Developing Efficient Algorithms

Archie Powell

November 30th, 2023

Developing Efficient Algorithms

Table of Contents

Section 1

Big O Notation (Recap)

Execution time vs growth rate

- Suppose two algorithms perform the same task, such as search (linear search vs. binary search). Which one is better?
- You could implement both and measure their execution times.
- However, there are 2 problems:
    - Other tasks running on your computer may be using computing resources which will affect the execution times.
    - The execution will depend on the input, such as the number of elements or where items are in the data structure.
- Thus, we instead analyse the *growth rate* of the algorithm.

Measuring Algorithmic Efficiency Using Big O Notation

- We use Big O notation to represent the order of magnitude of the growth rate of an algorithm.
  - e.g O(n) means the growth rate is of order of n in magnitude.
  - i.e the execution time is proportional to the size of the array.
- Big O notation represents the growth rate in the *worst-case scenario*, so the algorithm will never be slower than this.
- As Big O is measuring the order of magnitude, we ignore multiplicative constants:
  - $O(n/2) = O(n) = O(100n)$

Section 2

Time Complexity Analysis

Example 1

```
long k = 0;
for(int i = 1; i <=n; i++){
    k = k + 5;
}
```

- What is the time complexity of this code?

Example 1

```
long k = 0;
for(int i = 1; i <=n; i++){
    k = k + 5;
}
```

- What is the time complexity of this code?
- Since the loop is executed n times, the time complexity of the loop is:
- $T(n)$ = constant b + (constant c)*$n = O(n)$

Example 1 - Testing

```java
public class PerformanceTest {
  public static void main(String[] args) {
    getTime(1000000);
    getTime(10000000);
    getTime(100000000);
    getTime(1000000000);
  }

  public static void getTime (long n) {
    long startTime = System.currentTimeMillis();
    long k = 0;
    for (int i = 1; i <= n; i++) {
      k = k + 5;
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Execution time for n = " + n
      + " is " + (endTime - startTime) + " milliseconds");
  }
}
```

```
Execution time for n = 1000000 is 6 milliseconds
Execution time for n = 10000000 is 61 milliseconds
Execution time for n = 100000000 is 610 milliseconds
Execution time for n = 1000000000 is 6048 milliseconds
```

Example 2

```
long k = 0;
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        k = k + i + j;
    }
}
```

- What is the time complexity of this code?

Example 2

```
long k = 0;
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        k = k + i + j;
    }
}
```

- What is the time complexity of this code?
- The internal calculation (i + j + k) runs in constant time.
- The outer loop runs n times; for each outer loop iteration, the inner loop will run n times.
- Therefore the time compelxity for the loop is:
- $\mathrm{T}(n) = (\text{constant c}) * n * n = \mathrm{O}(n^2)$

Example 3

```
long k = 0;
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
        k = k + i + j;
    }
}
```

- What is the time complexity of this code?

Example 3

```
long k = 0;
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
        k = k + i + j;
    }
}
```

- What is the time complexity of this code?
- The internal calculation (i + j + k) runs in constant time.
- The outer loop runs n times.
- For i=1, 2, ... the inner loop will execute 1 time, 2 times, and n times. Thus the time complexity is:
- $T(n) = c + 2c + 3c + 4c + \ldots + nc$
- $T(n) = cn(n+1)/2$
- $T(n) = \frac{c}{2}n^2 + \frac{c}{2}n = O(n^2)$

Useful summations

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \ldots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$

$$a^0 + a^1 + a^2 + a^3 + \ldots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = O(2^n)$$

Section 3

Further Time Complexity Analysis

Binary Search

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low - 1; // Now high < low, key not found
  }
}
```

Binary Search - Runtime Complexity

- Let T(n) denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k = \log n$.

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low - 1; // Now high < low, key not found
  }
}
```

Binary Search - Runtime Complexity

- Let T(n) denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k = \log n$.
- $T(n) = T(\frac{n}{2}) + c$

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low - 1; // Now high < low, key not found
  }
}
```

Binary Search - Runtime Complexity

- Let T(n) denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and k = logn.
- $T(n) = T(\frac{n}{2}) + c = T(\frac{n}{2^2}) + 2c$
- $T(n) = T(\frac{n}{2^k}) + kc$
- $T(n) = T(\frac{n}{2^{logn}}) + clogn$
- $T(n) = T(\frac{n}{n}) + clogn$
- $T(n) = 1 + clogn = O(logn)$

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low - 1; // Now high < low, key not found
  }
}
```

Common Recurrence Functions

Recurrence Relation	*Result*	*Example*
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	Checkpoint Question 22.20
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 23)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

Recursive Fibonacci

```java
/** The method for finding the Fibonacci number */
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

Recursive Fibonacci - Runtime Complexity

```java
/** The method for finding the Fibonacci number */
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

- Let T(n) denote the complexity for the algorithm that finds fib(n) and c denote the constant time for comparing the index with 0 and 1; that is, T(1) and T(0) are c.
- $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$
- $T(n) \leq 2T(n\text{-}1) + c$
- $T(n) \leq 2(2T(n\text{-}2) + c) + c$
- $T(n) \leq 2(2(2T(n\text{-}3) + c) + c) + c$
- $T(n) \leq O(2^n)$

Recursive Fibonacci - Runtime Complexity

```
T(n) =  T(n-1) + T(n-2) + c
     =  T(n-2) + T(n-3) + c + T(n-2) + c
     =  2T(n-2) + T(n-3) + 2c
     >= 2T(n-2) + 2c
     =  2T(n-3) + T(n-4) + c) + 2c
     =  2T(n-3) + 2T(n-4) + 2c + 2c
     =  2T(n-3) + 2T(n-4) + 4c
     =  2T(n-4) + 2T(n-5) + 2c + 2T(n-4) + 4c
     =  4T(n-4) + 2T(n-5) + 6c
     >= 4T(n-4) + 6c
     =  4T(n-5) + 4T(n-6) + 10c
     =  4T(n-6) + 4T(n-7) + 4c + 4T(n-6) + 10c
     =  8T(n-6) + 4T(n-7) + 14c
     >= 8T(n-6) + 14c
     =  .......................
     =  .......................
     =  2^k T(n - 2k) + (2^(k+1)-2)c

for, k steps, n - 2k = 0 or, 1 the recursion stops
             => n = 2k
             => k = n / 2

    so, from the recurrence relation
    = 2^(n/2) * T(0) + (2^((n/2) +1) -2) * c
    = 2^(n/2) * c + (2^((n/2) +1) -2) * c
    = Ω(2^(n/2))
```

Iterative Fibonacci

```java
import java.util.Scanner;

public class ImprovedFibonacci {
  /** Main method */
  public static void main(String args[]) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for the Fibonacci number: ");
    int index = input.nextInt();

    // Find and display the Fibonacci number
    System.out.println(
      "Fibonacci number at index " + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long n) {
    long f0 = 0; // For fib(0)
    long f1 = 1; // For fib(1)
    long f2 = 1; // For fib(2)

    if (n == 0)
      return f0;
    else if (n == 1)
      return f1;
    else if (n == 2)
      return f2;

    for (int i = 3; i <= n; i++) {
      f0 = f1;
      f1 = f2;
      f2 = f0 + f1;
    }

    return f2;
  }
}
```

```
Enter an index for the Fibonacci number: 6 ↵Enter
Fibonacci number at index 6 is 8
```

```
Enter an index for the Fibonacci number: 7 ↵Enter
Fibonacci number at index 7 is 13
```

Dynamic programming

- This iterative implementation uses an approach called *dynamic programming*.
- Dynamic programming is the process of solving subproblems, then combining the solutions of the subproblems to obtain an overall solution.
- This naturally leads to a recursive solution.
- However, it would be inefficient to use recursion, because the subproblems overlap.
- The key idea behind dynamic programming is to solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.
- Can we write a recursive approach that does this?

Improved Recursive Approach

```java
    private static Map<Integer, Integer> results = new HashMap<>();

    public static int fibonacci(int n) {
        if (results.containsKey(n)) {
            return results.get(n);
        }

        int result;
        if (n <= 1) {
            result = n;
        } else {
            result = fibonacci(n - 1) + fibonacci(n - 2);
        }

        results.put(n, result);
        return result;
    }

    public static void main(String[] args) {
        // Example usage
        int n = 10;
        int result = fibonacci(n);
        System.out.println("Fibonacci(" + n + ") = " + result);
    }
```

Section 4

Further Performance Analysis

Considering performance

- The Big O notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient.
    - Two algorithms may both be $O(n)$ but if one completes in significantly less time then we shouldn't think of these algorithms as equally efficient.
- We will now look at another common algorithm, starting from an inefficient implementation and step-by-step improving its efficiency.

Greatest Common Divisor (GCD)

- The greatest common divisor (GCD) of two integers is the largest number that can evenly divide both integers.
- We will start by looking at a *brute force* algorithm.
- Brute force refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way.
  - As a result, such an algorithm can end up doing far more work to solve a given problem than a more sophisticated algorithm might do.
  - On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

Greatest Common Divisor (GCD) - Brute force approach

```java
public static int gcd(int m, int n) {
  int gcd = 1;

  for (int k = 2; k <= m && k <= n; k++) {
    if (m % k == 0 && n % k == 0)
      gcd = k;
  }

  return gcd;
}
```

- The greatest common divisor (GCD) of two integers is the largest number that can evenly divide both integers.
- We will start by looking at a *brute force* algorithm.
- Brute force refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way.
    - Drawback: does more work than more efficient algorithms.
    - Plus: easier to implement.

Greatest Common Divisor (GCD) - Output

```
Enter first integer: 2525 ↵Enter
Enter second integer: 125 ↵Enter
The greatest common divisor for 2525 and 125 is 25
```

```
Enter first integer: 3 ↵Enter
Enter second integer: 3 ↵Enter
The greatest common divisor for 3 and 3 is 3
```

Greatest Common Divisor (GCD) - Improved approach

```
for (int k = n; k >= 1; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
  }
}
```

- Rather than searching a possible divisor from 1 up, it is more efficient to search from n down.
- Here, we assume m$\geq$n.
- This way, the first common divisor found will be the greatest.
- When the GCD has been found, we break out of the loop.

Greatest Common Divisor (GCD) - Further improved approach

```
if (m % n == 0) return n;

for (int k = n / 2; k >= 1; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
  }
}
```

- A divisor for a number n cannot be greater than n / 2, so you can further improve the algorithm by updating the loop conditions.
- However, you need to check whether n is a divisor for m (e.g n=8, m=16) and if so, return n.
- Again, when the GCD has been found, we break out of the loop.

Greatest Common Divisor (GCD) - Euclid's Algorithm

```
public static int gcd(int m, int n) {
  if (m % n == 0)
    return n;
  else
    return gcd(n, m % n);
}
```

- A more efficient algorithm for finding the GCD was discovered by Euclid around 300BC.
- It can be defined recursively as follows:
  - Let gcd(m, n) denote the GCD for integers m and n:
  - If m % n = 0, gcd(m, n) = n.
  - Otherwise gcd(m, n) = gcd(n, m%n).
- For example:
  - gcd(15, 9) = gcd(9, 15%9) = gcd(9,6).
  - gcd(9,6) = gcd(6, 9%6) = gcd(6,3).
  - Since 6%3 = 0, gcd=3.

Why does this work?

- This works because of an observation that we can make about common divisors.
- Suppose m % n = r.
- Thus m = qn + r, where q is the quotient of $\frac{m}{n}$.
- Since m = a×gcd(m, n), n = b×gcd(m, n), and r = m - qn:
  - r = (a - qb)(gcd(m, n)).
- Therefore m, n and r all have the same gcd.
  - i.e gcd(m, n) = gcd(n, r), where r = m % n.

Time complexity

- The best case time complexity is O(1), which happens when m % n = 0.
- For the worst case, consider the method calls:
  - We are doing m % n every 2 method calls.
  - If n$> \frac{m}{2}$, then m % n = m - n $< \frac{m}{2}$
  - If n$\leq \frac{m}{2}$, then m % n $< \frac{m}{2}$ since the remainder will always be less than m.
- Therefore every 2 function calls, n is reduced at least half.
- After invoking gcd k times, this second parameter is less than $\frac{n}{2^{\frac{k}{2}}}$ (where n is the input size), which is greater than or equal to 1.
- $\frac{n}{2^{\frac{k}{2}}} \geq 1 \rightarrow n \geq 2^{\frac{k}{2}} \rightarrow logn \geq \frac{k}{2} \rightarrow k \leq 2logn$
- Therefore the worse case time complexity is O(logn).

Closing thoughts

- Consider algorithms and data structures that best fit the time and space complexity requirements of the problem you are trying to solve.
  - If you have very limited memory, some algorithms won't be suitable even if they are very time efficient.
  - If you are primarily performing a specific operation e.g searching through data, it makes sense to choose a data structure that is efficient at this operation even at the expense of other operations.
- If an algorithm has found the best solution to a problem already, it doesn't need to be run for any more iterations.
- Just because two algorithms have the same time complexity, doesn't mean they are equivalent. A significant amount of today's data structure efficiency research focuses on shaving off small amounts of complexity e.g from $O(2^{0.337n})$ to $O(2^{0.291n})$ in subset sum problem.

Section 5

Lecture summary

Lecture summary

Questions

Thank you! Questions?