Data Structures and Algorithms

Complexity - Big O Notation

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

September 17, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the terminology of complexity
- be able to prove that an algorithm belongs to a particular class
- understand the meaning of Big O notation
- be able to use Big O notation to compare algorithms

Table of Contents

1. Application Programmer Interface

2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation

3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Application Programmer Interface (API)

Java provides a lot of functionality for us to use when we are programming

- This functionality is documented in the application programmer interface
- `http://docs.oracle.com/javase/8/docs/api/`

Application Programmer Interface

Excerpt from the API

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in amortized constant time, that is, adding `n` elements requires O(n) time. All of the other operations run in linear time (roughly speaking).

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Asymptotic Complexity

- In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input
- We can use this to mathematically prove what class an algorithm belongs to
- More precisely, let f and g be functions of a real variable. We say that f and g are in the same complexity class if the limit $\lim_{x \to \infty} \frac{f(x)}{g(x)}$ exists and is both greater than 0 and less than infinity
  - We sometimes say that f is asymptotically dominated by g
  - Typically we will use basic functions to compare other algorithms to e.g. if we want to see if an algorithm with running time $f(x)$ belongs to the linear time class of algorithms we will compare it with $g(x) = x$

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Asymptotic Complexity

Lets look at the mathematical functions we have

- $a(n) = n$
- $b(n) = n + 50$
- $c(n) = 100 * n$
- $d(n) = 5000 + 1000 * n$

Here we want to prove that the algorithm with running time a belongs to the linear time class of algorithms

$$\lim_{n \to \infty} \frac{a(n)}{n}$$

$$= \lim_{n \to \infty} \left(\frac{n}{n} \right)$$

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

$$= 1$$

Asymptotic Complexity

Lets look at the mathematical functions we have

- $a(n) = n$
- $b(n) = n + 50$
- $c(n) = 100 * n$
- $d(n) = 5000 + 1000 * n$

Here we want to prove that the algorithm with running time b belongs to the linear time class of algorithms

$$= \lim_{n\to\infty} \left(\frac{n+50}{n}\right)$$

$$= \lim_{n\to\infty} \left(\frac{n}{n}\right) + \lim_{n\to\infty} \left(\frac{50}{n}\right)$$

$$= \lim_{n\to\infty} (1) + \lim_{n\to\infty} \left(\frac{50}{n}\right)$$

$$= \lim_{n\to\infty} 1 + \left(\frac{50}{\infty}\right)$$

Asymptotic Complexity

Let us look at the mathematical functions we have

- $a(n) = n$
- $b(n) = n + 50$
- $c(n) = 100 * n$
- $d(n) = 5000 + 1000 * n$

Here we want to prove that the algorithm with running time d belongs to the linear time class of algorithms

$$= \lim_{n \to \infty} \left(\frac{1000 * n + 5000}{n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{1000 * n}{n} \right) + \lim_{n \to \infty} \left(\frac{5000}{n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{1000}{1} \right) + \lim_{n \to \infty} \left(\frac{5000}{n} \right)$$

$$= \lim_{n \to \infty} 1000 + \left(\frac{5000}{\infty} \right)$$

Asymptotic Complexity

Big O notation

Asymptotic analysis, shows us that all of these functions are asymptotically dominated by n

- This means that it is the component of the function that determines the complexity

Linear time

- All algorithms that are asymptotically dominated by n belong to the class linear time
- We call this class of algorithms O(N) in Big O notation
- This is read as order n

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Asymptotic Complexity

Let us look at some more mathematical functions representing running time

- `a(n) =` $50 * n^2$
- `b(n) =` $10 * n^2 + 500 * n + 40$

- From last lecture we know that these methods belong to the quadratic time class of algorithms (Because the largest exponent of n is 2)
- But how do we prove this?
- The method is the same as before except here we compare the running time of the algorithm against $g(x) = x^2$

Asymptotic Complexity

Proving O(N^2)

Let us look at some more mathematical functions representing running time

- `a(n) =` $50 * n^2$
- `b(n) =` $10 * n^2 + 500 * n + 40$

We will prove that *a* belongs to the quadratic time class of algorithms

$$= \lim_{n \to \infty} \left(\frac{50 * n^2}{n^2} \right)$$

$$= \lim_{n \to \infty} \left(\frac{50 * 1}{1} \right)$$

$$= \lim_{n \to \infty} (50)$$

$$= 50$$

Asymptotic Complexity

Proving O(N^2)

Let us look at some more mathematical functions representing running time

- `a(n) =` $50 * n^2$
- `b(n) =` $10 * n^2 + 500 * n + 40$

We will prove that b does not belong to the linear time class of algorithms

$$= \lim_{n \to \infty} \left(\frac{10 * n^2 + 500 * n + 40}{n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{10 * n^2}{n} \right) + \lim_{n \to \infty} \left(\frac{500 * n}{n} \right) + \lim_{n \to \infty} \left(\frac{40}{n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{10 * n}{1} \right) + \lim_{n \to \infty} \left(\frac{500 * 1}{1} \right) + \lim_{n \to \infty} \left(\frac{40}{n} \right)$$

$$= \infty + 500 + 0 = \infty$$

Big O Notation

$O(N^2)$

- These algorithms all belong to the quadratic time class of algorithms
- This class is also called $O(N^2)$
- This is read as order n squared

Order n^x

- There are also classes of algorithms $O(N^3)$, $O(N^4)$ etc
- These are referred to as polynomial time classes

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Big O Notation

O(log n)

- In the last lecture we looked at the binary search algorithm
- We calculated that the running time as the function $t(n) = 120 * \lfloor \log_2 n \rfloor + 230$
- We will prove that this function is asymptotically dominated by log n
- We can ignore the floor and base in the calculations

$$= \lim_{n \to \infty} \left(\frac{120 * \log n + 230}{\log n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{120 * \log n}{\log n} \right) + \lim_{n \to \infty} \left(\frac{230}{\log n} \right)$$

$$= \lim_{n \to \infty} \left(\frac{120}{1} \right) + \lim_{n \to \infty} \left(\frac{230}{\log n} \right)$$

$$= 120 + 0 = 120$$

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Comparing Running Times

Big O notation is very useful for comparing algorithms that belong to different classes

- If we want to compare linear search and binary search
  - Linear search is O(n)
  - Binary search is O($\log n$)
- A basic understanding of maths will tell us that binary search is more efficient
- What about comparing insertion sort and selection sort?
  - Insertion sort is $O(n^2)$
  - Selection sort is $O(n^2)$
  - We cannot use big O notation to compare these

Table of Contents

1. Application Programmer Interface

2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation

3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Big O notation

There are a large number of commonly used classes of algorithms that we can represent

Name	Notation	Rough running time for n = 50
Constant time	$O(1)$	1
Logarithmic time	$O(\log n)$	5
Linear time	$O(n)$	50
Log linear time	$O(n * \log n)$	250
Quadratic time	$O(n^2)$	2500
Cubic time	$O(n^3)$	125000
Exponential time	$O(2^n)$	1125899906842624
Factorial time	$O(n!)$	3041409320171337804361 260816606500000000000 0000000000000000000000

Big O ordering

- AS the previous slide shows there can be a very large difference between the running times of algorithms for different classes
- Using the information we have just seen (and can calculate for any new classes we discover) we can order the classes from most to least efficient
  - $O(1)$
  - $O(\log n)$
  - $O(n)$
  - $O(n * \log n)$
  - $O(n^2)$
  - $O(n^3)$
  - $O(2^n)$
  - $O(n!)$

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Laws of Big O Notation

There are some laws that allow us to easily calculate the combination of different Big O values

Addition: absorption

1. $O(1) + O(1) + ... + O(1) = k * O(1) = O(1)$
2. $O(n) + O(n) + ... + O(n) = k * O(n) = O(n)$
3. $O(n) + O(m) = max(O(n), O(m))$

Multiplication

1. $O(n) * O(n) = O(n^2)$
2. $n * O(n) = O(n^2)$
3. $O(n) * O(m) = O(n * m)$
4. $O(k * f(n)) = k * O(f(n)) = O(f(n))$
5. $O(n^a) * O(n^b) = O(n^{a+b})$

Laws of Big O Notation

Example

Code	Line cost
`long pow1(int a, int b){`	O(1)
`    long z = 1;`	O(1)
`    int k = 0;`	O(1)
`    while(k < b){`	O(n)
`        z = z * a;`	O(n)
`        k = k + 1;`	O(n)
`    }`	
`    return z;`	O(1)
`}`	

Laws of Big O Notation

Example

We end up with the final result:

$$O(1) + O(1) + O(1) + O(n) + O(n) + O(n) + O(1)$$

$$4 * O(1) + 3 * O(n)$$

Addition rule 1 and 2

$$O(1) + O(n)$$

Addition rule 3

$$O(n)$$

Example 2

Consider the following code

```
int c = 1;
while(n > c){
  c = c * 2;
}
```

In terms of n, how many times will the loop execute?

- Every iteration the progress c makes towards n is doubled
- This is a logarithmic algorithm

Example 3

Code	Line cost
`long pow2(int a, int b){`	$O(1)$
`int c = 1; int s = b;`	$O(1)$
`long z = 1;`	$O(1)$
`while(b >= c){`	$O(\log n)$
`c = 2 * c;`	$O(\log n)$
`}`	
`while(c != 1){`	$O(\log n)$
`c = c / 2;`	$O(\log n)$
`z = z * z;`	$O(\log n)$
`if(s >= c){`	$O(\log n)$
`s = s - c;`	$O(\log n)$
`z = z * a;`	$O(\log n)$
`}`	
`}`	
`return z;`	$O(1)$
`}`	

Example 3

We end up with the final result:

$$4 * O(1) + 8 * O(\log n)$$

Addition rule 1 and 2

$$O(1) + O(\log n)$$

Addition rule 3

$$O(1) + O(\log n)$$

$$O(\log n)$$

Example 4

Code	Line cost
`int sum = 0, j, i;`	$O(1)$
` for(j = 1; j <= n; j = j * 2){`	$O(\log n)$
`   for(i = 0; i < n; i++){`	$O(n) * O(\log n)$
`      sum = sum + 1;`	$O(n) * O(\log n)$
`   }`	
` }`	

- The outer loop executes $\log n$ times
- The inner loop executes n times
- Therefore the code inside the inner loop is executed $n * \log n$ times

$$O(\log n) + 2 * O(n) * O(\log n) + O(1)$$

$$O(\log n) + O(n) * O(\log n) + O(1)$$

$$O(\log n) + O(n * \log n) + O(1)$$

$$O(n * \log n)$$

Example 5

```
int f[N][N];
int g[N][N];
int add[N][N];
int i, j;
for(i = 0; i < N;i++){
  for(j = 0; j < N; j++){
    f[i][j] = rand();
    g[i][j] = rand();
  }
}
// add elements on a row by row basis
for(i = 0; i < N; i++){
  for(j = 0;j < N; j++){
    add[i][j] = f[i][j] + g[i][j];
  }
}
```

Example 5

Code	Line cost
`int f[N][N];`	$O(1)$
`int g[N][N];`	$O(1)$
`int add[N][N];`	$O(1)$
`int i, j;`	$O(1)$
`for(i = 0; i < N;i++){`	$O(n)$
`  for(j = 0; j < N; j++){`	$O(n) * O(n)$
`    f[i][j] = rand();`	$O(n) * O(n)$
`    g[i][j] = rand();`	$O(n) * O(n)$
`  }`	
`}`	
`for(i = 0; i < N; i++){`	$O(n)$
`  for(j = 0;j < N; j++){`	$O(n) * O(n)$
`    add[i][j] = f[i][j] + g[i][j];`	$O(n) * O(n)$
`  }`	
`}`	

Example 5

$$5 * O(1) + 2 * O(n) + 3 * O(n) * O(n)$$

Addition rule 1/2

$$O(1) + O(n) + O(n) * O(n)$$

Multiplication rule 2

$$O(1) + O(n^2) + O(n)$$

Addition rule 3

$$O(n^2)$$

Table of Contents

1. Application Programmer Interface
2. Asymptotic Complexity
   - O(N)
   - $O(N^2)$
   - O(log n)
   - Comparing Algorithms Using Big O Notation
3. Big O notation
   - Laws of Big O Notation
   - Why use Big O Notation?

Why use Big O Notation?

- Big O notation provides a way to classify algorithms that can be used for comparison purposes
- A program that is $O(1)$ will perform better than a program that is $O(n)$, usually
- It does not provide comparison for algorithms in the same class.
- We can also extend this idea to analysing best case, average case and worst case scenarios for given algorithm

Linear Search

```c
int search(int f[],int n, int x){
   int found = 0;
   int j = 0;
   while(j < n && ! found){
      if(f[j] == x) {
         found = 1;
      } else {
         j++;
      }
   }
   return found;
}
```

Complexity of Linear Search

We will look at the complexity of the linear search algorithm

- What is the worst possible performance for this linear search algorithm?
  - $O(n)$
- When does the worst case happen?
  - When the element is in the last position or not in the array
- What is the best possible performance?
  - $O(1)$
- When does this happen?
  - When the element is in the first position.
- What is the average performance of the linear search algorithm?

Complexity of Linear Search

Average case

- First we assume that the probabilities are equal that the element may be in any position
- Then the running time is the sum of all the possible outcomes divided by the number of outcomes
  - The sum of all possible outcomes is

$$\sum_{i=0}^{n} i$$

  - We know this evaluates to

$$\frac{n * (n+1)}{2}$$

- Therefore the probability is

$$\frac{n * (n+1)}{2 * n} = \frac{n+1}{2} = O(n)$$

Binary Search

```
int bs(int f[], int n, int x) {
  int l = 0, u = n, m;
  while (l < u) {
    m = (u + l) / 2;
    if (f[m] < x)
      l = m + 1;
    else {
      u = m;
    }
  }
  return f[m] == x;
}
```

Complexity of Binary Search

We will look at the complexity of the binary search algorithm

- What is the worst possible performance for this binary search algorithm?
  - $O(\log n)$
- What is the best possible performance?
  - $O(\log n)$
- What is the average performance of the linear search algorithm?
  - $O(\log n)$

Further Information and Review

If you wish to review the materials covered in this lecture or get further information, read the following sections in Data Structures and Algorithms textbook.

- 4.1 - Algorithm Classes
- 4.2 - Analysis of Algorithms