School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Complexity

Overview

- Today we will have
  - Review on the topic of last session
  - Formal definition of Big O

Review

- We’ve talked about complexity in general terms.

- Assumptions:
  - The complexity analysis focuses on algorithms
  - The input size is taken as argument.
  - The machine model is used to eliminate the influence of hardware

- The running time complexity of an algorithm matters in
  - the worst case
  - the average case

Review

- Is it always easy to find the complexity of an algorithm?
- We provide a range for the complexity that we are after
  - Upper bound
  - Lower bound
- Usually used for the worst case or the average case.
- When you introduce an instance of the problem as the worst case, is it always possible to prove that it is the worst case?
  - No. (but for the simple search example it was obvious!)
  - Formally, it is usually called a hard instance.
  - It gives a lower bound on the worst case complexity, but for proving an upper bound we need to go generally for the proof

Example: A simple search function

```
// pseudo code
search(list, item)
{
    for(i=1 to n)
        if list[i]==item
            return i

}
```

Required time:

- Set up time
- For loop

Example

- The for loop will take approximately n times the effort to make decision on one item.
- Set-up for this algorithm is constant.
- Mathematically, the execution time is equal to $cn + s$.
  - Where c and s are constants and represents the overhead per iteration, and the set-up costs, respectively.
- $f(n) = cn + s$ represents the actual execution time of this searching process.

Big O [O(g(n))] – formal definition

- If f(n) represents the *actual* execution time of an algorithm, then, as we don't always know the details of f(n), we approximate it.

- $f(n) = O(g(n))$ if there exist positive constants c and n_o such that $f(n) <= c\, g(n)$ when $n >= n_o$.

- We refer to this as Big-Oh(g(n)), O(g(n)) or "order g(n)"

General Rules

- Some mathematical background is required for analyzing computational complexity

Rule 1. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

1. $f_1(n) + f_2(n) = O(g_1(n)+g_2(n))= O(max(g_1(n),g_2(n)))$
2. $f_1(n) * f_2(n) = O(g_1(n)*g_2(n)))$

Rule 2. If f(n) is a polynomial of degree k, then

$$f(n) = O(n^k)$$

Rule 3. if $f(n)=\log (n^k)$ then $f(n)= O(\log n)$ for any constant k

- Let's see some examples

Example

- $n+1 = O(n)$?
- $n+2 = O(n)$?
- $2n+2 = O(n)$?
- $n^2 = O(n)$?
- $\mathrm{sqrt}(n) = O(n)$?
- $1 = O(n)$?
- $\log n = O(n)$?
- $n \log n = O(n)$?

Example

- $n = O(n^2)$?

- $n^2 = O(n^2)$?

- $n^3 = O(n^2)$?

- We want our Big Oh bounds to be:
  - Tight: We want g(n) to be as close to f(n) as it can be.
  - Simple: we can drop low order terms and constants. (why?)

- Growth rates are important

How do you find the complexities?

- We have simple rules
  - Simple statements (Math operators: +, -, &&, *, etc …, Assignment , Array indexing, Comparisons) are all O(1)
  - The running time of a loop is at most the running time of the statements inside the loop (including tests) multiplied by the number of iterations
    - Nested loops?
  - The running time of an if/else statement is at most the running time of the test plus the larger of the running times of the statements in the if and else block.

Simple Statement

- Simple statements:
  - Math operators: +, -, &&, *, etc …
  - Assignment , array indexing, …
  - Comparison

- The simple statements are all O(1)

- What about blocks of simple statements?

Simple Statement

- Consider the code block below

```
int next, n1, n2;

next = n1 + n2;
n2 = n1;
n1 = next;
```

- These statements are all O(1).
- They take a constant amount of time to execute, ***independent of the input size!***
- The complexity of the entire code segment is O(1).
  – These statements altogether still take a constant amount of time to execute.

Loops

- For-loops: (n is the input)

```
int counter = 0;
for(int i = 0; i< n ; i++){
  counter += i;
 }
```

- The running time of a for loop is at most the running time of the statements inside the for loop (including tests) multiplies the number of iterations.
- O(n*[complexity of statements inside the loop])
  - O(n)

Loops

```
int counter = 0;
for(int i = 0; i< 100; i++){
  counter += i;
 }
```

- The statements are performed 100 times.
- The complexity of the entire code segment is 100 * [the complexity of the statements inside the loop]
  - 100*c =O(1)

Loops

- Nested loops:

```
int counter = 0;

for(int i = 0; i< n ; i++){
  for(int j = 0; j< n ; j++){
    counter ++;
  }
 }
```

- The total running time of the statements that form a group of nested loops is the running time of the inner statements multiplied by the product of the sizes of all the loops.
- O(n^2)

Loops

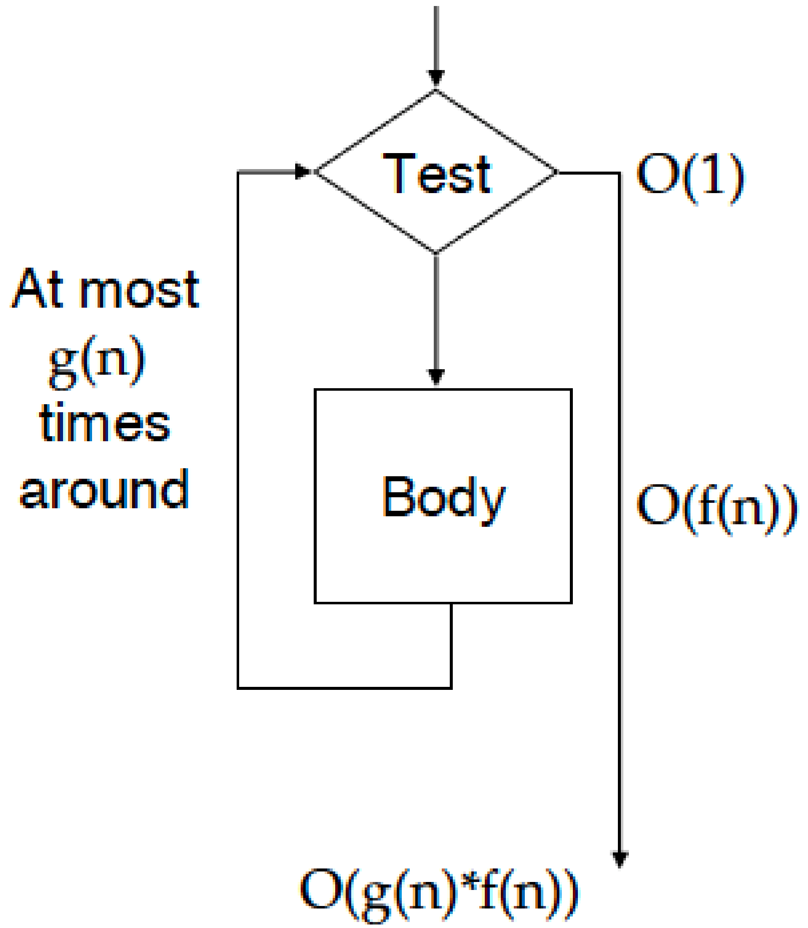
- Nested loops

```
for(int i = 0; i< n: i++){
  for(int j = 0; j< m; j++){
    counter ++;
  }
 }
```

```
for(int i = 0; i< n; i++){
  for(int j = 0; j< 100; j++){
    counter ++;
  }
 }
```

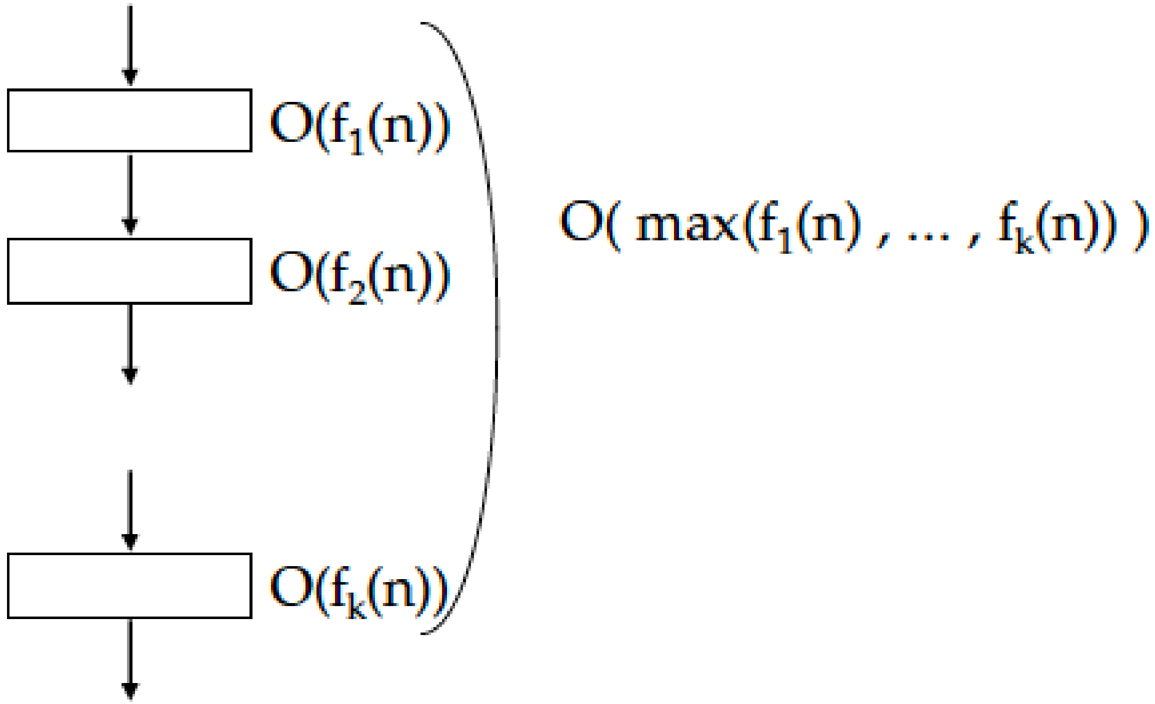
Loops

- While-loops:

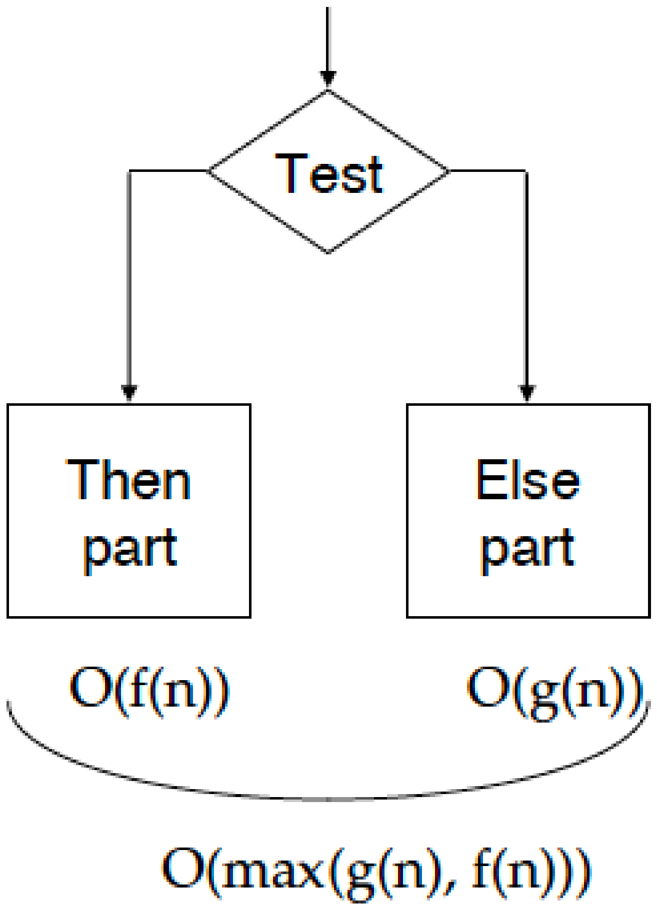


Consecutive Statements

- Block of statements without function calls is just summation.



If/else Statement



- The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of the statements in the if and else block.

If/else Statement

```
if(a>b){
  for(int i=0; i<n; i++){
    counter ++;
  }
 }else{
  counter = 0;
 }
```

Example

- Iterative version of Fibonacci number calculation

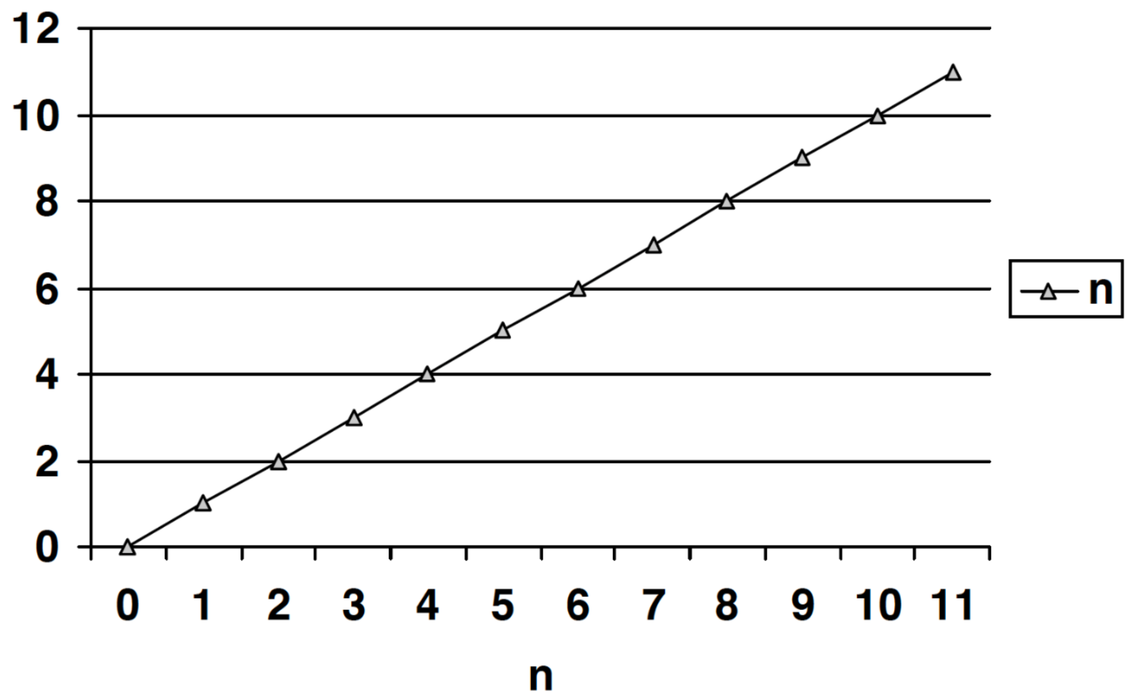
- The program structure tree

- O(n)

```
int Fib(int n) {

    if (n==0 || n==1) {
        return n;
    }

    int n1=0;
    int n2=1;
    int next;

    for(int i = 2; i <= n; i++) {
        next = n1 + n2;
        n1 = n2;
        n2 = next;
    }
    return next;
}
```

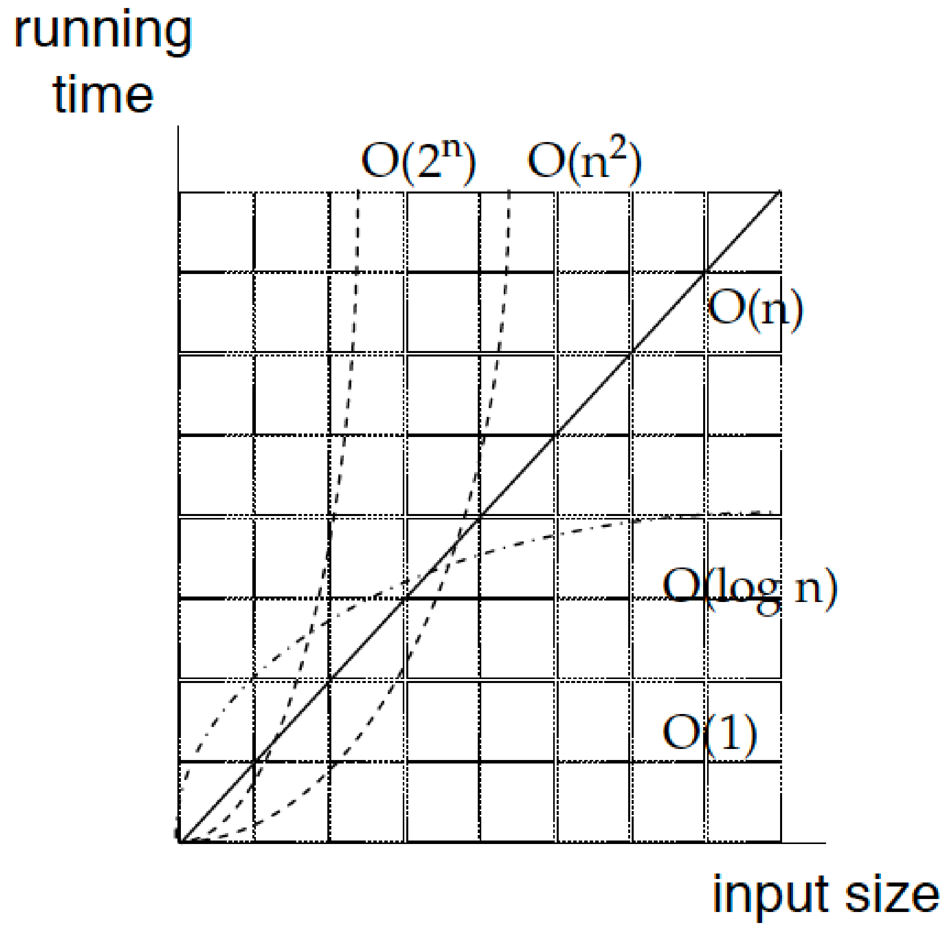
Example

- The iterative version of Fibonacci has a linear growth rate.
- The run time grows in proportion to the magnitude of the Fibonacci number we are computing.



Typical Big-Oh Running Times

Big-Oh	Informal name
O(1)	constant
O(log n)	logarithmic
O(n)	linear
O(n log n)	*n log n*
O(n^2)	quadratic
O(n^3)	cubic
O(2^n)	exponential



Summary

- Notations:
  - Big O: for presenting an upper bound
- Simple Rules
  - Summation and multiplication
  - Polynomials with degree k: O(n^k)
  - Analysis of Simple algorithms:
    - Simple statement, If/else statements, Loops, Consecutive statement