SE 2205a: Data Structures and Algorithm Design

Unit 3 – Part 3: Algorithm - Case Study

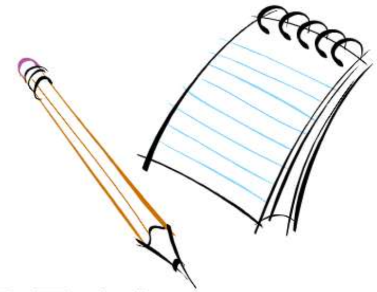
"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" *~Colin Powell*

Dr. Quazi M. Rahman, Ph.D, P.Eng, SMIEEE
Office Location: TEB 263
Email: QRAHMAN3@uwo.ca
Phone: 519-661-2111 x81399

"Genius is 1% talent and 99% percent hard work..."
~ Albert Einstein

Outline

- Case Study: Fibonacci Numbers
- In-Class Discussion on –
  - The Efficiency of Implementations of the ADT List
  - The Efficiency of Implementations of the ADT Map
- Case Study: GCD Algorithm



Case Study: Fibonacci Numbers (Recursive implementation)

- Fibonacci series:

Index: 0, 1, 2, 3, 4, 5, 6...
Value: 0, 1, 1, 2, 3, 5, 8…

```
Algorithm fib(index)
if (index == 0)then
    return 0;
else if (index == 1)then
    return 1;
else
    return fib(index - 1) + fib (index -2)
```

```
/** 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);
}
```

Case Study: Fibonacci Numbers – A Closer Look

```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 in a dynamic design
    return fib(index - 1) + fib(index - 2);
}
```

```
Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…
         indices: 0 1 2 3 4 5 6 7  8  9  10 11

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2
```

Complexity for Recursive Fibonacci Numbers

Assumptions:

- The time-complexity of the algorithm for data set n is T(n).
- Constant time for comparing index0 and 1 is c, i.e, T(0) = T(1) = c.

Time Complexity:

$T(n) = T(n-1) + T(n-2) + c$

Now that, T(n-2) <T(n-1), we can write,

$T(n) \leq 2T(n-1) + c$ ...... (1)

Now that, T(n-1) = T(n-2) + T(n-3) + c, we can write,

$T(n) \leq 2(2T(n-2) + c) + c$, which can be rearranged as,

$T(n) \leq 2^2T(n-2) + 2c + c$ ...... (2)

Expanding the R.H.S till the last index (n-1) we get,

$T(n) \leq 2^{n-1}T(n - (n-1)) + 2^{n-1-1}c + \ldots + 2c + c$

$T(n) \leq 2^{n-1}T(1) + 2^{n-2}c + \ldots + 2c + c$

$T(n) \leq 2^{n-1}T(1) + (2^{n-2} + \ldots + 2 + 1)c$; using the series summation (see slide #8, Unit 3-p2), we get

$T(n) \leq 2^{n-1}c + (2^{n-1} - 1)c$

$T(n) \leq 2^n c - c$

$T(n) \leq O(2^n)$; Therefore, the recursive Fibonacci method takes $O(2^n)$. How about space complexity?

```
Algorithm fib(index)
if (index == 0)then
        return 0;
else if (index == 1)then
        return 1;
else
        return fib(index – 1) + fib (index -2)
```

Case Study: Iterative version of Fibonacci Numbers

```
/** Iterative Version*/
public static long fib(long index){
  long f0 = 0; // For fib(0)
  long f1 = 1; // For fib(1)
  long f2 = 1; // For fib(2)
  if (index == 0) return f0;
  else if (index == 1) return f1;
  else if (index == 2) return f2;
  for (int i = 3; i <= index; i++) {
    f0 = f1;
    f1 = f2;
    f2 = f0 + f1;
  }
  return f2;
}
```

This is an example of dynamic algorithm that solves subproblems, then combine the solutions of subproblems to obtain an overall solution.

```
Algorithm fib(index)
f0 ← 0
f1 ← 1
f2 ← 1
if(index == 0)then
    return f0
else if (index == 1) then
    return f1
else if (index == 2) then
    return f2
for (i ← 3 to i ≤ index) do
    f0 ← f1
    f1 ← f2
    f2 ← f0 + f1
return f2
```

Complexity of this iterative algorithm is O(n). This is a tremendous improvement over the recursive algorithm. How about space complexity?

Fibonacci Number: The Dynamic Programming

combining the solutions of subproblems to obtain an overall solution.

```
                  f0 f1 f2
Fibonacci series: 0  1  1  2 3 5 8 13 21 34 55 89…

         indices: 0  1  2  3 4 5 6 7  8  9  10 11

                    f0 f1 f2
Fibonacci series: 0  1  1  2  3  5  8  13  21 34 55 89…

         indices: 0  1  2  3  4  5  6  7   8  9  10 11

                       f0 f1 f2
Fibonacci series: 0  1  1  2  3  5  8  13  21  34  55 89…

         indices: 0  1  2  3  4  5  6  7   8   9   10 11

                                                f0 f1 f2
Fibonacci series: 0  1  1  2  3  5  8  13 21 34 55 89…

         indices: 0  1  2  3  4  5  6  7  8  9  10 11
```

In-Class Discussion: Efficiency of Implementations of ADT List

`List`

- For array-based implementation
  - Add to end of list (if do not need to resize the array): O(1)
  - Add to list at a given position: O(n)
  - Retrieving an entry from a specific index: O(1)
  - Retrieving an entry: O(n)
- For linked implementation
  - Add to end of list (tail reference is implemented): O(1)
  - Add to list at a given position: O(n)
  - Retrieving an entry: O(n)

In-Class Discussion: Efficiency of Implementations of ADT Set

- Hash Set:
  - – Best case addition: O(1)
  - – Worst Case Addition: O(n)
  - – Retrieving an entry – Best case: O(1)
  - – Retrieving an entry – Worst case: O(n)
- Linked Hash Set:
  - – Add/remove – best: O(1), worst: O(n) (Collision)
  - – Retrieval – best: O(1), worst: O(n) (Collision)
- Tree Set (Sorted using heap-tree algorithm):
  - – Retrieval: O(log n).
  - – Remove, add: O(log n). [Note: if the sorting is implemented in an array, *remove* and *add* will be O(n)]

In Class Discussion: Efficiency of Implementations of ADT Map

- **Map (**Array-Based Implementations)

- **Unsorted worst-case efficiencies**
  - **Addition** **O(n) why not O(1)**
  - **Removal** **O(n)**
  - **Retrieval** **O(n)**
  - **Traversal** **O(n)**

- **Sorted worst-case efficiencies**
  - **Addition** **O(n) [Note: tree-based implementation: O(log n)]**
  - **Removal** **O(n) why not O(log n) [Note: tree-based implementation: O(log n)]**
  - **Retrieval** **O(log n)**
  - **Traversal** **O(n)**

In-Class Discussion: Efficiency of Implementations of ADT Map

- **Map (Linked Implementation)**

- **Unsorted worst-case efficiencies**
  - **Addition** **O(n)**
  - **Removal** **O(n)**
  - **Retrieval** **O(n)**
  - **Traversal** **O(n)**
- **Sorted worst-case efficiencies**
  - **Addition** **O(n)**
  - **Removal** **O(n)**
  - **Retrieval** **O(n)**
  - **Traversal** **O(n)**

Case Study: Iterative Implementation of GCD Algorithm Version 1

```
//Assumption: m>=n
public static int gcd(int m, int n) {
  int gcd = 1;
  for (int k = 2; k <= n; k++)
  {
    if (m % k == 0 && n % k == 0)
      gcd = k;
  }
  return gcd;
}
```

Note: GCD can not be a negative number.

Note: If both or either one of the two numbers (m and n) is negative, then use absolute values of m [i.e., Math.abs(m)] and n [i.e., Math.abs(n)] in the algorithm OR in the method-call

Algorithm gcd(m, n)
Input: two integers m and n (m>n)
Output: GCD of the two
gcd $\leftarrow$ 1
for (k $\leftarrow$ 2 to k $\leq$ m and k $\leq$ n) do
  if (m mod k == 0 and n mod k == 0)then
    gcd $\leftarrow$ k
return gcd

- Worst case time complexity of the algorithm: O(n).
- Question 1: In terms of design concept what kind of Algorithm is GCD Algorithm? (In-Class Discussion)
- Question 2: In case Implementation what kind of Algorithm is GCD Algorithm? (In-Class Discussion)

Case Study: Iterative Implementation of GCD Algorithm Version 2

```
//Assumption: m>=n
public static int gcd(int m, int n) {
  int gcd = 1;
  for (int k = n; k >= 2; k--) {
  if (m % k == 0 && n % k == 0) {
    gcd = k;
    break;
   }
  }
  return gcd;
}
```

```
Algorithm gcd(m, n)
Input: two integers m and n (m>n)
Output: GCD of the two
gcd ← 1
for (k ← n to k ≥ 2) do
  if (m mod k == 0 and n mod k == 0)then
    gcd ← k
    break
return gcd
```

- Best case time complexity of the algorithm: O(1).
- Worst case time complexity of the algorithm: O(n).

Case Study: Iterative Implementation of GCD Algorithm Version 3

```
//Assumption: m>=n
 public static int gcd(int m, int n) {
    int gcd = 1;
    if (m == n) return m;
    for (int k = n / 2; k >= 2; k--) {
      if (m % k == 0 && n % k == 0) {
        gcd = k;
        break;
      }
    }
    return gcd;
  }
```

Note: A divisor of a number n, can not be greater than n/2

Algorithm gcd(m, n)
Input: two integers m and n (m>n)
Output: GCD of the two
gcd ← 1
if (m == n)
  return m
for (k ← n/2 to k ≥ 2) do
  if (m mod k == 0 and n mod k == 0)then
    gcd ← k
    break
return gcd

- Worst case complexity of the algorithm: O(n/2) = O(n).

Comment: Although all three versions have same time complexity, version 3 is more efficient than the other two because it is using approximately half of the value of n in the worst case.

Case Study: Recursive Implementation of GCD Algorithm: `Euclid's` Algorithm (300 BC)

- This algorithm provides a recursive solution to the GCD problem.

```
//Assumption: m>=n

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

Algorithm gcd(m, n)
Input: two integers m and n (m>n)
Output: GCD of the two
if (m mod n == 0) then
  return n
else
  return gcd(n, m mod n)

- Worst case complexity of the algorithm: O(log n).
- *Note 1: Recursive solutions are not always more expensive (in terms of the time complexity) than iterative solutions.*
- *Note 2: Recursive solutions are always expensive in terms of space-complexity.*

Analyzing `Euclid's` Algorithm (worst case)

```
Algorithm gcd(m, n)
Input: two integers m and n (m>n)
Output: GCD of the two
if (m % n == 0) then
  return n
else
  return gcd(n, m % n)
```

- Assuming $m>n$: $m\%n < m$
- If $n>m/2$, $m\%n = m - n < m/2$, therefore, $n\%(m\%n)<n/2$
- Euclid's algorithm recursively invokes the gcd method.
  - It first calls gcd(m, n),
  - then it calls gcd(n, m%n),
  - then it calls gcd(m%n, n%(m%n)) and so on.
- Since $m\%n < m/2$ and $n\%(m\%n) < n/2$; the argument passed to the gcd method is reduced by half after every two iterations.
  - After calling gcd for two times, the second argument is less than $n/2$.
  - After calling gcd for four times, the second argument is less than $n/4 = n/(2^{4/2}) = n/(2^2)$
  - After calling gcd for six times, the second argument is less than $n/8 = n/(2^{6/2}) = n/(2^3)$
  - After calling gcd for k times, the second argument is less $n/(2^{k/2})$ which is greater than or equal to 1.
    - $n/(2^{k/2}) \geq 1 \Rightarrow n \geq 2^{k/2} \Rightarrow \log n \geq k/2 \Rightarrow k \leq 2\log n = O(\log n) = O(\log(\min(m,n)))$.

In-Class Question: What will be the best-case scenario?

In-Class Review

- The algorithm in finding the value from a particular index in a Fibonacci sequence is an example of the following design-concept-based Algorithm:
  a) Divide and Conquer
  b) Dynamic
  c) Decrease and Conquer
  d) Greedy
  e) Reduction

In-Class Review

- An ArrayList, or a dynamically resizing array, is a class in Java that allows you to have the benefits of an array while offering flexibility in size. You won't run out of space in the ArrayList since its capacity will grow as you insert elements.
- An ArrayList is implemented with an array. When the array hits capacity, the ArrayList class will create a new array with double the capacity and copy all the elements over to the new array.

- How do you describe the runtime of insertion?

In-Class Review

What would be the time-complexity of the following if-else statement?

```
if (condition)
        {Block 1}
else
        {Block 2}
```

End of this Unit