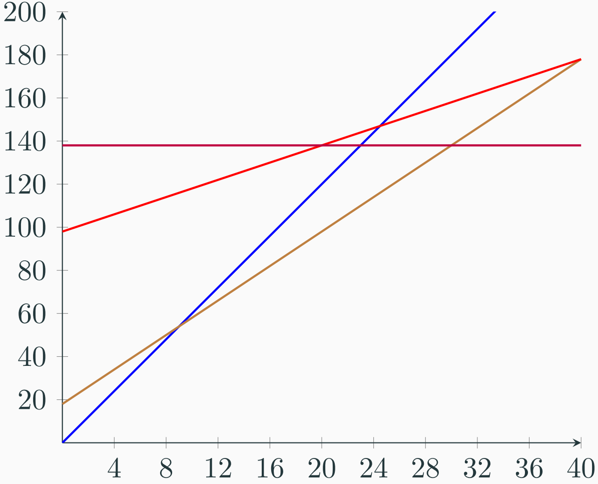
Sample 2 - Remove useless lines



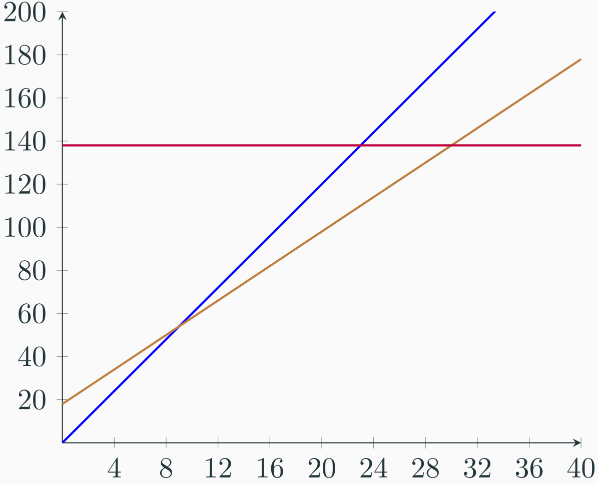
Sample 2 - Remove useless lines



Sample 2 - Remove useless lines



Sample 2 - Remove useless lines



Querying and Complexity

- How do we find the minimum y value then?

Querying and Complexity

- How do we find the minimum y value then?
- The intersection points are in ascending order.

Querying and Complexity

- How do we find the minimum y value then?
- The intersection points are in ascending order.
- Use binary search to find the line in question.

Querying and Complexity

- How do we find the minimum y value then?
- The intersection points are in ascending order.
- Use binary search to find the line in question.
- Construction takes $\mathcal{O}(n)$ time

Querying and Complexity

- How do we find the minimum y value then?
- The intersection points are in ascending order.
- Use binary search to find the line in question.
- Construction takes $\mathcal{O}(n)$ time
- Each query takes $\mathcal{O}(\log n)$ time.

Querying and Complexity

- How do we find the minimum y value then?
- The intersection points are in ascending order.
- Use binary search to find the line in question.
- Construction takes $\mathcal{O}(n)$ time
- Each query takes $\mathcal{O}(\log n)$ time.
- We have improved the time complexity from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}(n \log n)$.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.
- Easy to change from minimization to maximization.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.
- Easy to change from minimization to maximization.
- If query points are also known to be in ascending order, you can use a sliding window idea to get $\mathcal{O}(n)$ time complexity.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.
- Easy to change from minimization to maximization.
- If query points are also known to be in ascending order, you can use a sliding window idea to get $\mathcal{O}(n)$ time complexity.
- Alternatively, modify such that lines may come in any order while maintaining time complexity.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.
- Easy to change from minimization to maximization.
- If query points are also known to be in ascending order, you can use a sliding window idea to get $\mathcal{O}(n)$ time complexity.
- Alternatively, modify such that lines may come in any order while maintaining time complexity.
- Use a multiset instead of a vector/array to keep track of lines and quickly find insertion point. A bit more hassle but powerful.

Modifications

- Our current model only works if lines are inserted increasing/decreasing order by slope, but query points can be given in any order.
- Easy to change from minimization to maximization.
- If query points are also known to be in ascending order, you can use a sliding window idea to get $\mathcal{O}(n)$ time complexity.
- Alternatively, modify such that lines may come in any order while maintaining time complexity.
- Use a multiset instead of a vector/array to keep track of lines and quickly find insertion point. A bit more hassle but powerful.
- Need to consider removing neighbouring lines with higher and lower slopes.

Simple Implementation

```
struct line {
    ll m, b;
    ll get(ll x) { return m*x + b; };
    ll intersect(line other) { return (other.b - b) / (m - other.m); }
};

struct convex_hull_trick {
    vector<line> lines;
    void add(line l) {
        auto sz = lines.size();
        while (sz >= 2 &&
               lines[sz-2].intersect(lines[sz-1]) >= lines[sz-2].intersect(l))
            lines.pop_back();
            sz--;
        }
        lines.push_back(l);
    }
    // to be continued...
```

Simple Implementation - Continued

```
    // ...continued
    ll query(ll x) {
        int lo = 0, hi = static_cast<int>(lines.size()) - 2;
        int ind = hi+1;
        while (lo <= hi) {
            int mid = (lo+hi)/2;
            if (lines[mid].intersect(lines[mid+1]) >= x) {
                ind = mid;
                hi = mid-1;
            }
            else {
                lo = mid+1;
            }
        }
        return lines[ind].get(x);
    }
};
```

Try on these problems!

- Kalila and Dimna in the Logging Industry
- Covered Walkway
- Commando
- Avoiding Airports