# Introduction to Programming Assignments

15-451/15-651: Algorithm Design and Analysis

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.

## 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`, `permcrusher.sml`, or `permcrusher.py`. 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 -O2 permcrusher.cpp`
  - 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`
- Python
  - Ran with `python3 -O permcrusher.py`
  - AutoLab will use Python version 3.6.8

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.

# 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 do it. This code just illustrates one way to do it.)

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

## 4.1 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 C++

```
#include <iostream>
#include <vector>

int main(){
    int i, n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int i = 0; i < n; i++) {
        std::cin >> a[i];
    }
    std::cout << "n = " << n << "\n";
    std::cout << "a = [";
    for (int i = 0; i < n; i++) {
        std::cout << " " << a[i];
```

```
    }
    std::cout << "]\n";
}
```

## 4.3 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");
    }
}
```

## 4.4 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"
```

## 4.5 Rust

```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(())
}
```

## 4.6 SML

```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 ()
```

### 4.7 Python

```
n = int(input())
A = list(map(int, input().split()))

print(n)
print(A)
```

# 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 **avoid** allocating arrays on the stack (e.g., writing `long A[1000000];` within a function in C). In C, instead, please use `malloc` (or equivalent operations) to allocate memory. In C++, just use `std::vector`.
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 (in C/C++, use the type `long long`, which is guaranteed to be at least 64-bits wide. Be aware that the type `long` differs by platform and is often only 32-bits wide on 32-bit systems!).

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. (e.g., avoid `Scanner` in Java). In C++, you should avoid using `std::endl` when outputting a large number of lines, since it flushes the output buffer which is slow if done too much. Instead, just output a newline character `"\n"`.

    In C++, `std::cin` and `std::cout` might be slow by default. If you are having issues with I/O speed, you can speed them up by adding the following magic instructions at the beginning of your main function:

    ```
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    ```

    The first instruction turns off synchronization between C++ and C's I/O streams, which is enabled by default (this would allow you to mix/interleave `std::cin` and `scanf`, for example, but I don't know why you would want to do this). The second instruction tells `std::cin` to not flush `std::cout`'s buffer every time it is used.

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. The time limit may be higher for Python if necessary.

## 5.1 Programming and Debugging Tips for C/C++

1. By default, C++ functions pass arguments by value, which means it copies them. Therefore, if you write a function like `f(std::vector<int> A)`, it will copy the vector, which could slow down your code substantially if you call the function repeatedly! If you do not wish to copy the argument, you should pass it by reference, which is achieved by decorating the type with an ampersand (&), so the signature would look like `f(std::vector<int>& A)`. Since reference parameters refer to the same object as the source argument, they can be modified from within the function. If you don't need to modify a reference parameter, it is a good habit to pass a constant reference, so that you can not accidentally modify it, like so `f(const std::vector<int>& A)`.

2. Error messages in C++ can be cryptic sometimes. If an error message from GCC doesn't make sense, you could try compiling your code with Clang, another popular open-source C++ compiler. I find that it often gives much better error messages than GCC. Clang is installed for you on `unix.andrew.cmu.edu`. The command line arguments for Clang are the same as for GCC, so just replace `gcc`/`g++` with `clang` and everything should work. e.g.

   ```
   clang -std=c++17 permcrusher.cpp
   ```

3. 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.

4. Enable **sanitizers** while debugging your program. They will catch bugs such as going out of bounds of an array, using memory after it has been freed, dereferencing a null pointer, etc. In general, they can usually catch the cause of just about anything that would cause a segfault. To enable them and get the best possible error messages, add the following flags to GCC/Clang

   ```
   -g -O0 -fsanitize=address,undefined -fno-omit-frame-pointer
   ```

   When the sanitizers trip and catch a bug, the output can look a little scary. They tend to print the contents of memory around which the error occurred, so you'll often see a giant array of bytes output to the screen. Just ignore this and look for the stack trace, which should indicate the line number on which the bug originated.

5. 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"

6. 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;
   ```

7. For C++, know (and use) the standard library. 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). For example, in C++, it is worth knowing and using (when appropriate):

   - `std::vector` - A dynamically sized array
   - `std::set` - Stores elements in sorted order, supporting $O(\log n)$ insert / delete / lookup
   - `std::map` - Stores key/value pairs. Supports $O(\log n)$ insert / delete / lookup
   - `std::priority_queue` - Represents a priority queue. $O(\log n)$ insert and get min.
   - `std::sort` - Sorts a range of elements in $O(n \log n)$
   - `std::binary_search` - Searches for a key in a sorted range in $O(\log n)$
   - `std::lower_bound` - Finds the first element at least as large as the given key in a sorted range in $O(\log n)$

   This is just a small subset of the standard library. In general, if you have to solve a simple generic problem, check whether it exists in the standard library **before** trying to implement your own version. This will save you time and bugs.