The programming assignments are meant to help you (a) cement your understanding of the algorithmic concepts you learn in lecture, and (b) ensure that you can translate them into correct and fast programs. The programs are autograded on a set of test cases, of varying complexity and sizes. The suite of test cases is intended to push your program to the limits, to make sure it is correct, it scales, and that it is faithful to other aspects of the specifications. It is important that you have the concepts right, but also that you pay attention to the details.

This writeup tells you about the submission process, but also has a list of suggestions for how to debug your programs, and other dos-and-donts for programming assignments. Hope you find it useful!

1 Submitting to Autolab

When submitting your assignment to Autolab, please create a tar of your solution file that is named according to the specific assignment name. For purposes of this handout we'll use a fictitious name "permcrusher". So in this case you would name your submission `permcrusher.c`, `permcrusher.cpp`, `Permcrusher.java`, `permcrusher.ml`, `permcrusher.rs`, or `permcrusher.sml`. If you need to include additional files, please ensure they compile according to our rules in the Requirements section, and then include them in the tar file.

For example, if you are using C for this assignment, then create your tar file with the following command: `tar cvf handin.tar permcrusher.c`.

2 Requirements

Your solution should accept input from `stdin` and write to `stdout`, in the format described in the problem statement. (See section 4 for more info on reading and writing to `stdin` and `stdout`.)

You can write your solution in any of the following languages:

- C
  - Compiled with: `gcc -std=c11 permcrusher.c -lm`
  - Autolab will use `gcc` version 9.3.1.
  - `gcc` on `unix.andrew.cmu.edu` is version 9.3 0.
- C++
  - Compiled with: `g++ -std=c++17 permcrusher.cpp -lm`
  - Autolab will use `g++` version 9.3.1.
  - `gcc` on `unix.andrew.cmu.edu` is version 9.3 0.
- Java
  - Compiled with: `javac -Xlint:unchecked Permcrusher.java`
  - Autolab will use OpenJDK version 11.0.10.

- OCaml
  - Compiled with: `ocamlopt permcrusher.ml`
  - Autolab will use OCaml version 4.05.
  - The ocaml on `unix.andrew.cmu.edu` is 4.08.1.
- Rust
  - Compiled with: `rustc permcrusher.rs`
  - Autolab will use Rustc version 1.49.0.
  - The rust version on `unix.andrew.cmu.edu` is 1.51.0.
- SML
  - Compiled with: `mlton permcrusher.mlb`
  - Autolab will use MLton version 20210117
  - The MLton version on `unix.andrew.cmu.edu` is the same.
  - If you don't include a `permcrusher.mlb` file in `handin.tar`, the `mlb` file will be automatically generated by Autolab, and contain the following:

    ```
    $(SML_LIB)/basis/basis.mlb
    $(SML_LIB)/basis/mlton.mlb
    $(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb
    permcrusher.sml
    ```

  - These libraries will most likely include everything you need for the programming assignments. So, simply including `permcrusher.sml` in your `handin.tar` file should be sufficient.
  - If you'd like to use libraries besides the ones included by default, you can provide your own `permcrusher.mlb` file, and tar it (along with `permcrusher.sml`) into `handin.tar`. Just remember to include `permcrusher.sml` in `permcrusher.mlb`. List of available libraries here: `http://mlton.org/MLBasisAvailableLibraries`

The time in seconds (and in some cases memory limits) for your program will be specified in the homework assignment handout. These limits are usually generous enough to accept most reasonable solutions to the problem.

The homework handout may also ask you to write a short description of your algorithm, and/or its analysis in a comment at the top of your source file.

3 Autograding

The grader will either compare the output of your program against a reference output or process the output of your program to verify it is a correct solution. There are often many test cases of varying size, and some tests for edge cases. Part of your score on a given assignment will be a function of the number of test cases you pass. (It could be proportional, or all or nothing, depending on the assignment.)

A class-wide scoreboard is available on Autolab. The time taken for your program to run on each of the test cases will be measured, and the highest such run-time will be displayed on the scoreboard. Only your best (fastest by this measure) submission will be displayed. The scoreboard also shows which language was used. You can make up an anonymous user name for use on the scoreboard.

4 Examples in your Language of Choice

The following examples all read input from `stdin` for the the fictitious "permcrusher" problem. In this case the input is a number `n` followed by n numbers in the range $[0, n-1]$. It then prints out the same information. (In most languages there are many ways to deal with text input and output. This code just illustrates one way to do it.)

```
5
0 4 1 3 2
```

4.1 C/C++

```
#include <stdio.h>

int a[1000000];

int main(){
    int i, n;
    scanf("%d", &n);
    for(i=0;i<n;i++) scanf("%d", &a[i]);
    printf("n = %d\n", n);
    printf("a = [");
    for(i=0;i<n;i++) printf(" %d", a[i]);
    printf("]\n");
    return 0;
}
```

4.2 Java

```
import java.io.*;
import java.util.*;
import java.lang.*;

public class Permcrusher {
    static int n;
    static int[] A;

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String line = br.readLine().trim();
        n = Integer.parseInt(line);

        line = br.readLine().trim();
        String[] l = line.split(" ");
        A = new int[n];
        for(int k = 0; k < l.length; k++) {
            A[k] = Integer.parseInt(l[k]);
        }
        System.out.printf("A = [");
```

```
        for(int i = 0; i<n; i++) System.out.printf(" %d", A[i]);
        System.out.printf("]\n");
    }
}
```

4.3 OCaml

```
open Printf
open Scanf

let read_int _ = bscanf Scanning.stdib " %d " (fun x -> x)
let () =
  let n = read_int () in
  let a = Array.init n read_int in
  printf "n = %d\n" n;
  printf "a = [";
  for i=0 to n-1 do
    printf " %d" a.(i)
  done;
  printf "]\n"
```

4.4 Rust

```
use std::io::{self, Read};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Preallocate buffer
    let mut buffer = String::with_capacity(80_000_000);
    let input = io::stdin();
    let mut input_lock = input.lock();

    input_lock.read_to_string(&mut buffer)?;

    let nums: Vec<i32> = buffer
        .trim_end()
        .split(&[' ', '\n'] as &[_])
        .skip(1) // Skip n, as we don't need it
        .map(|x| x.parse())
        // Return `Err(_)` if any parse fails
        .collect::<Result<_, _>>()?;

    // Print out what we read
    println!("{:?}", nums);

    Ok(())
}
```

4.5 SML

```
fun main () =
    let
    val input = TextIO.inputAll TextIO.stdIn
    val tokens = String.tokens Char.isSpace input

    val n = Option.valOf (Int.fromString (List.hd tokens))
    val a = map (Option.valOf o Int.fromString) (List.tl tokens)
    val _ = print "The input was:\n"
    val _ = print ("n = " ^ (Int.toString n) ^ "\n")
    val _ = print ("a = [")
    val _ = map (fn x => print (" " ^ (Int.toString x))) a
    val _ = print "]\n"
    in
        0
    end

val _ = main ()
```

5 Programming and Debugging Tips

1. Read the problem carefully! Make sure you are solving the correct problem and your output matches the format specified. Be careful about spelling mistakes, and other trivial formatting errors in the output.

2. We use `diff -wB output correctoutput` to compare, but if you have extra lines or other formatting differences, it may cause problems with the diff.

3. Your program needs to both run within the time limit and have the specified time complexity for full points.

4. Make sure your idea for the algorithm is in fact correct before debugging your implementation.

5. If we've released some test files, please try your algorithm on them. If not, please generate some test files (both small and large) and test your program. See how slow/fast it runs, and be sure to test on the example input we give you.

6. If the program times out, use a profiler (or just use print statements) to figure out where your program is spending all its time. Then speed up the slow part of your program. Similarly, if your program runs out of memory, think about where you can save on your memory usage.

7. In the case we don't give you a required time complexity, a good rule of thumb is that approximately 10^8 (trivial) operations can be done per second.

8. Some test cases are too large to allocate on the stack, so you should not use operations like `long A[1000000];` within functions/procedures. Instead, please use `malloc` (or equivalent operations) to allocate memory.

9. Similarly, avoid implementing algorithms recursively (when possible) to prevent stack overflow on large test cases, and to speed up execution.

10. *(Especially for Java)* Allocating fresh arrays and copying over data between arrays is slow, so consider swapping data in-place to speed things up.

11. To debug compilation issues you have to get into the same environment used by Autolab. You do that by ssh-ing to `unix.andrew.cmu.edu`, and trying to compile there.

12. The unix command `/usr/bin/time -v a.out < huge_test_case.txt` outputs very useful data about space and time usage. (More useful than just `time -v a.out < huge_test_case.in` which just calls the in-built shell command.)

13. Returning a non-zero value upon exiting is a signal that the program failed, so please make sure your program returns 0 when successful, otherwise Autolab may grade incorrectly. And if you do get a non-zero exit code, find out what it means. (E.g.: Exit code 139 means core dumped.)

14. If your program fails, read the Autolab output for why it failed. Information like `FINISHED` / `MEM` / `TIMEOUT` / `RUNTIME_ERROR` can be useful.

15. Consider the size of the numbers that you need to compute over. If they may be bigger than $2^{32} - 1$, use `long` integers.

16. Along the same lines, some languages support a type `float`, which is typically a 32-bit representation for floating point numbers. Our recommendation is never to use floats, because it only has about six digits of precision. Always use `double` for floating point operations.

17. For cases with large input/output, avoid using slower forms of I/O (so avoid `cin`/`cout` in C++ and avoid `Scanner` in Java).

18. **You are not allowed to use other people's code (from the internet or otherwise), with citations or otherwise. You may not search for solutions to problems, but you can search for error messages and debugging help.**

19. Please don't just ask the TA's and Professors for hints. Tell us what you tried, what failed, and we can try to suggest a way forward.

20. The intended solution should run within the allocated time for every supported language.

5.1 Programming and Debugging Tips for C/C++

1. Always use the `-Wall -Wextra` compile command line options to turn on warnings. This will catch many common problems, like forgetting to include a return statement in a function that's supposed to return a value, or forgetting to initialize a variable.

2. Use `gdb` (or other debuggers) to debug your program. To get source-level debug info, add the `-g` flag to `gcc`. (Ex: `gcc -g file.c` will allow `gdb` to tell you the source line on which your program segfaults.)

   There are many `gdb` tutorials out there. google "gdb tutorial"

3. If high accuracy is required, and you need to print, say 8 digits after the decimal point of a double precision variable, you can use `printf("%.8lf", x);`. Alternatively it can be done with with `setprecision`, as in:

   ```
   std::cout << std::setprecision(8) << x << std::endl;
   ```

4. For C++, know (and use) the STL. If you find yourself re-implementing some very basic algorithms/data structures, make sure that you actually need to. This pretty much applies to most of the standard libraries of our languages (except C and SML).