Problem Solving Paradigms

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Outline

- Unlike previous lectures, this lecture won't be on a specific field. Instead, we will look at some generally useful paradigms and techniques that show up in all kinds of problems.
- These are linear sweeping and binary search. These are fundamental techniques that show up *everywhere*.
- From this week onwards, these techniques might just randomly show up in other problem sets. They will almost definitely show up somewhere in the finals.

Table of Contents

Linear Sweeps

- Very basic but fundamental idea. Instead of trying to do a problem all at once, try to do it in some order that lets you build up state.
- This lets you process events one by one. This can be easier than trying to handle them all at once.
- **General Principle:** Having an order is better than not having an order!
- Trying to sort and pick the right order to do a problem in is fundamental.
- If there isn't a natural order to a problem, you may as well try to do it in any sorted order.
- Even if there is a natural order, sometimes it isn't the right one!

Example: Committees

- **Problem Statement:** There are N people and D days. Each person can work for a range of days $[s_i, e_i)$. Each has a subject c_i and skill level v_i.
  A committee for day d is a subset of people who can work on day d and whose subjects are pairwise distinct.
  For each d, output the maximum skill out of all possible committees for day d.
- **Input Format:** First line 2 integers, `N D` ($1 \leq N, D \leq 100,000$). Next N lines describe a person as a 4 tuple `si ei vi ci`.

Example: Committees

- **Sample Input:**

```
4 5
0 2 2 1
1 3 3 1
0 5 1 1
2 4 1 2
```

- **Sample Output:**

```
2
3
4
2
1
```

Example: Committees

- The main difficulty here comes from the fact there are many ways to slice this problem.
- A natural first approach is to try to do a single subject at a time. So let's pick a subject s and only look at people studying subject s.
- Then for each day you just need to know the maximum skill of all people who are active for that day.
- People are active for a range and we are interested in specific days.
- So this is a standard range update, point query range tree.
- But you need one for all subjects :(
- This works if there are few subjects but to query all subjects simultaneously is hard work!

Example: Committees

- Here is an alternative approach.
- First, let us note why the earlier approach didn't work. The earlier approach was problematic as we had to do a lot of work per day.
- Our new approach will avoid this problem by building up a data structure incrementally, so on average, each extra day is not very different from the previous day.
- Our approach is to do the problem in order of days.

- To process people, we split a person into 2 events. An add at s_i and a removal at e_i.
- We track the active people with a set like structure.
- For person i, on day s_i we add person i to the set. On day e_i we remove the person from the set.
- So we use a set to track active people, instead of $O(\text{num subjects})$ range trees.

Example: Committees

- This is the general principle. But usually we need a bit more problem specific metadata.
- What we need to track is the *maximum* skill for each subject among people in our set.
- So we will keep a multiset for each subject.
- Each person gives us 2 *events*.
  - On day s_i an event to add v_i to the multiset for c_i.
  - On day e_i an event to remove v_i from the multiset for c_i.
- Our queries are the sum of maxes over all multisets.
- But, **crucially**, events only change the max of at most one multiset.

- So, in addition, we will track the sum of maxes of all multisets.
- And with some care, it is not too difficult to update this for each event (we will see this in the code).

Example: Committees

Implementation Details:

- We represent events either with a class or a pair/tuple. Example class:

```
struct event {
  bool typ; int v, c;
};
vector<event> events[D];
```

- On input, we split each person into two events, one each for the start and end of the range.
- A bit of care is needed with when we process events. Since our ranges are $[s, e)$, we will push events to days s and e and in our sweep, for each day we will process events before querying.

Example: Committees


```
// Solution by Declan
#include <bits/stdc++.h>
using namespace std;
// (is add event?, skill, subject)
vector<tuple<bool, int, int> > events[500005];
multiset<int> subjects[1000005];

int main() {
        int N, D; cin >> N >> D;
    // Removes special cases involving empty multisets
        for (int i = 0; i < 1000005; ++i) subjects[i].insert(0);
        for (int i = 0; i < N; ++i) {
                int s, e, v, c; cin >> s >> e >> v >> c;
                events[s].emplace_back(true, v, c);
                events[e].emplace_back(false, v, c);
        }
        int T = 0;
        for (int d = 0; d < D; ++d) {
                for (auto p: events[d]) {
                        int v = get<1>(p); int c = get<2>(p);
                        T -= *prev(subjects[c].end());
                        if (get<0>(p)) {
                                subjects[c].insert(v);
                        } else {
                                subjects[c].erase(subjects[c].find(v));
                        }
                        T += *prev(subjects[c].end());
                }
                cout << T << '\n';
        }
}
```

- **Moral:** Sweeping here allows us to manage a lot more data cause very little of it changes for each new day.
- Useful for many problems involving segments on a line.
- Again, be a bit careful with your ranges. Here we used $[s, e)$ so to get the answer for day d, we process day d's events first before querying. What would happen if instead the range was $[s, e]$?

Example: Line Segment Intersections

- **Problem Statement:** Given H horizontal line segments and V vertical line segments, count the number of intersections between horizontal line segments and vertical line segments. Note: If there are k intersections at point (x, y) then it should be counted k times.
- **Input Format:** First line, 2 integers, `H V`, $1 \leq H, V \leq 100,000$. Next H lines each describe a horizontal segment as a triplet, `s e y`, meaning a segment with $x \in [s, e)$ with height y. Next V lines each describe a vertical segment as a triplet, `s e x`, meaning a segment with $y \in [s, e)$ with x coordinate x. Coordinates are up to 100,000.

Example: Line Segment Intersections

- **Sample Input:**

```
3 3
1 3 3
2 4 2
1 6 1
2 4 1
0 3 2
0 10 3
```

- **Sample Output:**

```
5
```

Example: Line Segment Intersections

- Think about how one would do this globally.
- Kind of a mess really. Your data structure would have to be 2D-ish. (exercise left to reader).
- A standard trick in that case is to sweep in 1 direction, with the hope this reduces a dimension.
- Say we sweep from left to right. So at any point of our algorithm, we will be at some x coordinate c. The idea is to keep track of the horizontal segments that cross $x = c$.
- These will be the "active" segments.

Example: Line Segment Intersections

- Now we use the same framework as the previous problem. To track active segments, we split them into 2 events.
- For the horizontal segment $[s, e)$, we add an activate event to $x = s$ and a deactivate event to $x = e$.
- What should these events actually do? Same as asking what metadata we need to store.
- Well, what do we actually need to calculate for each x coordinate?
- We need to query the number of horizontal segments that intersect a vertical segment.

Example: Line Segment Intersections

- So our queries are "How many active segments are in a range?" Our updates are "Activate/Deactivate a segment". What data structure should we use?
- Range query, point update range tree!
- Once we have that, the code is similar to the previous problem.

Example: Line Segment Intersections


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

const int MAXDIM = 100000;
// Range sum, point add range tree:
int querySum(int qL, int qR); // [qL, qR)
void pointAdd(int ind, int v);

struct event {
    bool isactivate; int y;
    event(bool _isactivate=false, int _y=0) :
        isactivate(_isactivate), y(_y) {}
};
vector<event> events[MAXDIM+5];
vector<pair<int, int>> vertSegments[MAXDIM+5];
int H, V;

void input() {
    scanf("%d %d", &H, &V);
    for (int i = 0; i < H; i++) {
        int s, e, y;
        scanf("%d %d %d", &s, &e, &y);
        events[s].emplace_back(true, y);
        events[e].emplace_back(false, y);
    }
    for (int i = 0; i < V; i++) {
        int s, e, x;
        scanf("%d %d %d", &s, &e, &x);
        vertSegments[x].emplace_back(s, e);
    }
}
```

Example: Line Segment Intersections


```
int main() {
    input();
    long long ans = 0;
    for (int i = 0; i <= MAXDIM; i++) {
        for (auto ev : events[i]) {
            if (ev.isactivate) pointAdd(ev.y, 1);
            else pointAdd(ev.y, -1);
        }
        for (auto vs : vertSegments[i]) {
            ans += querySum(vs.first, vs.second);
        }
    }
    printf("%lld\n", ans);
    return 0;
}
```

Table of Contents

- Most of the examples in class have coordinates only up to 100,000 or so. But for most examples this is just a niceness condition.
- For most algorithms, the actual values of coordinates is irrelevant, just the relative order.
- So if coordinates are up to 1 billion but there are $N \leq 100,000$ points then usually there are only $O(N)$ interesting coordinates and we are bottle necked by $O(N)$.
- E.g: range queries on a set of points. I don't care exactly what the coordinates of the points or query is, just which points are within the query's range.

Coordinate Compression

- Coordinate compression is the idea of replacing each coordinate by its rank among all coordinates. Hence we preserve the relative order of values while making the maximum coordinate $O(N)$.
- This reduces us to the case with bounded coordinates.
- A few ways to implement this in $O(N \log N)$. E.g: sort, map, order statistics tree.
- I prefer one of the latter 2, since the data structure helps you convert between the compressed and uncompressed coordinates if needed (e.g: when querying).
- Also with the former, one needs to be careful of equality.

Coordinate Compression


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

// coordinates -> (compressed coordinates).
map<int, int> coordMap;

void compress(vector<int>& values) {
    for (int v : values) {
        coordMap[v] = 0;
    }
    int cId = 0;
    for (auto it = coordMap.begin(); it != coordMap.end(); ++it) {
        it->second = cId++;
    }
    for (int &v : values) {
        v = coordMap[v];
    }
}
```

Table of Contents

Binary Search

- Surprisingly powerful technique! You should have seen binary search in the context of searching an array before. For us, the power comes from binary searching on non-obvious functions instead.
- **Key problem:** Given a monotone function, find the largest/smallest x such that $f(x)$ is less than/greater than/equal to/... y.
- Actually we will pretty much only search on integer valued monotone functions with range $\{0, 1\}$.
- So, assuming they are non-decreasing, f is all 0's up to the first 1, after which it is all 1's.

Aside: Binary Search Implementation

- Hands up if you've ever messed up a binary search implementation.
- I think binary search is notorious for having annoying off-by-1s and possible infinite loops.
- Many ways to implement so pick one you're confident you can code with no thought. I'll present the one I use which I find avoids all these annoying corner cases.

Aside: Binary Search Implementation


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


// Find the smallest X such that f(X) is true;
int binarysearch(function<bool(int)> f) {
    int lo = 0;
    int hi = 100000;
    int bestSoFar = -1;
    // Range [lo, hi];
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (f(mid)) {
            bestSoFar = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return bestSoFar;
}
```

Binary Search

- The main magic application of binary search is binary searching the answer!
- The idea is in problems we are often asked to minimize a value X such that a condition holds. If we observe that increasing X does not make it any harder for the condition to hold (i.e: that if the condition holds with X then it also holds with $X + 1$) then we can binary search for the value X!
- This turns our optimization problem into a decision problem. And sometimes the decision problem is *much* easier since you no longer have to make as many choices.
- Overhead is just a factor of $O(\log A)$ where A is the range of possible answers.

Example: Gravy Surprise

- **Problem Statement:** You have a bar of chocolate with N squares, each square has a tastiness t_i. You have K friends. Break the bar into K contiguous pieces. The overall happiness of the group is the minimum total tastiness of any of these K pieces. What's the maximum overall happiness you can achieve?
- **Input Format:** First line, 2 integers, N, K with $1 \leq K \leq N \leq 1,000,000$. The next line will contain N integers, t_i, the tastiness of the ith piece. For all i, $1 \leq i \leq 100,000$.

Example: Gravy Surprise

- **Sample Input:**

  ```
  5 2
  9 7 3 7 4
  ```

- **Sample Output:**

  ```
  14
  ```

- **Explanation:** Break the bar into the first 2 squares and the last 3 squares.

Example: Gravy Surprise

- It is worth trying to approach the minimization problem directly, just to appreciate the difficulty.
- The problem is there's no greedy choices you can make. It's impossible to determine where the first cut should end. You can try a DP but the state space is large.
- We are asked to maximize the minimum sum of the K pieces.
- Let's turn this into asking about a decision problem.
- Define $b(X)$ to be True iff we can split the bar into K pieces, each with sum at least X.
- Then the problem is asking for the largest X such that $b(X)$ is True.
- **Note:** We define it to be *at least* X. This makes it monotone. If we instead defined it as *exactly* X then the function is too messy to be useful.

Example: Gravy Surprise

- **Rephrased Problem:** Define $b(X)$ to be True iff we can split the bar into K pieces, each with sum at least X. What is the largest X such that $b(X)$ is True?
- **Key(and trivial) Observation:** $b(X)$ is non-increasing.
- So we can binary search over $b(X)$. Hence to find the maximum such X, it suffices to be able to calculate $b(X)$ quickly.
- **New Problem:** Can I split the bar into K pieces, each with sum at least A?

Example: Gravy Surprise

- **New Problem:** Can I split the bar into K pieces, each with sum at least A?
- Note that we can rephrase this into a maximization question. Given each piece has sum at least A, what is the maximum number of pieces I can split the bar into?
- Let's try going one piece at a time. What should the first piece look like?
- **Key Observation:** It should be the minimum length possible while having total $\geq A$.
- This applies for all the pieces.
- So to get the maximum number of pieces needed, we sweep left to right making each piece as short as possible.

Example: Gravy Surprise


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

const int MAXN = 1000000;
int N, K; long long bar[MAXN];

bool canDo(long long A) {
    long long cPiece = 0;
    int nPieces = 0;
    for (int i = 0; i < N; i++) {
        cPiece += bar[i];
        if (cPiece >= A) {
            nPieces++;
            cPiece = 0;
        }
    }
    return nPieces >= K;
}
```

Example: Gravy Surprise


```
int main() {
    scanf("%d %d", &N, &K);
    for (int i = 0; i < N; i++) scanf("%lld", &bar[i]);
    long long lo = 1;
    long long hi = 1e12;
    long long ans = -1;
    while (lo <= hi) {
        long long mid = (lo + hi) / 2;
        // Trying to find the highest value that is feasible:
        if (canDo(mid)) {
            ans = mid;
            lo = mid + 1;
        } else {
            hi = mid - 1;
        }
    }
    printf("%lld\n", ans);
}
```

Binary Search

- **Complexity?** $O(N \log A)$ where A is max answer.
- This problem and solution is very typical of binary search problems.
- To start with, you are asked to maximize a value.
- But we can rephrase it into maximizing a value that satisfies a decision problem! In forming the decision problem, you ask if the answer could be *at least* A, not just exactly A.
- Now with the minimum tastiness of each bar fixed, you now switch to trying to maximize the number of pieces you can make. And this can be greedied since we know how small we can make each piece.
- Notice why fixing A made the problem easier. Because we had one less parameter influencing our choices and we could make greedy decisions now.

Binary Search

- One of the most common places binary search appears is in problems that ask us to maximize the minimum of something (or minimize the maximum of something).
- Another way to see if it's useful is just to see if the quantity you are minimizing is monotone.
- And this is very common! Usually, you are told to minimize a value because the problem only gets easier if it increases.
- Until you get the hang of it, it's worth just always trying to apply it.
- At worst, the decision problem can't be any harder than the optimization problem. (though it may lead you down a dead end).

Aside: Ternary Search

- Ternary search also exists. It applies to finding the maximum of a function that *strictly* increases to a peak, stays the same, then *strictly* decreases. Note the strictlys.
- Instead of splitting the range in 2, we instead now split it into 3 by querying 2 points. At each step we discard one of the thirds based on comparison of the 2 points.
- Alternatively, we can usually binary search the derivative. Usually this is the discrete form of the derivative (binary search on $h(x) := f(x+1) - f(x)$).
- Appears much less often so won't talk about it more but it is a useful thing to know exists.
- Exercise left to the reader to figure it out!

Example: Robot Moves

- **Problem Statement:** You have just created a robot that will revolutionize RoboCup forever. Well 1D RoboCup at least.
  The robot starts at position 0 on a line and can perform three types of moves:
  - **L**: Move left by 1 position.
  - **R**: Move right by 1 position.
  - **S**: Stand still.

  Currently the robot already has a loaded sequence of instructions.
  You need to get the robot to position X. To do so, you can replace a single **contiguous** subarray of the robot's instructions. What is the shortest subarray you can replace to get the robot to position X?

Example: Robot Moves

- **Input Format:** First line, 2 integers, N, X, the length of the loaded sequence and the destination. $1 \leq |X| \leq N \leq 200,000$. The next line describes the loaded sequence.
- **Sample Input:**

  `5 -4`

  `LRRLR`
- **Sample Output:**

  `4`
- **Explanation:** You can replace the last 4 instructions to get the sequence `LLLLS`.

Example: Robot Moves

- How would one do the problem directly?
- There is an $O(N^2)$ by trying all subsegments but we can't do better if we need to try all subsegments.
- Okay, well we can try binary searching now. How?
- **Key Observation:** If we can redirect the robot correctly by replacing M instructions, then we can also do so by replacing $M + 1$ instructions. Why?
- Let's turn this into a decision problem. $b(M)$ is true if...?

Example: Robot Moves

- $b(M)$ is true if we can correctly redirect the robot by replacing a subsegment of size M.
- We need to do this in around $O(N)$ now. How? It's worth considering how to do it in $O(1)$ if I tell you exactly what subsegment to replace.
- Reduces to, given a list of $N - M$ instructions, can I add M more instructions to get the robot to position X.

Example: Robot Moves

- **Key Observation:** In M instructions, the robot can move to every square within distance M.
- So we are reduced to finding if there is a subsegment of size M such that its removal leaves the robot within distance M of X.
- Now we just need to find where the robot is after the removal of each subsegment of size M.
- For this, we precompute a cumulative sum array from the front and back, where `L` is -1, `S` is 0 and `R` is 1.
- Then the position of the robot after removing the segment $[L, L+M)$ is `sum[0,..,L-1] + sum[L+M,..,N-1]`.

Example: Robot Moves


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

const int MAXN = 200000;
int N, X;
char moves[MAXN+5];
// cumFront[i] = sum moves[1..i]
int cumFront[MAXN+5];
// cumBack[i] = sum moves[i..N]
int cumBack[MAXN+5];

void precomp() {
    vector<int> moveDeltas(N+5, 0);
    for (int i = 1; i <= N; i++) {
        if (moves[i] == 'L') moveDeltas[i] = -1;
        if (moves[i] == 'S') moveDeltas[i] = 0;
        if (moves[i] == 'R') moveDeltas[i] = 1;
    }
    for (int i = 1; i <= N; i++)
        cumFront[i] = cumFront[i-1] + moveDeltas[i];
    for (int i = N; i >= 1; i--)
        cumBack[i] = cumBack[i+1] + moveDeltas[i];
}
```

Example: Robot Moves


```
bool canDo(int A) {
    for (int i = 1; i+A-1 <= N; i++) {
        // try replacing [i, i+A-1]
        int posAfterCut = cumFront[i-1] + cumBack[i+A];
        if (abs(posAfterCut - X) <= A) return true;
    }
    return false;
}

int main() {
    scanf("%d %d", &N, &X);
    for (int i = 1; i <= N; i++) scanf(" %c", &moves[i]);
    precomp();
    int lo = 0;
    int hi = N;
    int ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        // Trying to find the lowest value that is feasible:
        if (canDo(mid)) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

Example: Robot Moves

- **Complexity:** $O(N \log N)$.
- Hopefully you can see the similarities between this example and the earlier example.
- Again, we started with a problem where approaching it directly was too slow.
- But the problem naturally could be rephrased as finding the minimum M such that a decision problem $b(M)$ was true.
- So from that point onwards we only consider the decision problem.
- This still required some work but was more direct. The idea of trying all subsegments of length M is relatively straight forward. From that point on it was just trying to optimize this problem with data structures.

Example: Quality of Living

- **Problem Statement:** Given a $R \times C$ rectangle of integers, find the minimum median of all rectangles of size $H \times W$.
- **Input Format:** First line 4 integers, R, C, H, W, $1 \leq R, C \leq 1000$. H, W are both odd. Next R lines each contain C integers describing the rectangle. Each value is distinct, between 1 and RC.
- **Source:** IOI 2010.

Example: Quality of Living

- **Sample Input:**

```
5 5 3 3
5 11 12 16 25
17 18 2 7 10
4 23 20 3 1
24 21 19 14 9
6 22 8 13 15
```

- **Sample Output:**

```
9
```

- **Explanation:** Take the rectangle with top left corner in the 3rd cell of the 2nd row. One can check this is optimal.

Example: Quality of Living

- Try to approach this problem directly.
- One can do it on a line by sweeping. But a rectangle seems difficult.
- Let's try to binary search. So the first step is to rephrase it as a decision problem.
- Natural decision problem: Is there a $H \times W$ rectangle with median at most A? Note the appearance of *at most* again.

- **New Problem:** Is there a $H \times W$ rectangle with median at most A?
- How do we take advantage of the fact A is fixed?
- Think about how median is defined. What does it mean for median to be $\leq A$?
- It means at least half the elements are $\leq A$.
- So we are reduced to counting the number of elements $\leq A$ in each subrectangle of size $H \times W$.
- Again, we just need to know the subrectangle with the most elements $\leq A$.
- At this stage, we only care which elements are $\leq A$. Let's say those are 1s and the other elements are 0s.

Example: Quality of Living

- **New Problem:** Given a grid of 0s and 1s, what is the maximum sum of a subrectangle of size $H \times W$.
- Comes down to querying the sum of a subrectangle quickly. Hopefully you know how to solve the analogous 1D problem.
- We can generalize the approach directly. We can either use a 2D cumulative sum or 2 1D cumulative sums.
- I'm going to do the 2D cumulative sum approach. I think it's neat and it's worth learning, but it isn't the main purpose of this problem.

Example: Quality of Living

- In the 1D case, we have $\text{sum}[L, R] = \text{sum}[0, R] - \text{sum}[0, L-1]$.
- In the 2D case, we do an inclusion-exclusion. Let $\text{sum}[X1, Y1, X2, Y2]$ be the sum of the rectangle $X1 \leq x \leq X2, Y1 \leq y \leq Y2$. We get:

```
sum[X1,Y1,X2,Y2] = sum[0,0,X2,Y2]
                   - sum[0,0,X1-1,Y2]
                   - sum[0,0,X2,Y1-1]
                   + sum[0,0,X1-1,Y1-1]
```

- So we just need to calculate $\mathrm{sum}[0, 0, X2, Y2]$. This can be calculated in $O(RC)$ using the above formula, or a sweep top to bottom, left to right.
- The sweep calculates sum row by row. To calculate $\mathrm{sum}[0, 0, X, Y]$ you already have $\mathrm{sum}[0, 0, X, Y - 1]$ so you just need to add $\mathrm{sum}[0, Y, X, Y]$. This is just a 1D sum now.
- So we do rows in increasing order and each row from left to right. When processing the Y-th row, we maintain the sum of the row so far and add it to $\mathrm{sum}[0, 0, X, Y - 1]$ which we have already calculated.

Example: Quality of Living


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

const int MAXDIM = 1000;
int R, C, H, W;
// 1-indexed, this reduces special cases involving the 2D sum array.
int gr[MAXDIM+5][MAXDIM+5];
int sum[MAXDIM+5][MAXDIM+5];

// Returns if there is a H*W subgrid with median <= A.
bool canDo(int A) {
    for (int i = 1; i <= R; i++) {
        int cRowSum = 0;
        for (int j = 1; j <= C; j++) {
            cRowSum += (gr[i][j] <= A);
            sum[i][j] = sum[i-1][j] + cRowSum;
        }
    }
    for (int i = H; i <= R; i++) {
        for (int j = W; j <= C; j++) {
            int cSum = sum[i][j] - sum[i-H][j] - sum[i][j-W] + sum[i-H][j-W
                ];
            if (cSum >= (H*W)/2 + 1) return true;
        }
    }
    return false;
}
```

Example: Quality of Living


```
int main() {
    scanf("%d %d %d %d", &R, &C, &H, &W);
    for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= C; j++) {
            scanf("%d", &gr[i][j]);
        }
    }
    int lo = 1;
    int hi = R*C;
    int ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (canDo(mid)) {
            // Trying to find the lowest value that is feasible:
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

Example: Quality of Living

- Sometimes it is not obvious a binary search will help!
- But if it is an option, you may as well reduce to the decision problem.
- The decision problem may still be hard but it can't get any worse than the original problem.