CS 242 - CS252
Data Structures

Algorithm Analysis

Why need algorithm analysis ?

- Writing a working program is not good enough
- The program may be inefficient!
- If the program is run on a large data set, then the ***running time*** becomes an issue

Algorithm Efficiency

- The content of the input affects the running time
  - The input size (number of items in the input) is the main consideration
    - E.g. sorting problem: the number of items to be sorted
    - E.g. multiply two matrices together: the total number of elements in the two matrices
- Machine model assumed Instructions are executed one after another, with no concurrent operations
  - Not parallel computers

Experimental Analysis

Two algorithms for composing a string of repeated characters

```java
/** Uses repeated concatenation to compose a String with n copies of character c. */

public static String repeat1(char c, int n) {

    String answer = "";
    for (int j=0; j < n; j++)
        answer += c;
    return answer;
}
```

```java
/** Uses StringBuilder to compose a String with n copies of character c. */

public static String repeat2(char c, int n) {

    StringBuilder sb = new StringBuilder( );
    for (int j=0; j < n; j++)
        sb.append(c);
    return sb.toString( );
}
```

Experimental Analysis

Results of timing

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

3 days !

< 1 sec

Algorithm Efficiency

- Predicting the resources that the algorithm requires such as
  - Memory
  - Communication bandwidth
  - Computational time (usually most important)
- Factors affecting the running time
  - Computer, compiler, algorithm used, input to the algorithm

Moving Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
- Is performed by studying a high-level description of the algorithm without need for implementation.
- Takes into account all possible inputs.

Time complexity & Primitive Operation

- ***Time complexity*** is the execution time it takes for your algorithm to solve a problem.

- ***Primitive Operations are*** basic computations performed by an algorithm such as:
  - Evaluating an expression
  - Following an object reference
  - Performing an arithmetic operation (for example, adding two numbers)
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

Code	# operations
`public static int sum( int n )`	
`{`	
`    int partialSum;`	*The declarations count for no time*
`    partialSum = 0;`	1
`    for( int i = 1 ; i <= n ; i++ )`	$2N + 2$
`        partialSum += i * i * i;`	4N
`    return partialSum;`	1
`}`	

Total $6\boldsymbol{n} + 4$

- f(n)= $6\boldsymbol{n} + 4$

Big-O Notation

- We are not interested in knowing the exact number of operations the algorithm performs. mainly interested in knowing how the number of operations grows with increased input size.
- Why?
  - Given large enough input, the algorithm with faster growth will execute more operations and takes long time
- The simplification of efficiency is known as **big-O analysis.**
- We don't need to determine the complete measure of efficiency, only the factor that determines the magnitude. This factor is the **big-O**, and expressed as *O(n)* – that is, on the order of n.

Big-O Notation

- The big O notation can be derived from $f(n)$ using the following steps:

1. Ignore the lower order terms and the coefficients of the highest-order term
2. Keep the largest term in the function and discard the others.
3. No need to specify the base of logarithm
4. Nested loop
   - Iterations = inner loop iterations x outer loop iterations
5. Consecutive program fragments (Larger O($f(n)$))
6. If statement (worst-case scenario)

Example 1

- **Linear loop**

```
i = 1
loop ( i <= n )
    application code
    i=i+1
end loop
```

O(n)

iteration	Value of **i**
1	1
2	2
3	3
…	…
n	n

n

- What if we change the incremental condition? Ex: i=i+2

Example 2

- **Logarithmic loop**

```
i = 1
loop ( i < n )
    application code
    i = i x 2
end loop
```

$O(\log_2 n)$

iteration	Value of **i**
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512
11	1024

$2^{\text{iteration}} < 1000$

Example 3

- **Logarithmic loop**

```
i=n
loop ( i >= 1)
    application code
    i = i / 2
end loop
```

$O(\log_2 n)$

iteration	Value of i
1	1000
2	500
3	250
4	125
5	62
6	31
7	15
8	7
9	3
10	1
Exit	0

$1000 / 2^{\text{iteration}} >= 1$

Example 4

- **Linear logarithmic**

```
i=1
loop ( i <= n )
    j=1
    loop ( j <=n )
        application code
        j= j x 2
    end loop
    i = i + 1
end loop
```

$\log_2 n$

n

$O(n\ [\log_2 n])$

Example 5

- **Nested loop**
  - **Quadratic**

```
i = 1
loop ( i <= n)
    j = 1
    loop ( j <= n )                 n
    application code        n
        j = j +1
    end loop
    i = i + 1
end loop
```

$O(n^2)$

Example 6

- **Nested loop**
  - **Dependent Quadratic:** inner loop iteration depend upon the outer loop counter value

```
i = 1
loop ( i <= n )
    j = 1
    loop ( j <= i )
        application code
        j = j + 1
    end loop
    i = i + 1
end loop
```

1 + 2 + 3 + … + 9 + 10 = 55

the average is 55/10 =5.5

(n+1) / 2

n

$O(n^2)$

Worst-case Running Time

- Worst-case running time of an algorithm
  - The longest running time for any input of size n
  - An upper bound on the running time for any input guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
- Best case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average case running time
  - May be difficult to define what "average" means

Consecutive Program Fragments

- The total running time is the maximum of the running time of the individual fragments.

```
sum = 0;
for ( i = 0 ; i < n ; i++ )
      sum = sum + i ;
```

n

```
sum = 0 ;
for ( i = 0; i < n; i++)
      for ( j = 0; j < 2n; j++ )
            sum++;
```

n^2

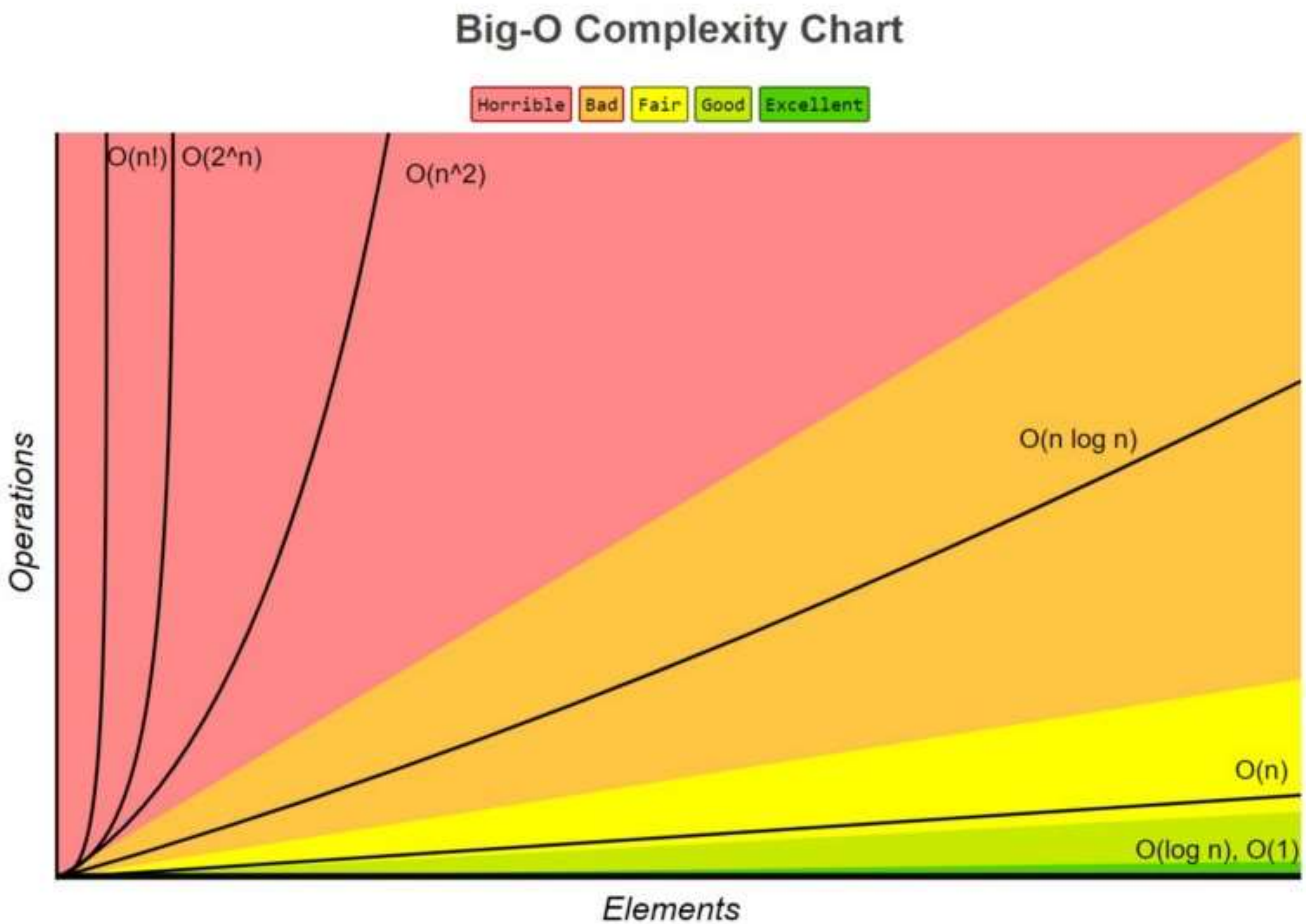
- The first loop runs in $O(n)$ time, the second - $O(n^2)$ time, the maximum is $O(n^2)$

If statement

- The running time is the maximum of the running times of S1 and S2.

```
if cond
{
    S1;
}
else
{
    S2;
}
```

Choose the worst case scenario to calculate O(f(n))



constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1 <	$\log n$ <	n <	$n \log n$	< n^2	< n^3	< a^n

Example

- Calculate the big-O notation for
  - $f(n) = n [(n+1)/2]$

Solution:

$\frac{1}{2} n^2 + \frac{1}{2} n$ (Remove the coefficient)

$n^2 + n$ (keep the largest term)

n^2

So, the big-O notation is stated as $O(f(n)) = O(n^2)$