# Problem-Solving Paradigms

## COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2025

# Table of Contents

- One approach to reduce the number of states explored by an algorithm is to simply make the best available choice at each stage, and never consider the alternatives
- This is known as a *greedy* strategy
- **General Principle:** Don't bother with states that will never contribute to the optimal solution!

- It is imperative that you prove (to yourself) that this process achieves the optimal solution, that is, it is not possible to beat the greedy strategy using a suboptimal choice at any stage.
- Look for a natural ordering of states
- For some problems, the greedy algorithm is not optimal, and we instead look to techniques such as dynamic programming

- **Problem statement** You are playing a 2-player game with $2 \leq n \leq 1000$ rounds. You and your opponent have $n$ different cards numbered from 1 to $n$. In round $i$, each player picks an unplayed card from their hand. The player with the higher card wins $i$ points (no points are given for draws).

  Through "psychology" you know exactly what cards your opponent will play in each round. What is your maximum possible margin of victory?

# Example problem: Psychological Jujitsu





- **Input** An integer $n$ and a permutation of 1 to $n$, the $i$-th term of which is the card your opponent plays in the $i$-th round.
- **Output** A non-negative integer, your maximum margin of victory assuming optimal play.
- **Source** Australian Informatics Olympiad 2006

# Example problem: Psychological Jujitsu



- **Example Input**

  3
  3 1 2

- **Example Output** 4

- **Explanation**: Play 1 2 3. You lose the first round ($-1$) but win the second and third ($+2$, $+3$).

# Example problem: Psychological Jujitsu





- Brute force? There are $n!$ possible play orders.
- But maybe we can eliminate many of these play orders as suboptimal.
- For this, it helps to imagine what a possible play order could look like.

# Example problem: Psychological Jujitsu





- Consider the round where the opponent plays card $n$.
- In such a round, we can either draw (play card $n$ too) or lose.
- If we lose, which card should we play?
- May as well play our worst card, 1.
- But now we can win every other round!

# Example problem: Psychological Jujitsu





- Okay, how about the play patterns where we play card $n$ and draw?
- Then it's like we're repeating the problem with $n-1$ in place of $n$.
- Unrolling this recursion, we now see, we can assume our play pattern is:
  - Pick a number $i$.
  - Draw all rounds with opponent card $> i$.
  - Lose the round with card $i$.
  - Win all rounds with cards $< i$.
- Only $n$ play patterns! Can simulate each in $O(n)$.
- Complexity is $O(n^2)$; approx $1,000,000$ operations.

# Example problem: Psychological Jujitsu



- **Implementation**

```
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 1010;
int opp[N];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) { cin >> opp[i]; }
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        // draw against > i, lose to i, win against < i
        int cur = 0;
        for (int j = 1; j <= n; j++) {
            // if (opp[j] > i) { cur += 0; }
            if (opp[j] == i) { cur -= j; }
            if (opp[j] < i) { cur += j; }
        }
        ans = max(ans, cur);
    }
    cout << ans << '\n';
}
```

# Example problem: Psychological Jujitsu





- **Moral:** One way to eliminate states is figure out conditions "good" states must satisfy. For this, it helps to consider a problem from different angles.
- Other angles would have worked too, e.g.
  - consider which round the opponent plays card 1
  - consider which round you play card $n$
  - etc.
- While we might intuitively prioritise the last few rounds, it turns out this is a red herring of sorts.

# 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 problem: Stabbing





- **Problem statement** You have a list of $n$ closed intervals, each with an integer start point and end point. For reasons only known to you, you want to stab each of the intervals with a knife. To save time, you consider an interval stabbed if you stab any position that is contained with the interval. What is the minimum number of stabs necessary to stab all the intervals?
- **Input** The list of intervals. $0 \leq n \leq 1,000,000$ and each start point and end point have absolute values less than $2,000,000,000$.
- **Output** A single integer, the minimum number of stabs needed to stab all intervals.

# Example problem: Stabbing

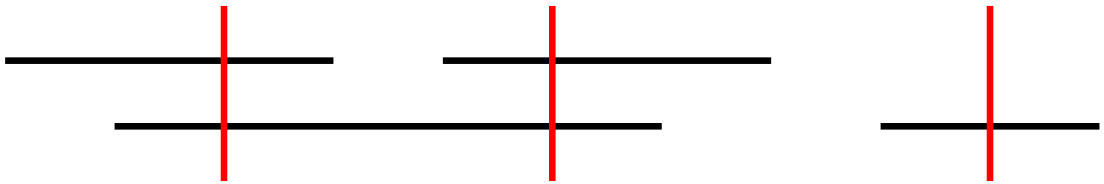


- **Sample Input**

```
4
0 3
1 6
4 7
8 10
```

- **Sample Output**

```
3
```



# Example problem: Stabbing





- How do we decide where to stab? State space is again laughably big.
- Again let's ask ourselves if we can eliminate many of the stab possibilities.
- Focus on a single stab for now.

- **Observation 1:** We can move it so it is an end point of an interval without decreasing the set of intervals we stab.

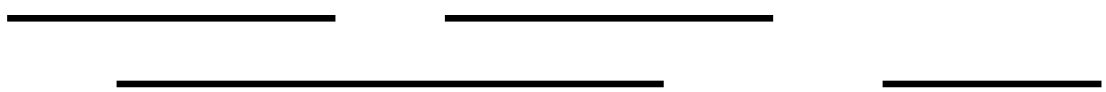
- **Proof:** Consider any solution where there is a stab *not* at the end point of an interval. Then we can create an equivalent solution by moving that stab rightwards until it hits an end point.

# Example problem: Stabbing

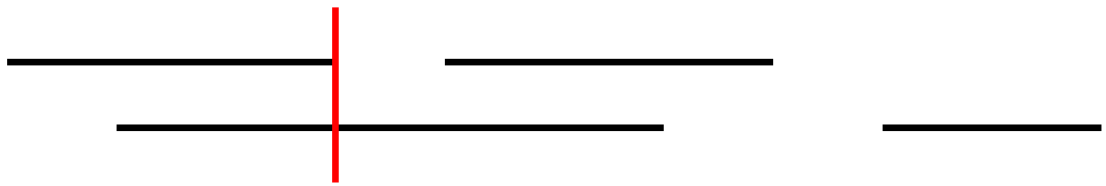




- Now let's try drawing sample data. Can we choose one of the stabbing points immediately?



- **Observation 2:** The first interval to end has to be stabbed somewhere. From **Observation 1**, we may as well stab at its end point.



# Example problem: Stabbing





- **Algorithm 1** Stab everything that overlaps with the first end point. Then, remove those intervals from the intervals to be considered, and recurse on the rest of the intervals.
- **Complexity** There are a few different ways to implement this idea, since the algorithm's specifics are not completely defined. But there is a simple way to implement this algorithm as written in $O(n^2)$ time.
- Can we do better?

- If we look closely at the recursive process, there is an implicit order in which we will process the intervals: **ascending by end point**

- If we sort the intervals by their end points and can also efficiently keep track of which intervals have been already stabbed, we can obtain a fast algorithm to solve this problem.

# Example problem: Stabbing





- Given all the intervals sorted by their end points, what do we need to keep track of? **The last stab point**
- Is this enough? How can we be sure we haven't missed anything?
- Since we always stab the next unstabbed end point, we can guarantee that there are *no unstabbed intervals* that are *entirely* before our last stab point.
- For each interval we encounter (in ascending order of end point), that interval can start *before/at* or *after* our last stab point.
  - If it starts before or at our last stab point, then it is already stabbed, so we ignore it and continue.
  - If it starts after our last stab point, then it hasn't been stabbed yet, so we should stab it and update the last stab point.

- **Algorithm 2** Sort the intervals by their end points. Then, considering these intervals in increasing order, we stab again if we encounter a new interval that doesn't overlap with our right most stab point.

- **Complexity** For each interval, there is a constant amount of work, so the main part of the algorithm runs in $O(n)$ time, $O(n \log n)$ after sorting.

- **Implementation**

```
#include <iostream>
#include <utility>
#include <algorithm>
using namespace std;

const int N = 1001001;
pair<int, int> victims[N];

int main() {
  // scan in intervals as (end, start) so as to sort by endpoint
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) { cin >> victims[i].second >> victims[i].first; }
  sort(victims, victims + n);

  int last = -2000000001, res = 0;
  for (int i = 0; i < n; i++) {
    // if this interval has been stabbed already, do nothing
    if (victims[i].second <= last) { continue; }
    // otherwise stab at the endpoint of this interval
    res++;
    last = victims[i].first;
  }

  cout << res << '\n';
}
```

# Example problem: Stabbing



- **Moral:** Sorting into a sensible order is often helpful. As is drawing pictures.

- I often find it helpful to play with a problem on paper and see how I would solve it manually for small cases.

- **Problem statement**

  There are $n \leq 2000$ countries, the $i$-th has $a_i \leq 20$ delegates.

  There are $m \leq 2000$ restaurants, the $i$-th can hold $b_i \leq 100$ delegates.

  For "synergy" reasons, no restaurant can hold 2 delegates from the same country.

  What's the minimum number of delegates that need to starve?

- **Input** An integer $n$ (the number of countries), and $n$ integers $a_i$ (the delegates of each country), followed by an integer $m$ (the number of restaurants), and $m$ integers $b_i$ (the capacity of each restaurant).
- **Output** A single integer, the minimum number of delegates that need to starve.
- **Source** Australian Informatics Olympiad 2007

# Example problem: Restaurants



- **Example Input**

  3
  4 3 3
  3
  5 2 3

- **Example Output** 2

- **Explanation**: Someone from the first country starves. Furthermore, the second restaurant has too few seats.

# Example problem: Restaurants



- Yet again, trying all assignments is laughably slow. So again, let us try to think about what conditions a good assignment may have?
- Makes sense to consider all delegates of a country at once so we don't have to keep track of who has been assigned where.
- Consider the countries in any arbitrary order. Suppose "Australia" is the first country we are considering.

# Example problem: Restaurants





- **Observation 1:** We should assign as many Australian delegates as possible.
- **Proof:** In any solution that does not, there is some restaurant with no Australian delegates and there is a starving Australian delegate.
- We can then kick out any delegate for an Australian delegate without making the solution any worse.
- But where should we assign the Australian delegates?
- Our main objective is to make it easier to seat the other country's delegates.
- From some extreme examples, the bottleneck seems to be the restaurants with few seats.

- **Observation 2:** We should assign delegates to the restaurants with the most seats remaining.
- **Proof:** Again, consider a solution that does not.
- Then when assigning Australian delegates, we skip restaurant $i$ in favour of a restaurant $j$ where $b_i > b_j$.
- Can we swap this Australian delegate from restaurant $j$ back to restaurant $i$?
- Need to avoid making a duplicate in restaurant $j$.
- Always possible since $j$ has fewer seats than $i$!
- By repeating these swaps, we obtain a solution just as good but also obeying **Observation 2**.

- Hence we need only consider allocations where Australia's delegates are assigned to the restaurants with the most seats remaining.
- Now assign the delegates of each other country in the same manner.
- One easy way to implement: Sweep through the countries one by one. For each country, sort the restaurants in decreasing capacity order and assign to them in that order.

- **Implementation**

```
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 2020, M = 2020;
int numDelegates[N], numSeats[M];

int main() {
    int n, m;
    cin >> n;
    for (int i = 0; i < n; i++) { cin >> numDelegates[i]; }
    cin >> m;
    for (int j = 0; j < m; j++) { cin >> numSeats[j]; }

    int starved = 0;
    for (int i = 0; i < n; i++) {
        int delegatesRemaining = numDelegates[i];
        sort(numSeats, numSeats+m, greater<int>());
        for (int j = 0; j < m; j++) {
            if (numSeats[j] > 0 && delegatesRemaining > 0) {
                numSeats[j]--;
                delegatesRemaining--;
            }
        }
        starved += delegatesRemaining;
    }
    cout << starved << '\n';
}
```

# Example problem: Restaurants





- **Complexity?** $O(n)$ countries. For each we sort a list of length $m$ and linear sweep through it.
- $O(nm \log m) \approx 2000 \times 2000 \times 11 = 4.4 \times 10^7$, fast enough.
    - We'll revisit this problem in the future to improve on this.
- **Moral:** One way to make observations is to think abstractly about what should hold. Often this is guided by examples.
- Once you have some guess, you can try to prove it after.

- 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.
- Example: range queries on a set of points. We 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.
- Use `pair` or `tuple` to associate compressed and uncompressed coordinates.
- Careful with equality: you might need `stable_sort()`.

- **Implementation (sort)**

```
#include <algorithm>
#include <iostream>
#include <utility>
using namespace std;

const int N = 100100;
int values[N]; // assume filled
int cmprsd[N]; // filled by compress()
pair<int,int> helper[N]; // initially blank
int n;

void compress(void) {
    for (int i = 0; i < n; i++) { // (uncompressed coordinate, original index)
        helper[i].first = values[i];
        helper[i].second = i;
    }
    stable_sort(helper,helper+n); // sort by uncompressed coordinate
    for (int i = 0; i < n; i++) {
        // overwrite uncompressed coordinates with compressed
        helper[i].first = i;
        // unsort using original index for reverse lookup
        cmprsd[helper[i].second] = i;
        // warning: unequal compressed coordinates assigned to equal
              uncompressed coordinates
        // this may or may not be desirable
    }
}
```

- **Implementation (map)**

```
#include <map>
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/...$y$.

# Aside: Binary Search Implementation





- Hands up if you've ever messed up a binary search implementation.
- Binary search is notorious for having annoying off-by-ones 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.

- **Implementation**

```
// Find the smallest X such that f(X) is true;
int binarysearch(function<bool(int)> f) {
    int lo = 0, hi = 100000, 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;
}
```

# Aside: Binary Search Implementation





- The best way to implement a binary search is not to implement it at all.
- If you are binary searching a range of explicit values, e.g. integers in a sorted array, use functions from `<algorithm>` to avoid bugs from writing your own.
  - `binary_search()`
  - `lower_bound()`
  - `upper_bound()`
  - `equal_range()`
- `set`, `multiset`, `map` and `multimap` have their own versions: see the Tips page of the course website.

# Aside: Binary Search Implementation





```
#include <algorithm>
#include <cassert>
#include <iostream>
using namespace std;

const int N = 100100;
int a[N];

int main () {
  int n;
  cin >> n;
  assert(n <= N);
  for (int i = 0; i < n; i++) { cin >> a[i]; }
  assert(is_sorted(a, a+n));

  int x;
  cin >> x;
  bool found = binary_search(a, a+n, x);
  cout << (found ? "found " : "did not find ") << x;
```

# Aside: Binary Search Implementation



```
  int y;
  cin >> y;
  int i = lower_bound(a, a+n, y) - a;
  if (i < n) {
    cerr << "a[" << i << "] = " << a[i] << " is the first entry to compare >= "
         << y;
  } else
    cerr << "all elements of a[] compare < " << y;
  }

  int z;
  cin >> z;
  int j = upper_bound(a,a+n,z) - a;
  if (j < n) {
    cerr << "a[" << j << "] = " << a[j] << " is the first entry to compare > "
         << z;
  } else {
    cerr << "all elements of a[] compare <= " << z;
  }
}
```

- Decision problems are of the form
  *Given the value x, can you ...*
- Optimisation problems are of the form:
  *What is the smallest x for which you can ...*
- An optimisation problem is typically *much* harder than the corresponding decision problem, because there are many more choices.
- Can we reduce (some) optimisation problems to decision problems?

- Let $f(x)$ be the outcome of the decision problem for a given $x$, so $f$ is an integer valued function with range $\{0, 1\}$.
- It is sometimes (but not always) the case in such problems that increasing $x$ does not make it any harder for the condition to hold, i.e. if it is true for $x$ then it is also true for $x + 1$.
- Thus $f$ is all zeros up to the first 1, after which it is all ones.
- This is a monotonic function, so we can use binary search!
- This technique of binary searching the answer, that is, finding the smallest $x$ such that $f(x) = 1$ using binary search, is often called *discrete* binary search.
- Overhead is just a factor of $O(\log A)$ where $A$ is the range of possible answers.

- **Problem Statement:** You have a bar of chocolate with $n$ squares. Each square has a tastiness $t_i$.

  You want to give a contiguous part of the bar to each of $k$ children. Each child will have happiness equal to the total tastiness of the squares they get.

  Maximise the happiness of the most unhappy child (i.e. minimise their crying).

- **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: Chocolate Bar



- **Sample Input:**

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

- **Sample Output:**

  ```
  14
  ```

- **Explanation:** Break the bar into the first two squares and the last three squares.

- It is worth trying to approach the minimisation 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.

# Example: Chocolate Bar





- We are asked to maximise 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 $x$.
- **Note:** the *at least* is important. If we instead required the sum of the smallest piece to be exactly $x$ then the function wouldn't be monotonic.
- Then the problem is asking for the largest $x$ such that $b(x)$ is True.

# Example: Chocolate Bar





- **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:** $b(x)$ implies $b(x-1)$, so $b(x)$ is non-increasing. Binary search!
- We still need to be able to calculate $b(x)$ quickly.
- **New Problem:** Can I split the bar into $k$ pieces, each with sum at least $x$?

# Example: Chocolate Bar





- **New Problem:** Can I split the bar into $k$ pieces, each with sum at least $x$?
- Note that we can rephrase this into a maximisation question. Given each piece has sum at least $x$, 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 $\geq x$.
- 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: Chocolate Bar





```
#include <iostream>
using namespace std;

const int N = 1001001;
long long bar[N];
int n, k;

bool canDo(long long x) {
    long long curhap = 0;
    int numpcs = 0;
    for (int i = 0; i < n; i++) {
        curhap += bar[i];
        if (curhap >= x) {
            if (++numpcs == k) { // early exit
                return true;
                    }
            curhap = 0;
        }
    }
    return false;
}
```

# Example: Chocolate Bar





```
int main() {
    cin >> n >> k;
    for (int i = 0; i < n; i++) { cin >> bar[i]; }
    long long lo = 1, hi = 1e12, 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;
        }
    }
    cout << ans << '\n';
}
```

# Example: Chocolate Bar





- **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 maximise a value.
- But we can rephrase it into maximising a value that satisfies a decision problem! In forming the decision problem, you ask if the answer could be *at least* $x$, not just exactly $x$.
- Now with the minimum tastiness of each bar fixed, you now switch to trying to maximise 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 $x$ 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 maximise the minimum of something (or minimise the maximum of something).
- Another way to see if it's useful is just to see if the quantity you are optimising is monotone.
- And this is very common! Usually, you are told to maximise a value because the problem is only more restricted as 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 optimisation problem (though it may lead you down a dead end).

- **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 $x$?

- **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 four 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 so long as we 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. $f(m)$ is true if ...?

- $f(m)$ 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 you're told exactly what subsegment to replace.
- Reduces to, given a list of $n - m$ instructions, can you 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 prefix and suffix sums, where `L` is $-1$, `S` is 0 and `R` is 1.
- Then the position of the robot after removing the segment $[i, i+m)$ is `sum[0,..,i-1]` (a prefix) plus `sum[i+m,..,n-1]` (a suffix).

# Example: Robot Moves





```
#include <iostream>
using namespace std;

const int N = 200200;
int n, x;
string moves;
// delta[i] = -1 if L, 0 if S, 1 if R
// pre[i] = sum of first i moves
// suf[i] = sum of last i moves
int delta[N], pre[N], suf[N];

void precomp() {
    for (int i = 0; i < n; i++) {
        if (moves[i] == 'L') { delta[i] = -1; }
        else if (moves[i] == 'S') { delta[i] = 0; }
        else if (moves[i] == 'R') { delta[i] = 1; }
    }

    for (int i = 1; i <= n; i++) { pre[i] = pre[i-1] + delta[i-1]; }
    for (int i = 1; i <= n; i++) { suf[i] = suf[i-1] + delta[n-i]; }
}
```

# Example: Robot Moves





```
bool f (int m) {
    for (int i = 0; i+m <= n; i++) {
        // try replacing [i, i+m)
        int posAfterCut = pre[i] + suf[n-(i+m)];
        if (posAfterCut >= x-m && posAfterCut <= x+m) { return true; }
    }
    return false;
}

int main() {
    cin >> n >> x;
    cin >> moves;
    precomp();
    int lo = 0, hi = n, ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        // Trying to find the lowest value that is feasible:
        if (f(mid)) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    cout << ans << '\n';
}
```

# 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 $f(m)$ 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 straightforward. From that point on it was just trying to optimize this problem with data structures.

# Aside: Ternary Search





- Ternary search also exists. It finds the maximum of a *bitonic* function: one that increases to a peak and decreases thereafter.
- Instead of splitting the range in two, we instead now split it into three by querying two points. At each step we discard one of the thirds based on comparison of the two points.
- Equal thirds is actually *not* the best decomposition. Exercise: what is?
- Alternatively, we can binary search the *finite difference*. This is the discrete form of the derivative, defined as $g(x) := f(x+1) - f(x)$.