Introduction

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Instructors

- Greg Omelaenko (z5116786)
- Clement Chiu (z5025019)
- Sahan Fernando (z5113187)
- LIC: Aleks Ignjatovic (ignjat)

Consultation: email, after lectures, etc. I don't have a room!

Course Goals

- To practice fundamental problem solving ability
- To learn algorithms and data structures
- 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
- COMP1917, COMP1927, COMP2911, COMP3121/3821 (coreq)
- Enthusiasm for problem solving

Topic Overview

1. Introduction
2. Graph algorithms
3. Shortest paths
4. Data structures
5. Dynamic programming
6. Mathematics
7. Computational geometry
8. Network flow
9. Strings

Table of Contents

Timetable

- Lectures:
  - Monday, 14:00 - 16:00 in Ainsworth 202
- Labs:
  - Friday, 14:00 - 17:00 in Strings Lab (J17 302)

- Lectures for each topic will present the basic theory, as well as some applications and example problems
- Any code in lectures will be in C++
- Slides will be available before each lecture
- Please feel free to ask questions if anything is unclear

Labs

- There are three hours of lab time assigned a week, to work on problem sets
- You may take this opportunity to ask for help, or just work on the problem sets by yourself
- In weeks 4, 8 and 12, we will hold 2.5 hour contests during lab time

Table of Contents

Assessment Breakdown

- Weekly problem sets: 36%
- Assignments: 24%
- Contests: 15%
- Final: 25%

Problem Sets

- A set of approximately 5 problems will be released most weeks (a full schedule can be found in the course outline)
- Links will be posted on the course website
- You have 2-3 weeks to complete each set
- Worth 4% each, for a total of 36%
- Each problem in a set is weighted equally
- You only need to complete 4 problems per set for full marks in that set, additional completed problems count for bonus marks
- These bonus marks only offset marks lost in contests or the final exam, they do *not* offset marks lost by not completing other problem sets, or marks lost in the assignments

Assignment 1

- Given a problem, write test data for it
- The goal is to think through all of the edge cases of the problem and come up with data that allows a correct solution to pass, but makes incorrect solutions produce wrong answers or time out
- Individual or in pairs
- Worth 9%
- More details will be available in the spec which will be released in the next few weeks

Assignment 2

- The same as assignment 1, but for a harder problem
- Ideally you should write (and submit) programs that can generate the test data, instead of producing it by hand
- Individual or in pairs, but not the same pair as assignment 1
- Worth 15%
- Due on the last day of the final exam period

Table of Contents

Competitions

- ACM-ICPC
  - South Pacific Programming Competition
  - ANZAC League
- Big companies
  - Google Code Jam
  - Facebook Hacker Cup
  - Microsoft Coding Competition (March 7th at UNSW)
- Online competitions
  - Codeforces
  - topcoder
  - CodeChef

Practice ("How to pass this course")

- The best practice is to solve lots of problems
- Live contests
  - UNSW ACM Training
  - ANZAC League
- Online problem sets and competitions
  - Codeforces, TopCoder, CodeChef
  - USACO, ORAC
  - Project Euler

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
- Reformulate and abstract the problem away from the flavour text

How to Solve a Problem

- Design an algorithm to solve the problem
  - It must be both correct and fast enough - complexity analysis
  - If you know your algorithm is not correct or too slow, then there is no point implementing or submitting it
  - Modern computers can handle about 200 million primitive operations per second
- 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.

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

Introduction

Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Example Problems

- **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;
}
```

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

Introduction

Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Example Problems

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

Example problem: Max Sum Window

- 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

Example problem: Max Sum Window

- **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 $O(k + n) = O(n)$

- **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++) {
    // add a[i] to the window, and remove a[i-k] if applicable
    if (i >= k) sum -= s[i-k];
    sum += s[i];

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

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

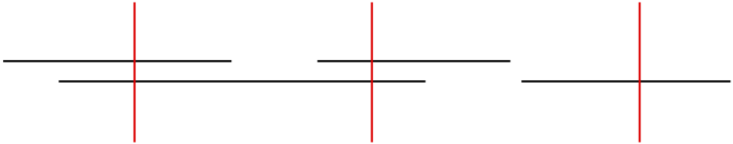
Table of Contents

Example problem: Stabbing

- **Problem statement** You have a list of 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?
- We can't do anything like a brute force enumeration of positions, because there are too many of them.
- We need to intelligently decide where to stab.
- What defines each interval?

- We only need to consider the end points of the intervals as possible candidates for stabbing positions.
- 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.
- Furthermore, combined with the fact that we must stab the first interval somewhere, we can say that we should always stab the first end point (the leftmost point at which any interval ends)

- **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.

- 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.

Introduction

Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Example Problems

- **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 = n; i --> 0;) cin >> victims[i].second >> victims[i].first;
  sort(victims, victims + n);

  int last = -2000000001, res = 0;
  for (int i = n; 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: Queens

- **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	$\star$	$\star$
				$\star$	$\star$	$\star$	
			$\star$		$\star$		$\star$
		$\star$			$\star$		
	$\star$				$\star$		

Example problem: Queens

- 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					

Introduction

Admin
Classes
Assessment
Competitions and Practice
Solving Problems
Example Problems

- **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, in lexicographic order.
- **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** 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.
- There are N squares for the queen in each row, so if we simply consider all possibilities, there are N^N placements of queens to check.
- Each placement must be checked for duplicates in any column or diagonal (note that we have already assigned exactly one queen per row). 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.
- We could improve by only recursing on *valid* placements, and simply discarding positions that fail before the last row.

Example problem: Queens

- **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 then requires an $O(N)$ check to ensure that the last queen has been placed legally, by scanning her column and diagonals.

- **Implementation**

```
#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++) {
    // check whether a queen can be placed at (i,j)
    bool ok = true;
    for (int k = 0; k < i; k++) {
      if ((a[k] == j) || (i - k == a[k] - j) || (i - k == j - a[k])) {
        ok = false;
      }
    }
    if (ok) {
      // place queen and recurse
      a[i] = j;
      go(i+1);
    }
  }
}
```

Introduction

Admin

Classes

Assessment

Competitions and Practice

Solving Problems

Example Problems

- **Implementation (continued)**

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