Introduction

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Instructors

- Raveen de Silva (z3372617)
- Angus Ritossa (z5311370)
- Kevin Zhu (z5206576)
- LIC: Aleks Ignjatovic (ignjat)

Consultation: tentatively Thursday 11:00 – 12:00, happy to reschedule as per Moodle. Also feel free to email me, or ask questions after lectures.
If you want a private consultation, best to email me to set up a time first.

Course Goals

- 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

Prerequisites

- 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 (coreq)
- Enthusiasm for problem solving

Topic Overview

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.

Table of Contents

Timetable

- Lectures: (weeks 1–3, 5–10)
  - Monday, 18:00 – 21:00
- Labs: (weeks 1–5, 7–10)
  - Wednesday, 09:00 – 12:00
  - Wednesday, 12:00 – 15:00
  - Wednesday, 15:00 – 18:00
  - Friday, 09:00 – 12:00
  - Friday, 12:00 – 15:00

- All times are in AEST (UTC+10) until the 4th of October, then AEDT (UTC+11) thereafter
- All classes will be conducted online via Blackboard Collaborate Ultra
- No lectures will be held in week 4 due to Labour Day
- In week 6 (flexibility week), we will have a revision lecture and no labs will take place

Lectures

- 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

Labs

- 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
- In week 4, your tutor will introduce a small amount of new material on Binary Search and related concepts

Table of Contents

Assessment Breakdown

- Weekly problem sets: 48%
- 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.

Problem Sets

- A set of (usually) 5 problems will be released each week with a lecture
- Links will be posted on the course website
- You have ~ 2 weeks to complete each set
- Worth 6% each, for a total of 48%
- Each problem in a set is weighted equally

Contests

- At the end of week 2 (date TBA), 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

Contests (continued)

- In weeks 5 and 9 (tentatively), you will undergo a timed contest with 3 problems
- The contest will be available for 24 hours, but you will have only 3 hours in which to submit your solutions
- Further details will be released closer to the date of each contest

Final Exam

- The final exam will be a timed contest with ~ 8 problems
- The exam will be available for 24 hours, but you will have only 6 hours in which to submit your solutions
- Further details will be released closer to the date of the exam

Table of Contents

- ACM-ICPC
  - South Pacific Programming Competition. Divisionals October 17!
  - 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

Practice

- The best practice is to solve lots of interesting problems
- Live contests
  - UNSW ACM Training.
  - ANZAC League
- Online problem sets and competitions
  - AtCoder, Codeforces, TopCoder, CodeChef
  - USACO, ORAC
  - Project Euler
- Or ask me or your tutor

Table of Contents

How to Solve a Problem

- 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

How to Solve a Problem

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

Example problem: A + B

- **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.

Example problem: A + B

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

Example problem: A + B

- **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;
}
```

Table of Contents

Complexity and Time Limit

- 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

Complexity and Time Limit

- 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

Example problem: Max Sum Window

- **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

Example problem: Max Sum Window

- **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 $S_i + S_{i+1} + \ldots + S_{i+k-1}$

- 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 $W_i (= S_i + S_{i+1} + \ldots + S_{i+k-1})$, 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 $O(k + n) = O(n)$

Introduction
Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Time Limit
Greedy Algorithms
Linear Sweep

- **Implementation**

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

const int N = 1e6 + 5;
int s[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 -= s[i-k];
    // add a[i] to the window
    sum += s[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.

		$\star$			$\star$		
			$\star$		$\star$		$\star$
				$\star$	$\star$	$\star$	
$\star$	$\star$	$\star$	$\star$	$\star$	Q	$\star$	$\star$
				$\star$	$\star$	$\star$	
			$\star$		$\star$		$\star$
		$\star$			$\star$		
	$\star$				$\star$		

- 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					

Example problem: Queens

- **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
```

Example problem: Queens

- **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?

Example problem: Queens

- 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

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

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);
}
```

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?
- **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 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!

- 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;
}
```

- **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...

Table of Contents

Introduction

Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Time Limit
Greedy Algorithms
Linear Sweep

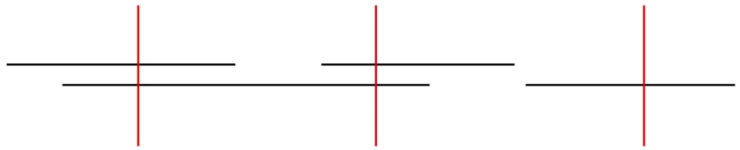
- Very basic but fundamental idea. Instead of trying to do a problem all at once, try to do it in some order that lets you build up state.
- This lets you process events one by one. This can be easier than trying to handle them all at once.
- **General Principle:** Having an order is better than not 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 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, 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 problem: Stabbing

- **Example**



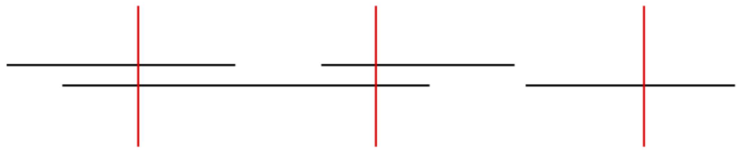
- The answer here is 3.

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 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.

Example problem: Stabbing

- 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 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**

```cpp
#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;
}
```

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.

Example problem: Restaurants

- **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 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.

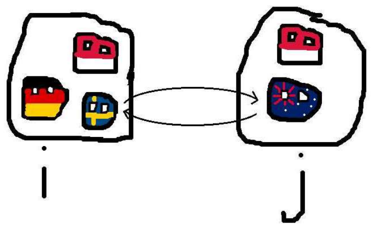
- Yet again, trying all assignments is laughably slow. So again, let us try to think about what conditions a good assignment may have?
- Makes sense to consider all delegates of a country at once so we don't have to keep track of who has been assigned where.
- Consider the countries in any arbitrary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

Example problem: Restaurants

Introduction
Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Time Limit
Greedy Algorithms
Linear Sweep

- **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.

Example problem: Restaurants

- 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.

Introduction
Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Time Limit
Greedy Algorithms
Linear Sweep

- **Implementation**

```cpp
#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.

- 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.

```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];
    }
}
```