Getting Started

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

Term 3, 2023

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

Getting Started
Running Time
C++
Debugging

- **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.

Example problem: Max Sum Window

- 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

- **Algorithm 2** Instead of computing the sum of each window from scratch, we can modify the sum of the previous window.
- To calculate

$$W_i = S_i + S_{i+1} + \ldots + S_{i+k-1},$$

we can instead evaluate

$$W_i = W_{i-1} - S_{i-1} + S_{i+k-1}.$$

- **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 $O(k + n)$, which we can simplify to $O(n)$ as $n \geq k$.

Example problem: Max Sum Window

- **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;
}
```

Example problem: Queens

- 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	⋆	⋆
				⋆	⋆	⋆	
			⋆		⋆		⋆
		⋆			⋆		
	⋆				⋆		

Example problem: Queens

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

					Q		
			Q				
						Q	
Q							
							Q
	Q						
				Q			
		Q					

Example problem: Queens

- **Problem statement** Given a board size N, list all the ways of placing N queens so that no two attack each other.
- **Input** An 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
```

Example problem: Queens

- **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?

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.
- 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.

Example problem: Queens

- 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 print_queens() {
  for (int k = 0; k < n; k++)
    cout << a[k] + 1 << ' ';
  cout << '\n';
}

bool check_queen (int i, int j) {
  for (int k = 0; k < i; k++) {
    if ((a[k] == j) || (i - k == a[k] - j) || (i - k == j - a[k])) {
      return false;
    }
  }
  return true;
}
```

Getting Started

Running Time

C++

Debugging

- **Implementation (continued)**

```
void go(int i) {
  if (i == n) {
    // we have placed all n queens legally, so print this solution
    print_queens();
    return;
  }

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

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

Example problem: Queens

- This problem is instructive, but in practice not a great contest problem.
- Why? Because there are only nine possible test cases!
- This allows you to hard-code the answer in your source code.
- But how do you obtain the answer in order to hard-code it?
- You still have to write a solver, but you could let it run locally for several minutes per test case while you work on another problem.

Table of Contents

C++ Documentation

- You can find C++ documentation at cplusplus.com or cppreference.com.

- cplusplus.com is easier to read for a non-expert, whereas cppreference.com is more thorough and technical.

- These are the only online resources that you can use during the contests, other than the course website and associated platforms.

C++ Standards

- The original C++ standard was C++98.
- All problems in the weekly problem sets will be judged using C++11 or more recent versions.
- In the contests, your submissions will be compiled using the C++17 standard.

C++11 Features for Competitive Programming

- C++11 has lots of nice things for us.
- Very important:
  - `long long` width
- Situational
  - `unordered_set` and `unordered_map`
  - `random`
  - `tuple`
- Conveniences
  - `auto` type
  - range-based `for` loops
  - various functions in `<algorithm>`
- There's very little in the later C++ standards that affects us.

Naming conflicts

- You might want to use `count` as a function name, but the `<algorithm>` header also defines a function named `count`.
- When you call `count()`, how does the compiler know which one you are calling?
- Two C++ features are relevant here.
- Unlike C, C++ allows overloading of function names. If two functions with the same name have different signatures (e.g. different number of parameters, differently typed parameters), then the compiler may be able to resolve the conflict automatically.

Namespaces

- A namespace defines a named scope, within which you can define variables, functions and classes.
- You can then refer to those entities by prepending the name of the namespace, differentiating them from other entities with the same name that might be defined elsewhere in the project.
- In the example from the previous slide, the `<algorithm>` header is part of the C++ standard library, so all functions in this header belong to the `std` namespace.
- Therefore there is no conflict; `count()` refers to the local function, whereas `std::count()` refers to the one from `<algorithm>`.

- A common theme in competitive programming is that in competitive programming, you can (and often should) use lot of shortcuts and hacks that would be considered bad style or worse in other settings.
- Namespaces are useful, but they make us type five extra characters whenever we want to use something from the standard library, which is a nuisance.
- Instead, we include the global directive `using namespace std;` immediately after our `#include` directives. This tells the compiler to try looking in the `std` namespace whenever it tries to resolve a variable or function name.
  - Global `using` directives are a bad idea in most contexts! Don't do this in larger projects.

Namespaces

- Now, we can write `cout << ans << endl;` rather than `std::cout << ans << std::endl;`! Surely the precious seconds saved will give us the edge to win a contest.
- The downside of this is that we reintroduce naming conflicts. If we name a variable or function as `count`, `rank` or various other common words which name entities in `std`, the compiler might complain.
- No matter, we will just use names like `cnt` and `rnk` instead if necessary.

Namespaces

Getting Started

Running Time

C++

Debugging

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

// count nonzero array entries

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

int main (void) {
  int n;
  cin >> n;

  int count = 0;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    if (a[i] != 0)
      count++;
  }

  // check our answer
  assert(count == n-count(a,a+n,0)); // conflict!

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

- This causes a naming conflict.

```
In file included from /usr/include/c++/9/cassert:44,
                 from count-conflict.cpp:2:
count-conflict.cpp: In function ‘int main()’:
count-conflict.cpp:23:34: error: ‘count’ cannot be used as a function
   23 |   assert(count == n-count(a,a+n,0));
      |                                  ^
```

Namespaces


```
#include<algorithm>
#include<cassert>
#include<iostream>

// count nonzero array entries

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

int main (void) {
  int n;
  std::cin >> n;

  int count = 0;
  for (int i = 0; i < n; i++) {
    std::cin >> a[i];
    if (a[i] != 0)
      count++;
  }

  // check our answer
  std::assert(count == n-std::count(a,a+n,0)); // no conflict

  std::cout << count << '\n';
}
```

- This is the ‘good’ way to resolve the naming conflict, by omitting the `using` directive that caused the problem.

Namespaces


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

// count nonzero array entries

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

int main (void) {
  int n;
  cin >> n;

  int cnt = 0;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    if (a[i] != 0)
      cnt++;
  }

  // check our answer
  assert(cnt == n-count(a,a+n,0)); // no conflict

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

- This is the lazy way to resolve the naming conflict, by renaming our local variable.

Input and Output

- In every problem in the contests and almost every problem in the weekly problem sets, input is from the standard input stream `stdin` and output is from the standard output stream `stdout`.
  - We will provide specific instructions for any problem that requires you to read from and write to files.
- C-style I/O (`scanf`, `printf`, `getline`, and so on) is supported in the `<cstdio>` header.
- C++-style I/O (`cin`, `cout`) is supported in the `<iostream>` header.

Input

- If we know the format of the input (as is the case in contest problems), we can reliably extract it from `cin` using the >> operator.
- This is very convenient; it's fast to type, and doesn't require us to think about pointers and addresses.
- We can read an entire line into a string `s` using `getline(cin,s)`.
- There is a trade-off in speed. Reading input from `cin` is quite a bit slower than using `scanf()`.
  - If you need to read 10MB per second, consider including the following statements:

```
cin.tie(nullptr); // prevents cout from flushing on every cin read
cin.sync_with_stdio(false); // unsyncs iostream from cstdio
```

  - or just revert to `scanf()`.

Output

- We can similarly print output to `cout` using the `<<` operator.

- Almost always print newline characters as `'\n'` rather than `endl`, since the latter flushes the output buffer, making it slower.
  - Flushing is only necessary for interactive problems.

Floating-Point Output

- Suppose `x` is a `double`, and we run `cout << x << '\n'`. The default behaviour is to print to six digits of precision. This is often insufficient.
- Worse still, if `x` exceeds $1,000,000$ in absolute value, it will be printed in scientific notation (yes, really).
- To insist on fixed-point (not scientific) notation, use the manipulator `std::fixed` from the `<ios>` header, which is included within `<iostream>`.
- To change the number of decimal places, use the manipulator `std::setprecision()` from the `<iomanip>` header.
- Therefore, to print to 9 decimal places, we write `cout << fixed << setprecision(9) << x << '\n';`. Consider just using `printf()` instead!

Memory

- We will (almost) never need dynamic memory in programming contests, so you don't need to know about `malloc` or `free` (or the analogous C++ operators `new` and `delete`).
- Instead, we will use the input size defined in the problem to allocate as much (or more) static memory than we could ever need.
- Good habit to declare arrays slightly larger than necessary.
  - If the length is up to $100,000$, I will usually declare it with size $100,100$. This is helpful in case I want to store the elements off-by-one or similar.
- In C++, array sizes must be constant expressions, so we use the keyword `const`.

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

Global Variables

- Global variables are unsafe in larger projects, but we never use more than one file so they're fine for our purposes.
- This saves us some typing, since we don't have to explicitly pass them in function calls.
- Global arrays are stored on the heap.
  - Create a large array (say 10^7 integers for a prime sieve) on the heap is fine.
  - However, it might not be possible to allocate enough contiguous memory on the stack.
- A convenience: global variables are initialised to the default value, so an integer array will be 0-initialised (rather than garbage values).

Integer Types

- `int` is a 32-bit integer. Be wary of overflow.
- Since C++11, `long long` is a 64-bit integer. On a 64-bit processor, the performance difference is usually negligible.
- Never use `long`. The language standard does not enforce whether it is 32 or 64 bits.
- You won't need 128-bit or arbitrary-precision integers in this course.

Floating-Point Types

- Never use `float`. The extra precision from `double` is necessary in most problems, and the performance difference is again negligible.
- You won't need `long double` in this course.
- Problems which require you to produce floating-point output will typically allow answers within some relative and/or absolute error. This will always be specified in the output format.

String Types

- C-style character arrays still work, and the `<cstring>` header corresponds to the C library `<string.h>`.
- We will usually use the `string` type instead, which integrates directly with C++-style I/O and various extra functions provided in the `<string>` header.
- To access the character array underlying string `s`, we use `s.c_str()`. This is useful for C-style formatted printing.

Converting Between Types

- You can convert between types using explicit or implicit casting, as in C.
- C++ also introduces the `stringstream`, which you can both insert to (<<) and extract from (>>).
- This is very useful for converting between strings and other types.
  - For example, you can read a string using `cin`, insert it to a stringstream, and then extract integers from there.
  - In other situations, `stoi()` might suffice.

- The `<utility>` header defines the class template `pair<T1,T2>`, which allows you to couple together two elements as a single unit.
- Since C++11, this has been generalised to `tuple<...>`, which supports any fixed-size collection of elements.
- The elements of a `pair` or `tuple` can be of any type, and they *do not* have to all be of the same type.
- This is extremely useful. For example, it lets you sort items while keeping track of their original indices, by making pairs of the form `(a[i],i)`.

Structures

- C++ introduces classes, but we won't use them.
- It also supports C-style structs, which we will sometimes use.
- You can define a function as a member of a struct, which we will occasionally use in this course.
- You can also define operators such as `operator==` and `operator<` for your structs, e.g. so that you can sort a collection of them.

Algorithms

- The `<algorithm>` header provides many useful functions for working on ranges of elements.
- Many of these (e.g. `fill()`, `search()`, `count()`, `max()`, `min()`) are just conveniences that replace two or three lines of very simple code.
- Even so, the more code you write yourself, the more bugs you can introduce.
- Swapping values is notoriously inconvenient in C, but in C++ we can just call `swap(x,y)`.

- The `<algorithm>` header includes a `sort()` function, which takes iterators (think generalised pointer) to the range to be sorted (left-inclusive, right-exclusive).
- Since C++11, this sorting algorithm is guaranteed to run in $O(n \log n)$ time in the worst case.
- This function takes an optional third argument, in which you can provide a custom comparison to be used in place of the default (`operator<`).
    - The default comparison for pairs is to compare the first entries, with ties broken by comparing the second entries.
- For example, to sort the first `n` entries of an integer array a in *descending* order, we can call `sort(a,a+n,greater<int>())`.

Shuffling and Searching

- Sometimes, sorted input is the worst case! Use `random_shuffle()`.

- `<algorithm>` also helps you avoid writing some binary searches from scratch. We'll discuss `binary_search()` and related functions in the Paradigms lecture.

Next Permutation

- `next_permutation()` rearranges the elements in a range into the lexicographically next greater permutation.
- For example, it would transform the character array `"permutation"` to `"permutatnio"`.
- It returns `true` if such a permutation exists, or `false` if the original permutation was already lexicographically greatest.
- An individual call to `next_permutation()` could take linear time, but the complexity is *amortized constant*!
- Why do we care about this peculiar function?

Next Permutation

- `next_permutation()` helps us run exhaustive search (i.e. brute force).
- The following snippet iterates through all bit sequences of length n in which exactly k bits are set.

```
int selected[n];
fill(selected,selected+n-k,0);
fill(selected+n-k,selected+n,1);

do {
    // ...
} while (next_permutation(selected,selected+n));
```

- Each of these sequences then corresponds to a different unordered selection of k items from a collection of n items.
- You can designate the selected items as those corresponding to set bits.

Table of Contents

Debugging

- Debugging is often the most difficult, time-consuming and frustrating part of competitive programming.
- The best way to debug is not to make bugs in the first place.
  - Take your time (especially in the weekly problem sets) and read what you're typing
  - In team contests, pair programming helps

How to Debug (pre-submission)

- Suppose you have solved the problem conceptually and implemented your algorithm, but your program fails the sample test cases. What should you do?
- *Run unit tests* to narrow down what part of your program is malfunctioning.
- The most primitive approach to debugging is to print heaps of logs as you go.
  - Print out the intermediate states of anything that might be important using `cerr` (or equivalently `fprintf(stderr,...)`).
- You may also want to look into more sophisticated debugging tools.
  - GDB
  - Some IDEs have an inbuilt debugger

How to Debug (post-submission)

- Suppose your program now passes the sample test cases, but your submission was unsuccessful. What should you do?
- What verdict did you get?
  - `TIME-LIMIT`
    - check your complexity analysis
    - look for infinite loops
  - `RUN-ERROR`
    - estimate how much memory you are using
    - check for array out of bounds
    - check for accessing into or deleting from empty data structures

How to Debug (post-submission)

- For `WRONG-ANSWER` and some `RUN-ERROR` verdicts, it's often difficult to even identify what the bug is.
    - Write test cases, including edge cases.
    - Force yourself to read your code carefully line-by-line, e.g. by opening it in a different editor or even printing out and annotating a hard copy.
- Write a slow (potentially brute force) algorithm that certainly produces the correct answer. You can then run it locally on medium to large cases, ignoring the time limit, and compare its answers to your other program's output.
- In the problem sets, ask for help – you can discuss this with other students and your tutor.
- If necessary, rethink your algorithm. Are there any hidden assumptions that you didn't examine carefully enough?

General Tips

- Read the tips page of the course website, which includes this advice and more.

- Try not to panic, especially in timed contests. Most of your bugs will be minor errors, perhaps even single character fixes.

General Tips

- Over the course of the term, there will probably be a couple of problems which you come very close to solving.
- This is a normal part of competitive programming, and it does average out in the long run.
- No single problem is worth many marks, so it's perfectly OK to give up on a problem sometimes.
- The problem diary gives you a chance to reflect on your efforts.
- In the weekly problem sets, make sure to also prioritise your other tasks and responsibilities.
- That said, if you can afford to sink days into a problem, there are few feelings quite as gratifying as finally receiving a `CORRECT` verdict on your 70th submission!