ALGO-101

Week 1 - Introduction

Murat Biberoğlu

ITU ACM

October, 2022

Topics

- Big O Notation
- CPP STL
  - Vector
  - String
  - Map
  - Set
- Recursion
- Backtracking
- Searching Algorithms
  - Linear Search
  - Binary Search

Big O Notation

There can be more than one solutions or algorithms for solving problems. But which one is faster or which one is less space consuming? This question's answer is very important for competitive programmer to write better code. To figure out these we need to analyze code. There is a 3 analysis methods most generally:

- Best Case Analysis Ω
- Average Case Analysis Θ
- Worst Case Analysis $\mathcal{O}$

In competitive programming most common one is the worst case analysis. What worst case analysis does is it determines a upper bound for run time and memory space of solution.

Big O Notation – How to Analyze Code

In the next examples assume that size of *array* is n.

- Constant Time Operation $\mathcal{O}(1)$

```
1      cout << array[0] << endl;
```

- Linear Time Operation $\mathcal{O}(n)$

```
1      for (int i = 0; i < n; i++)
2          cout << array[i] << endl;
```

- Quadratic Time Operation $\mathcal{O}(n^2)$

```
1      for (int i = 0; i < n; i++)
2          for (int j = 0; j < n; j++)
3              cout << array[i] * array[j] << endl;
```

Big O Notation – How to Analyze Code

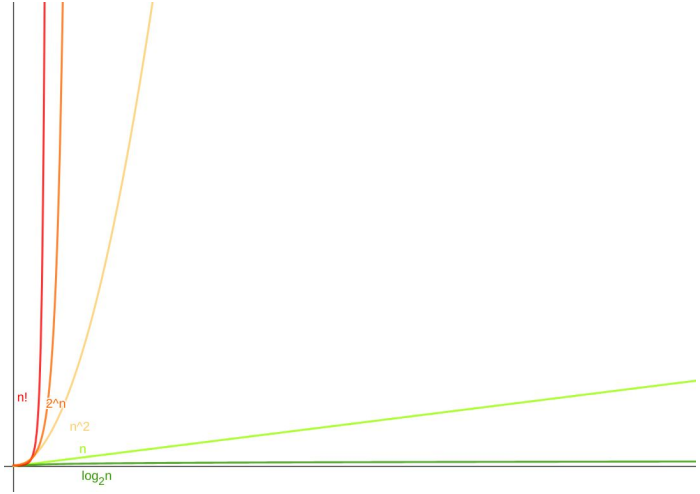
If there are nested terms or expressions we will multiply them like quadratic complexity at the previous slide. Because computer will do every operation in the inner expression for every outer expression. The below code's time complexity is $\mathcal{O}(nm)$. Asume that the size of *array_1* is m.

```
1     for (int i = 0; i < n; i++)
2         for (int j = 0; j < m; j++)
3             cout << array[i] * array_1[j] << endl;
```

If there is more than one terms that are not nested, we will only dominant terms for calculating complexities of code. Because when variables goes to the ∞ the terms other than dominant ones are getting insignificant. The below code's time complexity is $\mathcal{O}(n^2)$ not $\mathcal{O}(n^2 + n)$.

```
1     for (int i = 0; i < n; i++)
2         cout << array[i] << endl;
3     for (int i = 0; i < n; i++)
4         for (int j = 0; j < n; j++)
5             cout << array[i] * array[j] << endl;
```

Big O Notation – Comparison of Computational Complexities



CPP STL

C++ is a powerful language for competitive programming. It has a library that called STL which stands for "Standard Template Library". STL includes builtin data structures that makes job of competitive programmers easier. Some of them are:

- Vector
- String
- Map
- Set
- Stack
- Queue
- Deque
- Priority Queue

At this week we will cover most basic ones which are Vector, String, Map and Set. More advanced ones will be covered at week 4.

Vector

Vectors can be thought as arrays with additional methods and specs. Main differences are:

- vectors have dynamic memory space while arrays have static memory space
- vectors have methods that related to it's size while array have not.

We will cover the methods that listed down below of vector but there is much more than that you can go with this *link*. Assume that size of vector is n for below examples.

- **push_back** Appends an element to the end of the vector. Run time complexity is approximately $\mathcal{O}(1)$.
- **pop_back** Removes the element from end of the vector. Run time complexity is approximately $\mathcal{O}(1)$.
- **insert** Inserts an element to the given index. Run time complexity is $\mathcal{O}(n)$.
- **erase** Removes the element at the given index from vector. Run time complexity is $\mathcal{O}(n)$.
- **size** Returns the number of elements in the vector. Run time complexity is $\mathcal{O}(1)$.
- **empty** Returns true if vectors size is 0 otherwise false. Run time complexity is $\mathcal{O}(1)$.
- **begin** Returns an iterator to the front of the vector. Run time complexity is $\mathcal{O}(1)$.
- **end** Returns an iterator to the back of the vector. Run time complexity is $\mathcal{O}(1)$.

Vector – Methods

- Initialize an empty vector
- Push integers in the range [0, 4]
- Insert 100 to the index 1
- Erase element at the front of the vector
- Erase element at the back of the vector
- Get the size of the vector and print it
- Pop all the elements from back
- Print 0 if set is not empty otherwise 1

```
1   vector<int> nums;
2
3   for (int i = 0; i < 5; i++)
4       nums.push_back(i);
5
6   nums.insert(nums.begin() + 1, 100);
7
8   nums.erase(nums.begin());
9   nums.erase(nums.end());
10
11  int n = nums.size();
12  cout << n << endl;
13
14  for (int i = 0; i < n; i++)
15      nums.pop_back();
16
17  cout << nums.empty() << endl;
```

String

Strings are vectors with chars with additional methods. Unlike C strings, CPP strings have dynamic length. We will cover the methods that listed down below of string but there is much more than that you can go with this *link*.

- **append** appends given string to the end of the string. There is no run time complexity guarantees, typical implementations behave similar to vector::insert.
- **operator+=** appends given string to the end of the string. There is no run time complexity guarantees, typical implementations behave similar to vector::insert.
- **compare** compares two string and if given string is lexicographically smaller than this string returns negative number, if it is equal returns zero, else return positive number.
- **replace** replaces the given string to the given starting position and given length.
- **substr** returns the substring according to given strating position and length. Run time complexity is linear.
- **length** returns the length of string. Run time complexity is $\mathcal{O}(1)$.

String – Methods

Initialize str1 as "ITU" and str2 as "ITU - ACM Student Chapter"

```
1      string str1 = "ITU";
2      string str2 = "ITU - ACM Student Chapter";
```

Append " - " to the end of the str1 with operator +=

```
1      str1 += " - ";
```

Append " - " to the end of the str1

```
1      str1.append("ACM");
```

Compare str1 with str2. Output will be -16

```
1      cout << str1.compare(str2) << endl;
```

String – Methods

Replace str1's [6, 9) with given string and print the result

```
1     cout << str1.replace(6, 3, "Association for Computing Machinery") <<
      endl;
```

Replace str1's [0, 3) with given string and print the result

```
1     cout << str1.replace(0, 3, "Istanbul Technical University") << endl;
```

Print the lengths of str1 and str2

```
1     cout << str1.length() << ", " << str2.length() << endl;
```

Map

Maps are data structures that works with key-value pairs. In maps keys must be unique. There is a two type of map in C++ which are map and unordered map. Their implementation is different so their **insert**, **erase** and **find** functions have different run time complexities.

Function	Map	Unordered Map
insert	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
erase	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
find	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ on average, $\mathcal{O}(n)$ on worst case

Table: comparison of run time complexities of functions, size of both maps are n

Map – Methods

- **insert**
  Inserts the given key-value to the map.
- **erase**
  Removes the given key-value from the map.
- **find**
  Returns the address of key-value with given key.
- **size**
  Returns the number of key-value pairs in the map. Run time complexity is $\mathcal{O}(1)$.
- **empty**
  Returns true if vectors size is 0 otherwise false. Run time complexity is $\mathcal{O}(1)$.

Map – Methods

- Initialize an empty map
- Insert pairs {0, 'a'}, {1, 'b'}, {2, 'c'} to the
- Print the size of the map
  Print 0 if map is not empty otherwise print 1
- Try to find key 2 in map
- Erase all the elements from the map

```
1   map<int, int> m;
2
3   m.insert({0, 'a'});
4   m.insert(make_pair(1, 'b'));
5   m[2] = 'c';
6
7   cout << m.size() << endl;
8   cout << m.empty() << endl;
9
10  if (m.find(2) != m.end())
11      cout << "found\n";
12  else
13      cout << "not found\n";
14
15  for (int i = 0; i < 3; i++)
16      m.erase(i);
```

Set

Sets are data structures like in mathematics. Every element can be occur at most once in a set. There is a two type of set in C++ which are set and unordered set. Their implementation is different so their **insert**, **erase** and **find** functions have different run time complexities.

Function	Set	Unordered Set
insert	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
erase	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
find	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ on average, $\mathcal{O}(n)$ on worst case

Table: comparison of run time complexities of functions, size of both sets are n

Set – Methods

- **insert**
  Inserts an element to the set.
- **erase**
  Erases the value of given address from set.
- **find**
  Returns the address of a given value, if it is not found returns the address of end of set.
- **size**
  Returns the size of set. Run time complexity is $\mathcal{O}(1)$.
- **empty**
  Returns true if sets size is 0 otherwise false. Run time complexity is $\mathcal{O}(1)$.

Set – Methods

- Initialize an empty set
- Insert integers in the range [0, 10) to the set
- Try to find 4 in set
- Print the size of the set and print 0 if set is not empty otherwise 1
- Erase all the elements from the set

```
1   set<int> s;
2
3   for (int i = 0; i < 10; i++)
4       s.insert(i);
5
6   if (s.find(4) != s.end())
7       cout << "found\n";
8   else
9       cout << "not found\n";
10
11  cout << s.size() << endl;
12  cout << s.empty() << endl;
13
14  for (int i = 0; i < 10; i++)
15      s.erase(i);
```

Recursion

Recursion is a way that finding values of a function which repeats itself by definition. We call that functions as Recursive Functions. Recursive functions can be used at problems that can be divisible to the subproblems.
Fibonacci series, factorial series or triangular series can be given as an example to recursive functions.
Recursive fibonacci function:

```
1 int fibonacci(int n) {
2     if (n == 0)
3         return 0;
4     if (n == 1 || n == 2)
5         return 1;
6     return fibonacci(n - 1) + fibonacci(n - 2);
7 }
```

Recursion

Recursive factorial function:

```
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    return n * fibonacci(n - 1);
}
```

Recursive triangular function:

```
int triangular(int n) {
    if (n == 0)
        return 0;
    return n + triangular(n - 1);
}
```

Calculating Complexity of Recursive Functions – Master Theorem

In the recursive functions and algorithms we can use Master Theorem to calculate run time complexity of it.

$$T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$$

n = size of input
a = number of subproblems in the recursion
$\frac{n}{b}$ = size of each subproblem. All subproblems are assumed to have the same size.
$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions.
If $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, then the time complexity of a recursive relation is given by:

$$\begin{cases} \Theta(n^{\log_b a}) & f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right) \\ \Theta(n^{\log_b a} \times logn) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n) = \Omega\left(n^{\log_b a + \epsilon}\right) \end{cases}$$

Backtracking

According to geeksforgeeks,
Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree). Backtracking can also be said as an improvement to the brute force approach.

Backtracking – Permutation Example

Printing all the permutations with backtracking:

```
1  void permute(int index, vector<int> &nums) {
2      if (index == nums.size()) {
3          print(nums);
4          return;
5      }
6      for (int i = index; i < nums.size(); i++) {
7          swap(nums[index], nums[i]);
8          permute(index + 1, nums);
9          swap(nums[index], nums[i]);
10     }
11 }
```

and function call is

```
1      vector<int> nums = {1, 2, 3};
2      permute(0, nums);
```

Backtracking – Subset Example

Printing all the subsets with backtracking:

```
1  void subset(int index, vector<int> &nums, vector<int> &set) {
2      if (index == nums.size()) {
3          print(set);
4          return;
5      }
6
7      set.push_back(nums[index]);
8      subset(index + 1, nums, set);
9
10     set.pop_back();
11     subset(index + 1, nums, set);
12 }
```

and function call is

```
1      vector<int> nums = {1, 2, 3};
2      vector<int> set;
3      subset(0, nums, set);
```

Linear Search

Search a specific value at an array with starting from zeroth index and ending at $n-1$th index. Run time complexity is $\mathcal{O}(n)$. Example code:

```
1 int linearSearch(vector<int> &nums, int value) {
2     int n = nums.size();
3
4     for (int i = 0; i < n; i++)
5         if (nums[i] == value)
6             return i;
7     return -1;
8 }
```

Binary Search

If given array is sorted we can do better than linear search with binary search in terms of run time complexity with reducing it to $\mathcal{O}(\log n)$. Here is the algorithm:

1. Get middle index if left $\leq$ right else return -1
2. Compare middle value with searching value
   - If they are equal return middle
   - If middle value is less than searching value, search [middle + 1, right]
   - If middle value is greater than searching value, search [left, middle -1]

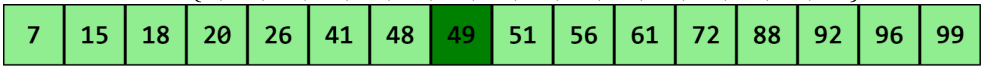
   Go back to step 1.

$$\textbf{left} = \text{lower boundary of the range}$$
$$\textbf{right} = \text{upper boundary of the range}$$

$$\textbf{middle} = \begin{cases} \frac{right-left}{2} & right - left \equiv 0 \mod 2 \\ \frac{right-left-1}{2} & right - left \equiv 1 \mod 2 \end{cases}$$

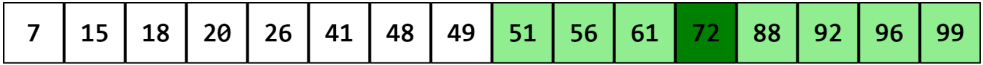
Binary Search – Example

Search 61 in the $\{7, 15, 18, 20, 26, 41, 48, 49, 51, 56, 61, 72, 88, 92, 96, 99\}$.



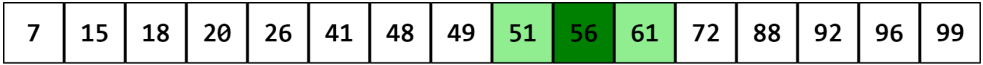

```
left   = 0
right  = 15
middle = 7
```

49 is less than 61, search the [middle + 1, right]



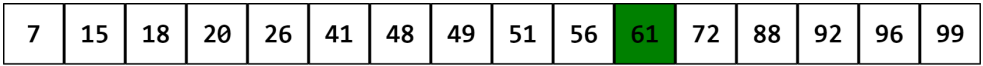

```
left   = 8
right  = 15
middle = 11
```

72 is greater than 61, search the [left, middle - 1]




```
left   = 8
right  = 10
middle = 9
```

56 is less than 61, search the [middle + 1, right]




```
left   = 10
right  = 10
middle = 10
```

61 is equal to 61, algorithm finished

Binary Search – Recursive Implementation

```
1  int binarySearch(vector<int> &nums, int left, int right, int value) {
2      if (right < left)
3          return -1;
4
5      int middle = (right - left) / 2;
6      if (nums[middle] = value)
7          return middle;
8      else if (nums[middle] < value)
9          return binarySearch(nums, middle + 1, right, value);
10     else
11         return binarySearch(nums, left, middle - 1, value);
12 }
```

Binary Search – Iterative Implementation

```
1  int binarySearch(vector<int> &nums, int value) {
2      int left = 0;
3      int right = nums.size() - 1;
4      while (left <= right) {
5          int middle = (right - left) / 2;
6          if (nums[middle] == value)
7              return middle;
8          else if (nums[middle] < value)
9              left = middle + 1;
10         else
11             right = middle - 1;
12     }
13     return -1;
14 }
```

How to Decide to Use Which Searching Algorithm

Assume that we have given an array with size n and q queries.

- If given array is sorted then use binary search for every query. Run time complexity is $\mathcal{O}(q \times \log n)$.
- If given array is unsorted
  - Sort array with $\mathcal{O}(n)$ sorting algorithm and use binary search for every query, then run time complexity will be $\mathcal{O}(n + q \times \log n)$.
  - Sort array with $\mathcal{O}(n \times \log n)$ sorting algorithm and use binary search for every query then total run time complexity will be $\mathcal{O}(n \times \log n + q \times \log n)$.
  - Don't sort array and use linear search for every query then total run time complexity will be $\mathcal{O}(q \times n)$.

Decide to use a way according to n, q and remaining memory space value.

Related Questions

Question	Topic	Difficulty
Number of Good Pairs	Array, Vector, Hash Table	Easy
Matrix Block Sum	Array, Vector	Medium
Jewels and Stones	String	Easy
Count Vowel Substrings of a String	String, Set	Easy
Number of Laser Beams in a Bank	String, Array, Vector	Medium
Generate Parentheses	Backtracking	Medium
Combination Sum	Backtracking	Medium
Count Number of Maximum Bitwise-OR Subsets	Backtracking	Medium
Search Insert Position	Binary Search	Easy
Sqrt(x)	Binary Search	Easy
Peak Index in a Mountain Array	Binary Search	Medium
Find Minimum in Rotated Sorted Array	Binary Search	Medium

References

- https://en.cppreference.com/w/cpp/container/vector
- https://en.cppreference.com/w/cpp/string/basic_string
- https://en.cppreference.com/w/cpp/container/map
- https://en.cppreference.com/w/cpp/container/unordered_map
- https://en.cppreference.com/w/cpp/container/set
- https://en.cppreference.com/w/cpp/container/unordered_set
- https://www.programiz.com/dsa/master-theorem
- https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/
- https://leetcode.com/problemset/all/