inzva Algorithm Programme 2018-2019

Bundle 1

Intro

Editor
Muhammed Burak Buğrul

Reviewers
Kadir Emre Oto
Yusuf Hakan Kalaycĭ

Contents

1 Introduction

First of all, this is an intensive algorithm programme prepared by inzva, which includes lectures, contests, problem-solvings and a variety of practises. "Competitive Programming" term will be mentioned frequently in this programme, especially for it's community, help in progress in algorithms and data structures etc.

Just for a quick cover up we will have a look at what happens when you compile and run a program, basic data types and functions. After that, we will examine C++ Standard Template Library(STL), time complexity and memory space.

2 Command Line

A lot of people don't use command line if they can use an alternative. There are powerful IDEs (Integrated Development Environments) and they really make programming easier in some aspects. However, knowing how to use command line is important, especially for competitive programming. Firstly, it gives a low level knowledge and full control; secondly, every computer and environment has command line interface.

In this document, you will find only a basic introduction to the command line, which is no more than the basic usage of a file system, compiler, and programs.

There are a lot of differences between command line of Windows and Linux. But the differences between those of Mac and Linux are less.

2.1 Linux and Mac

Mac users can use the built-in Terminal. You can find it by searching from Spotlight. Linux users can use gnome-terminal or any other installed one. Again, you can find them by using the built-in search tab.

Some basic commands:

- `ls` - list files in current directory. Usage: `ls`
- `cd` - change directory. Usage: `cd ~/Desktop`.
- `mkdir` - make a new directory. Usage: `mkdir directory_name`
- `mv` - move command(cut). Usage: `mv source_path destination_path`.
- `cp` - copy command. Usage: `cp source_path destination_path`

- `rm` - remove command. Usage: `rm file_path`

You can read more about the unix command line at:
http://linuxcommand.org

3 Compiling and Executing Programs

3.1 G++

G++ is installed in Linux environments but in Mac, you should install Xcode first.
You can compile your cpp souce file by typing `g++ source.cpp`. Default output of this command is `a.out`.

3.2 Running Executable Files

For Linux and Mac, the command to run a program is `./program_name`. If you use default g++ command, the name of your program will be `a.out`, so you should type `./a.out` in order to run it.

3.3 Closing a Program

When you want to kill a program in running step, you can simply hit *Control + C*.

When you want to suspend a program in running step, you can simply hit *Control + Z*.

When you want to register EOF on standart input, you can simply hit *Control + D*.

4 Input/Output Redirection

You can redirect input and output streams of a program by using command line and it is surprisingly easy.

4.1 Saving Output to a File

The only thing you need is > symbol. Just add it at the end of your run command with the output file name:
`./a.out > output.txt`

Note: This redirection process creates output.txt if it doesn't exist; otherwise deletes all content in it, then writes into it. If you want to use > in appending mode you should use » instead.

4.2 Reading Input from a File

It is almost the same as output file redirection. The symbol is < now. Usage:
`./a.out < input.txt`

This will make your job easier than copying and pasting input to test your program, especially in the contests.

4.3 Using Both at the Same Time

One of the wonderful things about these redirections is that they can be used at the same time. You can simply add both to the end of your run command:
`./a.out < input.txt > output.txt`

4.4 pipe

Sometimes, you may want to redirect the output of a program to another program as input. You can use the `|` symbol for this. Usage:
`./program1 | ./program2`

4.5 diff

As the name denotes, it can check two files line by line if they are the same or not. If not, it outputs different lines. Usage:
`diff file1.txt file2.txt`.

It is very useful for comparing output of brute force solution and real solution.

5 Structs and Classes

In almost every programming language, you can define your own data type. C++ has structs, classes; Python has dictionaries, classes etc. You can think of them as packets that store more than one different data and implement functions at the simplest level. They have a lot more abilities than these two(You can check OOP out).

Let us examine a fraction struct written in C++.

We need to store two values for a fraction, numerator and denominator.

```
struct Fraction {
  int numerator, denominator;
};
```

This is the simplest definition of a struct. Fraction struct contains two **int** variables. We call them members. So, Fraction struct has two members called numerator and denominator.

```
#include <cstdio>

struct Fraction {
    int numerator, denominator;
};

Fraction bigFraction(Fraction a, Fraction b) {

    if( a.numerator * b.denominator > a.denominator * b.numerator )
        return a;

    return b;
}

int main() {
    // Create two Fractions in order to compare them
    Fraction a, b;

    a.numerator = 15;
    a.denominator = 20;

    b.numerator = 12;
    b.denominator = 18;

    // Create a new Fraction in order to store biggest of Fraction a and Fraction b.
    fraction biggest = bigFraction(a, b);

    printf("The biggest fraction is %d / %d\n", biggest.numerator, biggest.denominator);
    return 0;
}
```

Let us do the same in Python3:

```
class Fraction:

    def __init__(self, numerator, denominator):
        self.numerator, self.denominator = numerator, denominator

def bigFraction(a, b):

    if a.numerator * b.denominator > a.denominator * b.numerator:
        return a

    return b

a, b = Fraction(15, 20), Fraction(12, 18)  # Create two Fractions in order to compare them
biggest = bigFraction(a, b)

print(biggest.numerator, biggest.denominator)
```

In the sample codes above, `a`, `b`, and `biggest` are called **objects** of `Fraction`. Also, the word **instance** can be used instead of object.

5.1 The Arrow Operator(C++)

Sometimes, usage of struct can change in C++. When you have a pointer to a struct, you should use `->` to access its members instead of `.` operator. If you still want to use `.` operator, you should do in this way: `(*ptr).member`. But arrow operator is simpler: `ptr->member`.

6 Big O Notation

When dealing with algorithms or coming up with a solution, we need to calculate how fast our algorithm or solution is. We can calculate this in terms of number of operations. Big O notation moves in exactly at this point. Big O notation gives an upper limit to these number of operations. The formal definition of Big O is[1]:

Let f be a real or complex valued function and g a real valued function, both defined on some unbounded subset of the real positive numbers, such that g(x) is strictly positive for all large enough values of x. One writes:

$$f(x) = O(g(x)) \; as \; x- > \infty$$

If and only if for all sufficiently large values of x, the absolute value of f(x) is at most a positive constant multiple of g(x). That is, f(x) = O(g(x)) if and only if there exists a positive real number

M and a real number x_0 such that:

$$|f(x)| \leq Mg(x) \ for\ all\ x\ such\ that\ x_0 \leq x$$

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that:

$$f(x) = O(g(x))$$

Almost every case for competitive programming, basic understanding of Big O notation is enough to decide whether to implement a solution or not.

Note: Big O notation can be used for calculating both the run time complexity and the memory space used.

7 Recursion

Recursion occurs when functions repeat themselves in order to create repeated applications or solve a problem by handling smaller situations first. There are thousands of examples in mathematics. One of the simple ones is *factorial* of n. It can be shown by $n!$ in mathematics and it gives the product of all positive integers from 1 to n, for example, $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. If we write factorial in a mathematical way, it will be:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{if } n > 0 \end{cases}$$

The reason why we didn't simply write it as $f(n) = n \cdot f(n-1)$ is that it doesn't give sufficient information about function. We should know where to end the function calls, otherwise it can call itself infinitely. Ending condition is $n = 0$ here. We call it *base case*. Every recursive function needs at least one base case.

So if we write every step of $f(4)$, it will be:

$$\begin{aligned} 4! &= 4 \cdot f(3) && \text{recursive step} \\ &= 4 \cdot 3 \cdot f(2) && \text{recursive step} \\ &= 4 \cdot 3 \cdot 2 \cdot f(1) && \text{recursive step} \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot f(0) && \text{recursive step} \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 && \text{base case} \\ &= 24 && \text{arithmetic} \end{aligned}$$

Basically, we can apply this recursive logic into programming:

```
int factorial(int n){

    int result = 1;

    for( int i=1 ; i<=n ; i++)
        res *= i;

    return result;
}
```

We can say a function is a recursive if it calls itself. Let us change this iterative factorial function into a recursive one. When you imagine how the recursive code will look like, you will notice it will look like the mathematical one:

```
int factorial(int n){

    if( n==0 )
        return 1;

    return n * factorial(n - 1);
}
```

Note that we didn't forget to put our base case into the recursive function implementation.

7.1 Time Complexity

In case above, it can be seen that both recursive and iterative implementations of factorial function runs in $O(n)$ time. But this equality doesn't occur always. Let us examine fibonacci function, it is mathematically defines as:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 0 \end{cases}$$

We can implement this function with just one for loop:

```
int fibonacci( int n ){

    int result = 1, previous = 1;

    for( int i=2 ; i<=n ; i++ ){
        int tmp = result;
        result += previous;
        previous = tmp;
    }

    return result;
}
```

Again, we can implement recursive one according to the mathematical formula:

```
int fibonacci( int n ){

    if( n == 0 || n == 1 )
        return 1;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Let us calculate time complexity of iterative one. There are three basic operations inside a for loop that repeats $n - 2$ times. So time complexity is $O(n)$. But what about the recursive one? Let us examine its recursion tree(diagram of function calls) for $n = 5$ on visualgo.

f() function called more than one for some values of n. Actually in every level, number of function calls doubles. So time complexity of the recursive implementation is $O(2^n)$. It is far away worse than the iterative one. Recursive one can be optimized by techniques like memoization, but it is another topic to learn in further weeks.

7.2 Mutual Recursion

Mutual recursion occurs when functions call each other. For example function `f` calls another function `g`, which also somehow calls `f` again.

Note: When using mutual recursions in C++, don't forget to declare one of the functions so that the other function can know first one from its' prototype.

Note 2: You can chain more than two functions and it will be still a mutual recursion.

7.3 Enumeration and Brute-Force

Enumeration is numbering method on a set.

For example, permutation is one of enumeration techniques. First permutation of numbers in range 1 and n is:

$$1, 2, 3...n-1, n$$

And second one is:

$$1, 2, 3...n, n-1$$

Finally, the last one is:

$$n, n-1...3, 2, 1$$

Additionally, we can try to enumerate all possible distributions of n elements into 3 different sets. An example of a distribution of 5 elements can be represented as:

$$1, 1, 2, 1, 3$$

In this distribution the first, the second and the fourth elements goes into the first set; third element goes into second set and the last element goes into the third set.

Enumerations can be done with recursive functions easily. We will provide example implementations of 3-set one. But before examining recursive implementation, let us try to implement iterative one:

```
#include <cstdio>

int main(){

    for( int i=1 ; i<=3 ; i++ )
        for( int j=1 ; j<=3 ; j++ )
            for( int k=1 ; k<=3 ; k++ )
                for( int l=1 ; l<=3 ; l++ )
                    for( int m=1 ; m<=3 ; m++ )
                        printf("%d %d %d %d %d\n", i, j, k, l, m);

    return 0;
}
```

It will print all possible distributions of 5 elements into 3 sets. But what if we had 6 elements? Yes, we should have added another for loop. What if we had n elements? We can not add infinite number of for loops. But we can apply same logic with recursive functions easily:

```
#include <cstdio>

int ar[100];

void enumerate( int element, int n ){

    if( element > n ){ // Base case

        for( int i=1 ; i<=n ; i++ )
            printf("%d ", ar[i]);

        printf("\n");
        return;
    }

    for( int i=1 ; i<=3 ; i++ ){
        ar[element] = i;
        enumerate(element + 1, n);
    }
}

int main(){
    enumerate(1, 5);
    return 0;
}
```

Brute-Force is trying all cases in order to achieve something(searching best, shortest, cheapest etc.).

One of the simplest examples of brute-forces approaches is primality checking. We know that for a prime P there is no positive integer in range $[2, P-1]$ that evenly divides P. We can simply check all integers in this range to decide if it is prime:

```
bool isPrime( int N ){

    for( int i=2 ; i<N ; i++ )
        if( N % i == 0 )
            return false;

    return true;
}
```

It is a simple function, but its' time complexity is $O(N)$. Instead we can benefit from the fact if there is a positive integer x that evenly divides N, there is a positive integer $\frac{N}{x}$ as well. As we know this fact, we can only check the integer in range $[2, \sqrt{N}]$:

```
bool isPrime( int N ){

    for( int i=2 ; i*i <= N ; i++ )
        if( N % i == 0 )
            return false;

    return true;
}
```

Now, its' time complexity is $O(\sqrt{N})$. It is far away better than $O(N)$.

8 Built-In Data Structures and Functions

There is no need to reinvent the wheel. Every language has its' own built in data structures, functions etc. After this point, the document will be C++ centered. But Python alternatives will be given.

8.1 The C++ Standard Template Library(STL)

The STL is a well known library for C++ that includes variety of data structures and algorithms.

Note: Third party libraries generally not allowed in contests.

8.2 Pairs

C++: Sometimes you may need to store two elements for an object. We can do this by creating a struct/class or two dimensional array. They will all work well but using pairs will be much more easier. You can think of pair as a class that has two variables named **first** and **second**. That's all for the basic. The good part is, you can decide their types(int, double, your own struct/class etc.). An example for pairs in C++:

```cpp
#include <iostream>

using namespace std;

int main(){

    pair<int, int> p(1, 2);
    pair<int, string> p2;

    p2.first = 3;
    p2.second = "Hey there!";

    pair<pair<int, string>, string> nested;

    nested.first = p2;
    nested.second = "This is a nested one";

    cout << "Info of p -> " << p.first << " " << p.second << endl;
    cout << "Info of p2 -> " << p2.first << " " << p2.second << endl;
    cout << "Info of nested -> " << nested.first.first << " " << nested.first.second
            << " " << nested.second << endl;

    return 0;
}
```

Python: You can simply create a tuple or an array:

```python
p = (1, 2)
p2 = [1, "Hey there!"]
nested = ((3, "inner?"), "outer", "this is a tuple you can add more")

p2[0] = 3
# nested[0] = "don't"  # In python you can't change tuples, but you can change arrays

print(p, p2, nested)
```

8.3 Vectors

C++: When using array, we should decide its size. What if we don't have to do this, what if we could add elements into it without considering the current size? Well, all these ideas take us to vectors.

C++ has this structure. Its name is vector. It is a dynamic array but you don't have to think about its size. You can simply add elements into it. Like pairs, you can use it with any type (int, double, another vector, your struct/class etc.). Usage of a vector is very similar to classic array:

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

using namespace std;

int main(){

    vector<int> ar;

    for( int i=0 ; i<10 ; i++ )
        ar.push_back(i);

    for( int i=0 ; i<(int)ar.size() ; i++ )
        cout << ar[i] << " ";

    cout << endl;
    return 0;
}
```

Python: Python lists already behave like vectors:

```
ar = []

for i in range(10):
    ar.append(i)

print(ar)
```

8.4 Stacks, Queues, and Deques

C++: They are no different than stack, queue and deque we already know. It provides the implementation, you can simply include the libraries and use them. See queue, stack, deque

8.5 Priority Queues

It is basically a built-in heap structure. You can add an element in $O(logN)$ time, get the first item in $O(logN)$ time. The first item will be decided according to your choice of priority. This priority can be magnitude of value, enterence time etc.

C++: The different thing for priority queue is you should add `#include <queue>`, not `<priority_queue>`. You can find samples here. Again, you can define a priority queue with any type you want.

Note: Default `priority_queue` prioritizes elements by highest value first. Here is three ways of defining priority.

Python: You can use heapq in python

8.6 Sets and Maps

C++: Now that we mentioned binary trees (heap above), we can continue on built-in self balanced binary trees. Sets are key collections, and maps are key-value collections. Sets are useful when you want to add/remove elements in $O(logN)$ time and also check existence of an item(key) in $O(logN)$ time. Maps basically do the same but you can change value associated to a key without changing the position of the key in the tree. You can check c++ references for set and map. You can define them with any type you want. If you want to use them with your own struct/class, you must implement a compare function.

Python: You can use dictionaries for map and sets for set in python without importing any other libraries.

8.7 Iterators

C++: You can use iterators for every built-in data structure in C++ for pointing their objects.

Python: You can iterate through any iterable in python by using **in**. You can check this example.

8.8 Sorting

C++: In almost every language, there is a built-in sort function. C++ has one as well. It runs in $O(NlogN)$ time. You can pass your own compare function into sort function of C++.

Python: You can use .sort() function for any list or list like collection in order to sort them or if you don't want to change the original collection, you can use sorted() function instead. You can pass your own compare function into sort function of python by using key variable.

9 Suggested Readings

9.1 C++

- **next_permutation:** Link.
- **STL document of Topcoder:** Link.
- **binary_search:** Link.
- **upper_bound:** Link.
- **lower_bound:** Link.
- **reverse:** Link.
- **fill:** Link.
- **count:** Link.

9.2 Python

- **bisect:** Link.
- **collections:** Link.

- **built-in functions:** Link.
- **lambda:** Link.

References

[1] Landau, Edmund (1909). Handbuch der Lehre von der Verteilung der Primzahlen [Handbook on the theory of the distribution of the primes] (in German). Leipzig: B. G. Teubner. p. 31.