IEEEXtreme Türkiye Camp 24' Program

Day 1

1 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 suÿciently 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 i s l eft u nstated, a nd o ne w rites m ore s imply 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.

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

Note:

2 Recursion

Recursion occurs when functions repeat themselves in order to create repeated applications or solve a problem by handling smaller situations first. T here a re t housands o f e xamples i n m a thematics. 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 suÿcien t 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.

2.1 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 f unction, i t is mathematically defines as:

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

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.

2.2 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 o ne f rom i ts' prototype.

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

2.3 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 di˙eren t 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:

```cpp
#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:

```cpp
#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 $O(N)$.

3 Linear Data Structures

3.1 Hash Map

A hash table or hash map, is a data structure that helps with mapping keys to values for highly efficient operations like the lookup, insertion and deletion operations.

A hash map is a concrete implementation of the abstract data type known as an associative array. In a hash map, keys are hashed to determine the index where the corresponding values will be stored, allowing for efficient retrieval and storage of key-value pairs.

This implementation typically provides fast access times for operations like insertion, deletion, and lookup of values based on their associated keys.

3.2 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.

3.3 Linked List

In a Linked List, elements are stored such that each element contains its own value and the address of the next element. The elements in the structure can be traversed from the head (first element) to the tail (last element). Compared to arrays, its advantage is the dynamic use of memory. The operations that can be performed on this data structure are:

- Adding an element to the end of the data structure.
- Traversing the current data structure from the head to the tail.

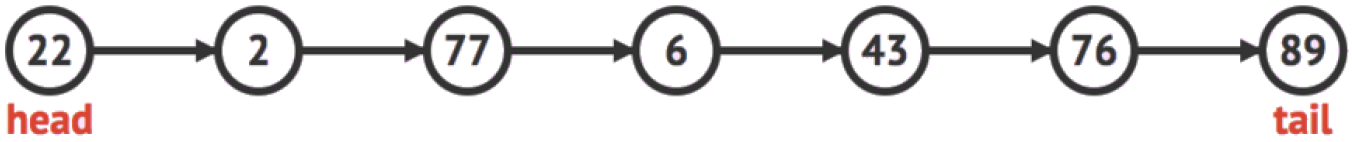

Figure 1: An example Linked List structure

```cpp
// Her bir elemani (burada sayilari, yani int) tutacak struct olusturuyoruz.
struct node
{
  int data;
  node *next;
};
node *head, *tail;

void push_back(int x) {
  // Yeni elemanimizi hafizada olusturuyoruz.
  node *t = (node*)malloc(sizeof(node));
  t -> data = x; // Elemanin verisini atiyoruz.
  t -> next = NULL; // Sona ekledigimizden sonraki elemanina NULL atiyoruz.

  // Eger veri yapimiza hic eleman eklenmediyse head
  // ve tail elemanlarini olusturuyoruz.
  if(head == NULL && tail == NULL) {
    head = t;
    tail = t;
  }
  // Eklenmisse yeni tail elemanimizi guncelliyoruz.
  else {
    tail -> next = t;
    tail = t;
  }
}

void print() {
  // Dizideki tum elemanlari geziyoruz.
  node *t = head;
  while(t != NULL) {
    printf("%d ", t -> data);
    t = t -> next;
  }
}
```

3.4 Stack

In a stack, elements are stored according to the Last In First Out (LIFO) rule. The operations we can perform on this data structure are:

- Adding an element to the top of the data structure.
- Accessing the top element of the data structure.
- Removing the top element of the data structure.
- Checking if the data structure is empty.

The usage of the stack structure in the STL library in C++ is as follows.

```cpp
int main() {
  stack < int > st;
  cout << st.empty() << endl; // Ilk bashta Stack bosh oldugu icin burada True donecektir.
  st.push(5); // Stack'in en ustune 5'i ekler. Stack'in yeni hali: {5}
  st.push(7); // Stack'in en ustune 7'yi ekler. Stack'in yeni hali: {7, 5}
  st.push(6); // Stack'in en ustune 6'yi ekler. Stack'in yeni hali : {6, 7, 5}
  st.pop(); //Stack'in en ustundeki elemani siler. Stack'in yeni hali : {7, 5}
  st.push(1); // Stack'in en ustune 1'i ekler. Stack'in yeni hali : {1, 7, 5}
  cout << st.top() << endl; // Stack'in en ustundeki elemana erisir. Ekrana 1 yazirir.
  cout << st.empty() << endl; // Burada Stack bosh olmadigindan oturu False donecektir.
}
```

3.5 Queue

In a queue, elements are stored according to the First In First Out (FIFO) rule. The operations we can perform on this data structure are:

- Adding an element to the back of the data structure.
- Accessing the front element of the data structure.
- Removing the front element of the data structure.
- Checking if the data structure is empty.

The usage of the stack structure in the STL library in C++ is as follows.

```cpp
int main() {
  queue < int > q;
  cout << q.empty() << endl; // Ilk bashta Queue bosh oldugu icin burada True donecektir.
  q.push(5); // Queue'in en ustune 5'i ekler. Queue'in yeni hali: {5}
  q.push(7); // Queue'in en ustune 7'yi ekler. Queue'in yeni hali: {7, 5}
  q.push(6); // Queue'in en ustune 6'yi ekler. Queue'in yeni hali : {6, 7, 5}
  q.pop(); //Queue'in en altindaki elemani siler. Queue'in yeni hali : {6, 7}
  q.push(1); // Queue'in en ustune 1'i ekler. Queue'in yeni hali : {1, 6, 7}
  cout << Q.front() << endl; // Queue'in en ustundeki elemana erisir. Ekrana 7 yazdirir.
}
```

3.6 Deque

The deque data structure is more comprehensive compared to stack and queue data structures. In this data structure, elements can be added both to the top and the bottom of the structure. Similarly, it is possible to access and remove elements from both the top and the bottom of the structure. The operations we can perform on this data structure are:

- Adding an element to the top of the data structure.
- Adding an element to the bottom of the data structure.

- Accessing the top element of the data structure.
- Accessing the bottom element of the data structure.
- Removing the top element of the data structure.
- Removing the bottom element of the data structure.

The usage of the stack structure in the STL library in C++ is as follows.

```
int main() {
  deque < int > q;
  q.push_front(5); // deque'nin en altina 5'i ekler.
  q.push_back(6); // deque'nin en ustune 6'yi ekler.
  int x = q.front(); // deque'nin en altindaki elemanina erisim.
  int y = q.back(); // deque'nin en ustundeki elemanina erisim.
  q.pop_front(); // deque'nin en altindaki elemanini silme.
  q.pop_back(); // deque'nin en ustundeki elemanini silme.
}
```

P.S. Since the deque data structure is more comprehensive than stack and queue data structures, it can be clearly stated that it uses twice as much memory compared to stack and queue data structures.

3.7 Priority queue

Priority queues are a type of container adaptors, specifically designed so that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array. [9, 10]

```cpp
#include <iostream>        // std::cout
#include <queue>           // std::priority_queue
using namespace std;
int main ()
{
  priority_queue<int> mypq;

  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);

  cout « "Popping out elements...";
  while (!mypq.empty())
  {
     cout « ' ' « mypq.top();
     mypq.pop();
  }
  return 0;
}
```

4 Sorting Algorithms

Sorting algorithms are used to put the elements of an array in a certain order according to the comparison operator. Numerical order or lexcographical orders are the most common ones, and there are a large number of sorting algorithms, but we discuss four of them: *Insertion Sort*, *Merge*

For a better understanding, you are strongly recommended to go into this visualization site after reading the topics: Link

4.1 Insertion Sort

Think that you are playing a card game and want to sort them before the game. Your sorting strategy is simple: you have already sorted some part and every time you pick up the next card from unsorted part, you insert it into the correct place in sorted part. After you apply this process to all cards, the whole deck would be sorted.

This is the basic idea for sorting an array. We assume that the first element of the array is the sorted part, and other elements are in the unsorted part. Now, we choose the leftmost element of the unsorted part, and put it into the sorted part. In this way the left part of the array always remains sorted after every iteration, and when no element is left in the unsorted part, the array will be sorted.

```cpp
void insertionSort(int *ar, int size){
    for (int i=1; i < size; i++)
        for (int j=i-1; 0 <= j and ar[j] > ar[j+1]; j--)
            swap(ar[j], ar[j+1]);
}
```

4.2 Merge Sort

Merge Sort is one of the fastest sorting algorithms that uses *Divide and Conquer* paradigm. The algorithm **divides** the array into two halves, solves each part **recursively** using same sorting function and **combines** them in linear time by selecting the smallest value of the arrays every time.

Procedure:

1. If the size of the array is 1, it is sorted already, stop the algorithm (base case),
2. Find the middle point of the array, and split it in two,
3. Do the algorithm for these parts separately from the first step,
4. After the two halves got sorted, merge them in linear time and the array will be sorted.

Complexity:

$$\begin{aligned} T(N) &= T(N/2) + O(N) \\ T(N) &= O(N \cdot logN) \end{aligned}$$

```
1  void mergeSort(int *ar, int size){
2      if (size <= 1)  // base case
3          return;
4
5      mergeSort(ar, size / 2);  // divide the array into two almost equal parts
6      mergeSort(ar + size / 2, size - size / 2);
7
8      int index = 0, left = 0, right = size / 2;  // merge them
9      int *temp = new int [size];
10
11     while (left < size / 2 or right < size){
12         if (right == size or (left < size / 2 and ar[left] < ar[right]))
13             temp[index++] = ar[left++];
14         else
15             temp[index++] = ar[right++];
16     }
17     for (int i=0; i < size; i++)
18         ar[i] = temp[i];
19     delete [] temp;
20 }
```

4.3 Quick Sort

Quick Sort is also a *Divide and Conquer* algorithm. The algorithm chooses an element from the array as a pivot and partitions the array around it. Partitioning is arranging the array that satisfies those: the pivot should be put to its correct place, all smaller values should be placed before the pivot, and all greater values should be placed after the pivot. The partitioning can be done in linear time, and after the partitioning, we can use the same sorting function to solve the left part of the pivot and the right part of the pivot recursively.

If the sellected pivot cannot divide the array uniformly after the partitioning, the time complexity can reach $O(n^2)$ like insertion sort. To avoid this, the pivot can generally be picked randomly.

Procedure:

1. If the size of the array is 1, it is sorted already, stop the algorithm (base case),
2. Choose a pivot randomly,
3. For all values in the array, collect smaller values in the left of the array and greater values in the right of array,
4. Move the pivot to the correct place,
5. Repeat the same algorithm for the left partition and the right partition.

Complexity:

$$\begin{aligned} T(N) &= T(N/10) + T(9 \cdot N/10) + O(N) \\ T(N) &= O(N \cdot logN) \end{aligned}$$

```
1  void quickSort(int *ar, int size){
2      if (size <= 1)  // base case
3          return;
4
5      int position = 1;  // find the correct place of pivot
6      swap(ar[0], ar[rand() % size]);
7
8      for (int i=1; i < size; i++)
9          if (ar[0] > ar[i])
10             swap(ar[i], ar[position++]);
11     swap(ar[0], ar[position-1]);
12
13     quickSort(ar, position-1);
14     quickSort(ar + position, size - position);
15 }
```

4.4 Radix Sort

Quick Sort and *Merge Sort* are comparison-based sorting algorithms and cannot run better than $O(NlogN)$. However, *Radix Sort* works in linear time ($O(N+K)$, where K is $log(max(ar))$).

Procedure:

1. For each digit from the least significant to the most, sort the array using *Counting Sort* according to corresponding digit. *Counting Sort* is used for keys between specific range, and it counts the number of elements which have different key values. After counting the number of distict key values, we can determine the position of elements in the array.

Complexity:

$$T(N) = O(N)$$

```
void radixSort(int *ar, int size, int base=10){
    int *temp = new int [size];
    int *count = new int [base]();

    //Find the maximum value.
    int maxx = ar[0];
    for(int i=1; i<size; i++){
        if(ar[i]>maxx){
            maxx=ar[i];
        }
    }

    for (int e=1; maxx/e > 0; e *= base){
        memset(count, 0, sizeof(int) * base);

        for (int i=0; i < size; i++)
            count[(ar[i]/e) % base]++;

        for (int i=1; i < base; i++)
            count[i] += count[i-1];

        for (int i=size-1; 0 <= i; i--)
            temp[--count[(ar[i]/e) % base]] = ar[i];

        for (int i=0; i < size; i++)
            ar[i] = temp[i];
    }

    delete [] temp;
    delete [] count;
}
```