Lecture 2: Analysis of Algorithms

Date: 08/24/2023

Analysis of algorithms is the process of finding the computational complexity of algorithms—the amount of **time**, **storage**, or **other resources** needed to execute them.

Generally the algorithm is measured in two ways,

- Time Complexity
- Space Complexity

We generally care about worst / best / average cases,

- **Worst Case**: the maximum time/space the algorithm can take for some specific input cases
- **Best Case:** the least time/space the algorithm can take for some specific input cases
- **Average Case:** on an average, how much time/space the algorithm can take

⇒ **Assumptions**:

- Random Access Machine (RAM) Model: instructions execute one after another, with no concurrent operations.
- Each instruction takes the same amount of time
- There is no caching involved

Time Complexity

It is intuitive that the time taken by a program is proportional to the total number of instructions in that program, as the CPU takes one cycle to run one instruction.
We can count how many instructions the program will run and predict how much time the program will take.

Example 1: Count Zeros

1_countZeros.cpp

```
void solve(int n, vector<int> arr){
   int answer = 0;

   for(int i=0; i<n; i++){
       if(arr[i] == 0)
       answer += 1;
   }

   cout<<answer;
}
```

We can expand the for loop and make it easier for us to count total instructions

```
void solve(int n, vector<int> arr){
   int answer = 0;

   if(arr[0] == 0)
   answer ++;

   if(arr[1] == 0)
   answer ++;

   if(arr[2] == 0)
   answer ++;

   .
   .
   .

   cout<<answer;
}
```

The for loop will run the body n times. So we can note instruction run count for every instruction

```
void solve(int n, vector<int> arr){
   int answer = 0;                 // 1

   for(int i=0; i<n; i++){
       if(arr[i] == 0)             // n
       answer += 1;                // x
   }

   cout<<answer;                   // 1
}
```

The run count of instruction “answer +=1” depends on the input. For now let's say it's x.

⇒ **Total Instructions**: n + x + 2

Example 2: Two Sum

Count how many pairs of elements in the array add up to zero.

2_twoSum.cpp

```
void solve(int n, vector<int> arr){
   int answer = 0;                     // 1

   for(int i1=0; i1<n; i1++){          // 1
       for(int i2=i1+1; i2<n; i2++){   // n
           if(arr[i1] + arr[i2] == 0)  // n*(n-1) / 2
           answer += 1;                // x
       }
   }

   cout<<answer;                       // 1
}
```

The body of the first for loop iterates for “n” times.
The body of the second for loop iterates for “iteration number” times.
We can now calculate total number of times the if statement runs,
1 + 2 + 3 + 4 … n = n * (n+1) / 2
([Proof of Sum of Natural Numbers])

⇒ **Total Instructions**: (n * (n+1) / 2) + x + 2

Example 3: Three Sum

Count how many triplets in the array add up to zero.

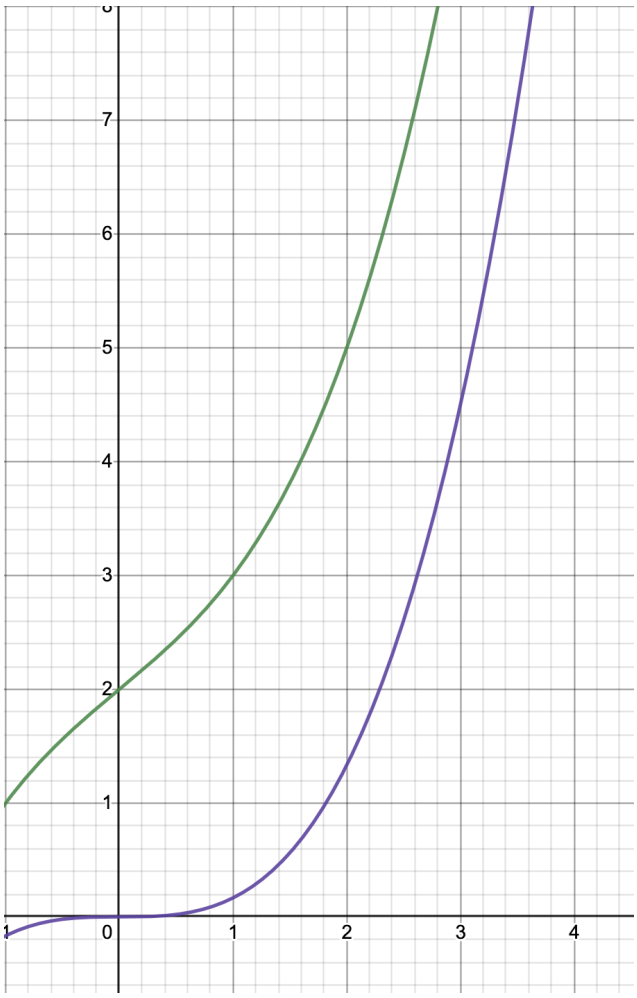
3_threeSum.cpp

```
void solve(int n, vector<int> arr){
   int answer = 0;                                    // 1

   for(int i1=0; i1<n; i1++){                         // 1
       for(int i2=i1+1; i2<n; i2++){                  // n
           for(int i3=i2+1; i3<n; i3++){              // n*(n-1)/2
               if(arr[i1] + arr[i2] + arr[i3] == 0)   // n*(n-1)*(n-2)/6
               answer += 1;                           // x
           }
       }
   }

   cout<<answer;                                      // 1
}
```

⇒ **Total Instructions**: (n * (n-1) * (n-2) / 6) + x + 2

Tilde Approximation

As you can see, the expression for total instructions has started to get messier.
If we notice we don't need every single instruction count. As input gets larger, only higher-order terms will matter.



Purple: (n * (n-1) * (n-2) / 6)
Green: n^3 / 6

⇒ **Definition**: We write ~g(N) to represent any function that, when divided by g(N), approaches 1 as N grows, and we write f(N) ~ g(N) to indicate that f(N)/g(N) approaches 1 as N grows.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

Total Instruction in terms of Tilde approximation,

- Count Zeros:

f(n) = n + x + 2,
g(n) = n (higher order term in f(n))
f(n) / g(n) = (n + x + 2) / n = 1 + x/n + 2/n
As n grows, x/n and 2/n will approach zero.
⇒ **So, f(n) is ~n**

- Two Sum:
  f(n) = (n * (n+1) / 2) + x + 2
  f(n) = $n^2/2$ + n/2 + x + 2
  If we take **g(n) = $n^2/2$** (higher order term in f(n))
  f(n) / g(n) = 1 + 2/n + $2x/n^2$ + $4/n^2$
  As n grows, every term which has n in denominator, will approach zero.
  ⇒ **So, f(n) is ~$n^2/2$**

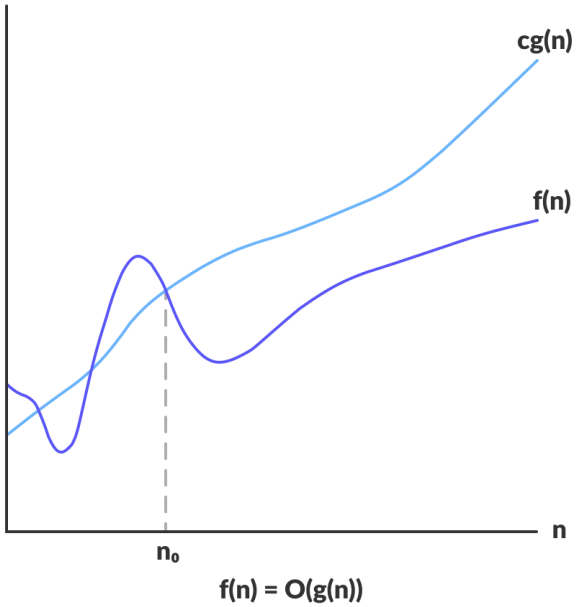
- Three Sum: (n * (n-1) * (n-2) / 6) + x + 2 => **~n^2 / 6**

Big O Notation

O-notation characterizes an **Upper Bound** on the asymptotic behavior of function.

⇒**Definition**: For a given function g(n), we denote by O(g(n)) (pronounced “big-oh of g of n” or sometimes just “oh of g of n”) the set of functions
O(g(n)) = f(n): there exist positive constants c and n_0 such that
$0 \le f(n) \le c*g(n)$ for all $n \ge n_0$

⇒ In layman’s term, for enough big inputs, f(n) will be always **less** than or equal to c*g(n).



Example,

- $10n^2 + 8n + 2 = O(n^2)$
- $n^4 + n^2 + 10 = O(n^4)$

Note that, O-notation gives an upper bound, so we can write any function that is greater than or equal to f(n).
Following is still correct,

- $10n^2 + 8n + 2 = O(n^4)$
- $n^4 + n^2 + 10 = O(2^n)$

But, we often want to have tight bounds, which means, find g(n) that is as close to f(n).

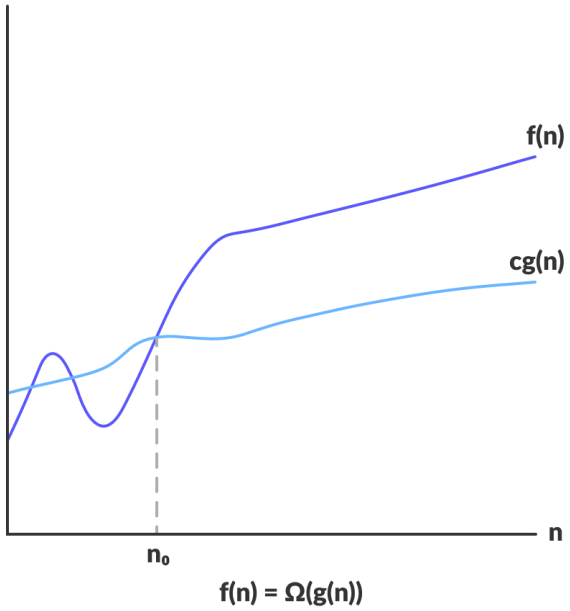
Big Omega Notation:

Ω-notation characterizes a **Lower Bound** on the asymptotic behavior of function.

⇒ **Definition**: For a given function g(n), we denote by O(g(n)) (pronounced “big-oh of g of n” or sometimes just “oh of g of n”) the set of functions
Ω(g(n)) = f(n): there exist positive constants c and n_0 such that

$$0 \leq c*g(n) \leq f(n) \text{ for all } n \geq n_0$$

⇒ In layman's term, for enough big inputs, f(n) will be always **greater** than or equal to c*g(n).



Example,

- $10n^2 + 8n + 2 = \Omega(n^2)$
- $n^4 + n^2 + 10 = \Omega(n^4)$

Note that, Ω-notation gives a lower bound, so we can write any function that is less than or equal to f(n).
Following is still correct,

- $10n^2 + 8n + 2 = \Omega(1)$
- $n^4 + n^2 + 10 = \Omega(n^2)$

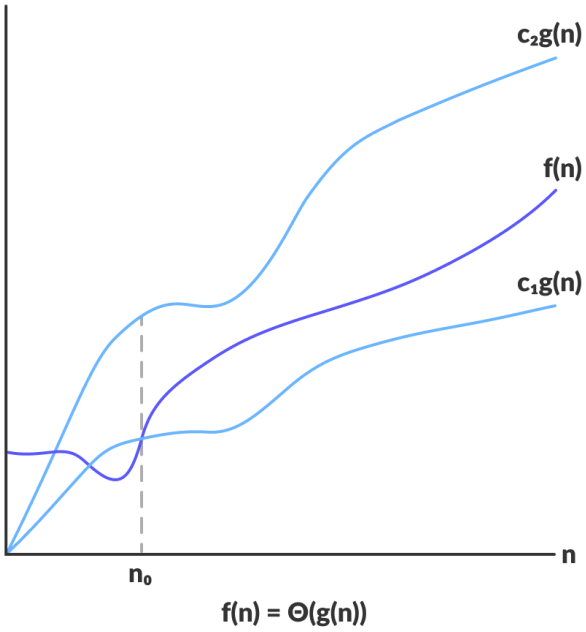
But, we often want to have tight bounds, which means, find g(n) that is as close to f(n).

Theta Notation

Θ-notation is used for asymptotically **tight** bounds.

Definition: For a given function g(n), we denote by θ(g(n)) ("theta of g of n") the set of functions,
θ(g(n)) = f(n) : there exist positive constants c1, c2 and n_0 such that
$0 \le c1*g(n) \le f(n) \le c2*g(n)$ for all $n \ge n_0$

⇒ In layman's term, for enough big inputs, f(n) will always be in range of c1*g(n) and c2*g(n)



f(n) = Θ(g(n))

Example,

- $10n^2 + 8n + 2 = \theta(n^2)$
- $n^4 + n^2 + 10 = \theta(n^4)$

Note that, θ-notation gives tight bounds, so we can NOT write any functions. We can only write a function that is in the given range. Following is NOT correct,

- $10n^2 + 8n + 2 = \theta(n^3)$
- $n^4 + n^2 + 10 = \theta(n^2)$

⇒ ~ notation is similar to θ-notation

Example:

4_findMax.cpp

```cpp
bool solve(int n, vector<int> arr, int target){
   for(int i=0; i<n; i++){
       if(arr[i] == target)
       return true;
   }

   return false;
}
```

The running time of this algorithm is not as simple as others as it involves early stopping. Whenever the algorithm finds the target element, it will stop the execution.

Worst Case Analysis:

The worst case will be when the target element does not exist in the array. The algorithm will go through every element in the array.
In this case, the if statement will be executed n times.

Let's write algorithm's worst-case time complexity in all asymptotic notation,

- **Big O Notation**: we can always write a really big function. e.g. $O(2^n)$ which says that, our worst-case time will be less than 2^n
  But, we just proved that in the worst-case the algorithm will go through all elements. So, tighter bound we can write is **O(n)**
- **Big Omega Notation**: we can always write a really small function. e.g. Ω(1) which says that our worst case time will be greater than constant.
  But, as per our worst case analysis we can write, **Ω(n)**
- **Theta Notation:** our BigO and Omega notations are same, so we can write the worst case time of our program is **θ(n)**

Most of the time, we want to write our time complexities in terms of Theta Notation.

Best Case Analysis:

The best case will be when the target element appears at the start of the array. The algorithm will only check the first element and stop the execution.

In this case, the if statement will be executed 1 time.

Let's write algorithm's best-case time complexity in all asymptotic notations,

- **Big O Notation**: we can always write a really big function. e.g. $O(2^n)$ which says that, our worst-case time will be less than 2^n But, we just proved that in the best-case the algorithm will only run for one instruction. So, tighter bound we can write is **O(1)**
- **Big Omega Notation**: we can write our best-case time will be Ω(1) which is true and we can not write any bigger function that still satisfies Big Omega notations. So, tightest bound we can write is **Ω(1)**
- **Theta Notation:** our BigO and Omega notations are same, so we can write the worst case time of our program is **θ(1)**

Asymptotic Notation Properties

Informally we can say,

- f(n) = O(g(n)) is like a ≤ b,
- f(n) = Ω(g(n)) is like a ≥ b,
- f(n) = θ(g(n)) is like a = b

Transitivity

f(n) = O(g(n)) and g(n) = O(h(n))
Then, we can say
f(n) = O(h(n))
⇒ it's similar to, if a < b and b < c then a < c.

Reflexivity

We can write asymptotic notation for function itself.
f(n) = θ(f(n)) means, f(n) = f(n)
f(n) = O(f(n)) means, f(n) <= f(n)
f(n) = Ω(f(n)) means, f(n) >= f(n)
⇒ it's similar to, a = a, a<=a, a>=a

Symmetry

When we write f(n) in terms of asymptotic notation of g(n), then transpose of that will also be true.

- f(n) = θ(g(n)) means, g(n) = θ(f(n))
- f(n) = O(g(n)) means, g(n) = Ω(f(n))

⇒ it's similar to, if a <= b then it means b >= a

Addition

O(f(n)) + O(g(n)) = O(max(f(n), g(n)))

⇒ **Proof**: Without loss of generality, assume
f(n) ≤ g(n) ⇒ O(f(n)) + O(g(n)) = c1*f(n) + c2*g(n)
From what we assumed, we can write,
O(f(n)) + O(g(n)) ≤ c1*g(n) + c2*g(n) ≤ (c1 + c2) g(n) ≤ c*g(n) ≤ c*max(f(n), g(n))

Space Complexity

Space complexity is measured by the amount of memory space required by the program. This includes memory used by inputs, outputs, and any other **auxiliary** space.

We denote space complexity by using asymptotic notation, Big O, Big Omega, Theta Notations.

Examples:

HashTable Search

5_hashTable.cpp

```
bool solve(map<int, bool> hashTable, int target){
   return hashTable[target];
}
```

- Input Space: O(n), Ω(n), θ(n)

- Output Space: O(1), Ω(1), θ(1)
- Auxiliary Space: O(1), Ω(1), θ(1)

In most of the cases, we want to optimize Auxiliary space.

Find Factorial

6_findFactorial.cpp

```cpp
bool solve(int n){
   if(n == 1) return 1;

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

- Input Space: O(1), Ω(1), θ(1)
- Output Space: O(1), Ω(1), θ(1)
- Auxiliary Space: ?

It seems like the Auxiliary Space should be θ(1) as we are not creating any variables.
But, the CPU requires space to implement recursion. (function stack)
It has to push the function and its arguments to the stack.

So, to count space complexity for recursive functions, we need to count how many times the function gets called.

Not every algorithm will have a worst case / best case.
In this specific algorithm, there are no best/worst cases as it just runs for n recursions.

Worst Case Analysis

Regardless of the input, the function will call solve n times
Auxiliary Space: O(n), Ω(n), θ(n)

Best Case Analysis

Regardless of the input, the function will call solve n times
Auxiliary Space: O(n), Ω(n), θ(n)

Caveats of Assumption

Large Constants

With leading-term approximations, we ignore constant coefficients in lower-order terms, which may not be justified. For example, when we approximate the function $2N^2 + cN$ by $\sim 2N^2$, we are assuming that c is small. If that is not the case (suppose that c is 10^3 or 10^6) the approximation is misleading. Thus, we have to be sensitive to the possibility of large constants.

Instruction Time

The assumption that each instruction always takes the same amount of time is not always correct. For example, most modern computer systems use a technique known as caching to organize memory, in which case accessing elements in huge arrays can take much longer if they are not close together in the array.

System considerations

Typically, there are many, many things going on in your computer. Java is one application of many competing for resources, and Java itself has many options and controls that significantly affect performance. A garbage collector or a just-in-time compiler or a download from the internet might drastically affect the results of experiments.

Readings from Book

Book	**Chapter / Sections**

Algorithm Design In Three Acts	Chapter 3
Algorithms_4e	Section 1.4
Grokking Algorithms	Section 1.3
Introduction to Algorithms 4e	Section 1.2, Chapter 3