

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

# Introduction

## COMP4128 Programming Challenges

School of Computer Science and Engineering  
UNSW Sydney

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

## Introduction

### Admin

### Classes

### Assessment

### Competitions and Practice

### Solving Problems

### Time Limit

### Greedy Algorithms

### Linear Sweep

### Binary Search

- Raveen de Silva (z3372617)
- Kevin Zhu (z5206576)
- Paula Tennent (z5255101)
- Angus Ritossa (z5311370)

Consultation: Tuesday 17:00 – 18:00.

If you want a private consultation, best to email me to set up a time first.

You can also contact us (and other students) on the [Ed forum](#).

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- To learn algorithms and data structures
- To practice fundamental problem solving ability
- To practice your implementation and general programming skills
- To prepare for programming competitions

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Significant programming experience in a C-like programming language
- Understanding of fundamental data structures and algorithms:
  - Arrays, structs, heaps, merge sort, BSTs, graph search, etc...
- COMP3121/3821
- Enthusiasm for problem solving

## Introduction

### Admin

#### Classes

#### Assessment

#### Competitions and Practice

#### Solving Problems

#### Time Limit

#### Greedy Algorithms

#### Linear Sweep

#### Binary Search

- 1 Introduction
- 2 Data structures
- 3 Dynamic programming
- 4 Graph algorithms
- 5 Network flow
- 6 Mathematics

There is a tentative course schedule on the website.

Introduction

1 Admin

Admin

2 **Classes**

Classes

3 Assessment

Assessment

4 Competitions and Practice

Competitions  
and Practice

5 Solving Problems

Solving  
Problems

6 Time Limit

Time Limit

7 Greedy Algorithms

Greedy  
Algorithms

8 Linear Sweep

Linear Sweep

9 Binary Search

Binary Search

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Lectures: (weeks 1–10)
  - Tuesday, 13:00 – 16:00
- Tute/Labs: (weeks 1–5, 7–10)
  - W13A: Wednesday, 13:00 – 16:00
  - W13B: Wednesday, 13:00 – 16:00
  - F10A: Friday, 10:00 – 13:00



## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- All times are in AEST (UTC+10) until the 3rd of October, then AEDT (UTC+11) thereafter
- All classes will be conducted online via Zoom
- In week 6 (flexibility week), we will have a revision lecture and no labs will take place

## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- Lectures for each topic will present the theory, and apply this to some example problems
- Any code in lectures will be in C++
- Slides will be available before each lecture
- Please ask questions at any time if anything is unclear

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- There are three hours of lab time assigned a week
- Your tutor will discuss one or two example problems related to the topic introduced in the most recent lecture, and demonstrate how to implement and test a solution
- You can spend the remaining time working on the problem sets by yourself or ask your tutor for help

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 1 Admin
- 2 Classes
- 3 Assessment**
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- Weekly problem sets: 40%
- Problem diary: 8%
- Contests: 18%
- Final: 34%

Marks are scaled generously in each assessment. Your performance relative to the cohort is of greater importance than your raw mark.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- A set of (usually) 5 problems will be released each week with a lecture
- Links will be posted on the [course website](#)
- You have  $\sim 2$  weeks to complete each set
- Worth 5% each, for a total of 40%
- Each problem in a set is weighted equally

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Up to 3 pages (excl code snippets) explaining your problem-solving process and any challenges you encountered
- Submit within 48h of problem set deadline, using give (TBC)
- Worth 1% each, for a total of 8%

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- At the end of week 1, you will undergo a timed contest with 5 problems, to be completed within 48 hours
- This first contest is intended to practice coding in a time-constrained environment, and does not require extensive technical knowledge
- We recommend that you try to complete the task within a shorter time frame, say 5 hours, but the full time is available in this case to minimise stress for you



## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- In weeks 5 and 9, you will undergo a timed contest with 3 problems, to be completed within 3 hours
- We will run up to 8 timeslots over a 24 hour period, to allow for time differences
- Further details will be released closer to the date of each contest

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- The final exam will be a timed contest with  $\sim 8$  problems, to be completed within 6 hours
- Further details will be released closer to the date of the exam

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice**
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

- ACM-ICPC
  - [South Pacific Programming Competition](#). Divisionals October 16! Register this week (email me) for free T-shirt.
  - [ANZAC League](#)
- Big companies
  - [Google Code Jam](#)
  - [Facebook Hacker Cup](#)
  - Microsoft Coding Competition (Probably around T1 at UNSW)
  - AmazAlgo (Probably around May at UNSW)
  - Will be announced on CSESoc Facebook page.
- Online competitions
  - [Atcoder](#)
  - [Codeforces](#)
  - [topcoder](#)
  - [CodeChef](#)

## Introduction

### Admin

### Classes

### Assessment

### Competitions and Practice

### Solving Problems

### Time Limit

### Greedy Algorithms

### Linear Sweep

### Binary Search

- The best practice is to solve lots of interesting problems
- Join [CPMSoc](#)
  - Fortnightly workshops
  - Other events including competitions
- Live contests
  - [ANZAC League](#)
- Online problem sets and competitions
  - AtCoder, Codeforces, TopCoder, CodeChef
  - [USACO](#), [ORAC](#)
  - [Project Euler](#)
- Or ask me or your tutor

## Introduction

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems**
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Read the problem statement
  - Check the input and output specification
  - Check the constraints
  - Check for any special conditions which might be easy to miss
  - Check the sample input and output
- Reformulate and abstract the problem away from the flavour text

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Design an algorithm to solve the problem
- Implement the algorithm
  - Debug the implementation
- Submit!



- **Problem statement** Alice and Bob are two friends who are visiting a milk bar. The milk bar is owned by the crotchety old Mr Humphries. If Alice buys  $A$  dollars worth of items and Bob buys  $B$  dollars, how much must they pay in total?
- **Input** Two integers,  $A$  and  $B$  ( $0 \leq A, B \leq 10$ )
- **Output** A single integer, the total amount Alice and Bob must pay.

Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- **Problem** Output  $A + B$
- **Algorithm** Calculate  $A + B$ , and then print it out.

- **Complexity**  $O(1)$  time and  $O(1)$  space
- **Implementation**

```
#include <iostream>

int main() {
    // read input
    int a, b;
    cin >> a >> b;

    // compute and print output
    cout << (a + b) << '\n';
    return 0;
}
```

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit**
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

## Introduction

### Admin

### Classes

### Assessment

### Competitions and Practice

### Solving Problems

### Time Limit

### Greedy Algorithms

### Linear Sweep

### Binary Search

- Your solution must give the correct output for each possible input, but it must also run within the specified time limit
- If you know your algorithm is not correct or too slow, then there is no point implementing or submitting it
- You can assess whether your algorithm is fast enough using complexity analysis
  - Calculate the number of states your algorithm will enter, and multiply by the amount of work performed in each state
  - Sometimes more sophisticated techniques are required, e.g. recursive algorithms
  - Your solution will not be accepted if it times out on even one test case, so assume the worst case input

## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- Modern computers can handle about 200 million primitive operations per second
- In some easy problems, the naïve algorithm will run in time
- If not, you can use a variety of techniques to reduce the number of states or the amount of work per state
- We'll see more advanced methods in future topics, e.g. data structures

- **Problem statement** Given an array of positive integers  $S$  and a window size  $k$ , what is the largest sum possible of a contiguous subsequence (a *window*) with exactly  $k$  elements?
- **Input** The array  $S$  and the integer  $k$   
( $1 \leq |S| \leq 1,000,000$ ,  $1 \leq k \leq |S|$ )
- **Output** A single integer, the maximum sum of a window of size  $k$

- **Algorithm 1** We can iterate over all size  $k$  windows of  $S$ , sum each of them and then report the largest one
- **Complexity** There are  $O(n)$  of these windows, and it takes  $O(k)$  time to sum a window. So the complexity is  $O(nk)$ . So we will need roughly around 1,000,000,000,000 operations in the worst case.
- This is way bigger than our 200 million figure from before! We need a way to improve our algorithm.



- What are we actually computing?
- For some window beginning at position  $i$  with a window size  $k$ , we are interested in  $S_i + S_{i+1} + \dots + S_{i+k-1}$

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- Let's look at an example with  $k = 3$
- We compute:
  - $S_0 + S_1 + S_2$
  - $S_1 + S_2 + S_3$
  - and so on

- **Algorithm 2** Instead of computing the sum of each window from scratch, we can use the sum of the previous window and just subtract off the first element, then add our new element to obtain the correct sum.
- To calculate  $W_i (= S_i + S_{i+1} + \dots + S_{i+k-1})$ , we can instead just do  $W_{i-1} - S_{i-1} + S_{i+k-1}$
- **Complexity** After the  $O(k)$  computation of the sum of the first window, each subsequent sum can be computed in  $O(1)$  time. Hence the total complexity of the algorithm is  $O(k + n) = O(n)$

## ● Implementation

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

const int N = 1e6 + 5;
int a[N];

int main() {
    // read input
    int n, k;
    cin >> n >> k;
    for (int i = 0; i < n; i++) cin >> a[i];

    long long ret = 0, sum = 0;
    for (int i = 0; i < n; i++) {
        // remove a[i-k] if applicable
        if (i >= k) sum -= a[i-k];
        // add a[i] to the window
        sum += a[i];

        // if a full window is formed, and it's the best so far, update
        if (i >= k - 1) ret = max(ret, sum);
    }

    // output the best window sum
    cout << ret << '\n';
    return 0;
}
```

- **Problem statement** In chess, a queen is allowed to move any number of squares horizontally, vertically or diagonally in a single move. We say that a queen *attacks* all squares in her row, column and diagonals.

		★			★		
			★		★		★
				★	★	★	
★	★	★	★	★	Q	★	★
				★	★	★	
			★		★		★
		★			★		
	★				★		

- For  $N \geq 4$ , it is always possible to place  $N$  queens on an  $N$ -by- $N$  chessboard so that no pair attack each other.

					Q		
			Q				
						Q	
Q							
							Q
	Q						
				Q			
		Q					

- **Input** The integer  $4 \leq N \leq 12$
- **Output** For each valid placement of queens, print out the sequence of column numbers, i.e. the column of the queen in the first row, the column of the queen in the second row, etc., separated by spaces and on a separate line.
- **Sample** For  $N = 6$ , the output should be:

```
2 4 6 1 3 5
3 6 2 5 1 4
4 1 5 2 6 3
5 3 1 6 4 2
```

- **Algorithm 1** Try placing queens one row at a time. The easiest way to do this is through recursion (sometimes this is called "recursive backtracking").
- We place queens one row at a time, by simply trying all columns, and then recurse on the next row. When  $N$  queens have been placed, we check whether the placement is valid.
- Complexity? Naively there are  $N^N$  placements of queens to check. We need to check if this queen duplicates any column or diagonal. This check takes  $O(N)$  time.
- Thus the naïve algorithm takes  $O(N^{N+1})$  time, which will run in time only for  $N$  up to 8.
- How can we improve on this?



- We need to cut down the search space;  $N^N$  is simply too large for  $N = 12$ .
- Many of the possibilities considered earlier fail because of conflicts within the first few rows — indeed, a single pair of conflicting queens in the first two rows could rule out  $N^{N-2}$  of the possibilities.
- Add pruning! Only recurse on *valid* placements, and simply discarding positions that fail before the last row.

- Algorithm 2** We place queens one row at a time, by trying all *valid* columns, and then recurse on the next row. When  $N$  queens have been placed, we print the placement.
- Unfortunately, as is typical of backtracking algorithms like this, it is difficult to formulate a tight bound for the number of states explored; there are theoretically up to

$$\frac{N!}{N!} + \frac{N!}{(N-1)!} + \dots + \frac{N!}{0!} < N \times N!$$

states, but in practice most of these are invalid. The true numbers turn out to be as follows:

$N$	8	9	10	11	12
states	15720	72378	348150	1806706	10103868

- Each state requires an  $O(N)$  check to ensure that the last queen does not share her column or diagonal.

## ● Implementation Gist

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

int n, a[12];

void go(int i) {
    if (i == n) {
        // we have placed all n queens legally, so print this solution
        for (int k = 0; k < n; k++) cout << a[k] + 1 << ' ';
        cout << '\n';
        return;
    }

    for (int j = 0; j < n; j++) {
        // TODO: check whether a queen can be placed at (i,j)
        if (can_place(j)) {
            a[i] = j;
            go(i+1);
        }
    }
}

int main() {
    cin >> n;
    go(0);
}
```

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms**
- 8 Linear Sweep
- 9 Binary Search

- 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?

- **Input** An integer  $N$  and a permutation of 1 to  $N$ , the  $i$ -th value is the card your opponent plays in the  $i$ -th round.
- **Output** A single integer, your maximum margin of victory assuming optimal play.
- **Source** [Orac](#)

- **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).

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 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.



- 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!

- 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  $O(N)$ . Total  $O(N^2) = O(1,000,000)$ .

## ● Implementation

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

const int MAXN = 1005;
int N, opp[MAXN];

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

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- **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: one could have considered the round the opponent plays card 1, or the round you played card  $N$ , etc...

## Introduction

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep**
- 9 Binary Search

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

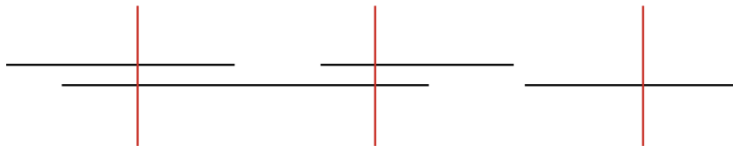
Linear Sweep

Binary Search

- 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!

- **Problem statement** You have a list of 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 within the interval. What is the minimum number of stabs necessary to stab all the intervals?
- **Input** The list of intervals,  $S$ .  $0 \leq |S| \leq 1,000,000$  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**



- The answer here is 3.



## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

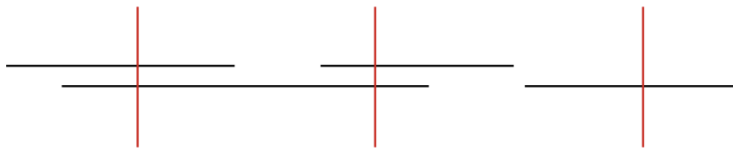
Linear Sweep

Binary Search

- 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 endpoint of an interval. Then we can create an equivalent solution by moving that stab rightwards until it hits an end point.

- Now let's try drawing sample data and consider moving from left to right. Where do we put our first stab?



- **Observation 2:** By Observation 1, we may assume it is at the first endpoint.

- **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  $O(|S|^2)$  time.

- 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.

- 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 next interval we encounter (iterating in ascending order of end point), that interval can start before or on/after our last stab point.
- If it starts before our last stab point, 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 do that.

- **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  $O(|S|)$  time,  $O(|S| \log |S|)$  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';
    return 0;
}
```



## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- **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.

- **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  $N$ ,  $N$  integers  $a_i$ . An integer  $M$ ,  $M$  integers  $b_i$ .
- **Output** A single integer, the minimum number of delegates that need to starve.
- **Source** [Orac](#)

- **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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

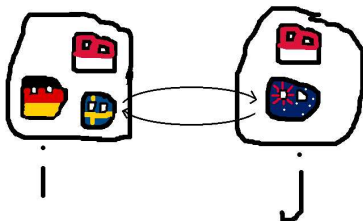
Linear Sweep

Binary Search

- 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.

- **Observation 1:** We should assign as many 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 we skip restaurant  $i$  for a restaurant  $j$  where  $b_i > b_j$ .
- But this means we can swap some delegate from restaurant  $i$  with the Australian delegate in  $j$  while preserving uniqueness.



- By repeating these swaps, we obtain a solution just as optimal except **Observation 2** was obeyed.

## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- Hence we may consider just solutions where Australia's delegates are assigned to the restaurants with the most seats remaining.
- Now repeat all other countries 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 <bits/stdc++.h>
using namespace std;

const int MAXN = 2005, MAXM = 2005;
int N, numDelegates[MAXN], M, numSeats[MAXN];

int main() {
    cin >> N;
    for (int i = 0; i < N; i++) cin >> numDelegates[i];
    cin >> M;
    for (int i = 0; i < M; i++) cin >> numSeats[i];
    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';
    return 0;
}
```



- **Complexity?**  $O(N)$  countries. For each we sort a  $M$  length list and then a linear sweep.
- $O(NM \log M) \approx O(4\text{mil} \cdot 11)$ , fast enough.
- **Moral:** One way to make observations is think abstractly about what should hold. Often this is guided by examples.
- Once you have some guess, you can try to prove it after.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

- 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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

```
#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];
    }
}
```

## Introduction

- 1 Admin
- 2 Classes
- 3 Assessment
- 4 Competitions and Practice
- 5 Solving Problems
- 6 Time Limit
- 7 Greedy Algorithms
- 8 Linear Sweep
- 9 Binary Search

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

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/...  $y$ .

## Introduction

## Admin

## Classes

## Assessment

## Competitions and Practice

## Solving Problems

## Time Limit

## Greedy Algorithms

## Linear Sweep

## Binary Search

- 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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

```
#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;
}
```



- Decision problems are of the form  
*Given some parameters including  $X$ , 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: that if the condition holds with  $X$  then it also holds with  $X + 1$ ).
- Thus  $f$  is all 0's up to the first 1, after which it is all 1's.
- 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 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  $i$ th piece. For all  $i$ ,  $1 \leq i \leq 100,000$ .

- **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.

- 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.

- **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  $A$ ?

- **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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

Binary Search

```
#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;
}
```



```
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);
}
```

- **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.

## Introduction

Admin

Classes

Assessment

Competitions  
and Practice

Solving  
Problems

Time Limit

Greedy  
Algorithms

Linear Sweep

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).

- 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!

- **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 4 instructions to get the sequence LLLLS.

- 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...?

- $b(M)$  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  $X$ .



- **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  $\text{sum}[0, \dots, L-1] + \text{sum}[L+M, \dots, N-1]$ .

```
#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];
}
```

```
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;
}
```

- **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.