- References:
  - Text book : Chapter 4
  - Previous CSC313 class notes

1. Introduction to Performance Analysis

- Goal : To analyze the efficiency of algorithms

- Definition :

  - The space complexity of a program is the amount of memory that it needs to run to completion. Example: additional storage such as recursive calls, new operation to allocate objects

  - The time complexity of a program is the amount of computer time that it needs to run to completion.

- Time Complexity

  - The time taken by a program

    T(p) = compile time (fixed) + run (or execution) time Ep(I)

    - Tp is the total time requirement for program p
    - Ep(I) is the total run time for program p with particular instance I.

  - For Ep(I), need to know a detailed knowledge of executable code and the time needed to perform each operation on specific hardware.

    For example : c = a+b; → load a; load b; add; store c

    ** very difficult.

- Use other methods to estimate T(p)

  - use system command such as "time" in Unix to approximate the run time. difficult to analyze!

  - set a global counter in your program to count the number of steps that a program needs to solve an instance I.

    very difficult for a complex problem, we may need to find out the best, average and worst case scenarios

    Example : Add two arrays a and b

    ```
    for (i=0; i < rows; i++)  /* count++ */
    for (j=0; j < cols; j++)  /* count++ */
        c[i][j] = a[i][j]+b[i][j]; /* count++ */
    ```

    Assume count = 0 (initially) and each step takes constant time, we have

    - i for-loop statement, executed *rows + 1* times,
    - j for-loop statement, executed *rows*(cols + 1)* times,
    - the statement in j-loop, executed *rows*cols* times
    - total counts : *2*rows*cols + 2*rows + 1*

    If *rows >> cols*, should interchange the matrices to minimize the total counts.

  - <u>Asymptotic notation</u> : The approximation of step counts (Only cover Big O here. You also need to know definitions of Big Omega Ω and Big Theta Θ)

- Def [Big O] : A function f(n) is said to be O(g(n)) iff there exist positive constants c and n_0 such that f(n) <= c*g(n) for all n, n >= n_0.
  Read : f of n is big o of g of n

- O(g(n)) is an upper bound of f(n), should try to find as small g(n) as possible

- ignore constants; drop low-order terms; interest to look at the growth rate as n getting bigger; use big O notation (upper bound)

- Example :

  - f(n) = 3n+2, O(n) because 3n+2 <= 4n for all n >= 2

  - f(n) = 10, O(1) because 10 <= 10*1 for all n > 0

  - f(n) = $10n^2$+4n+2, O(n^2) because f(n) <= 11*(n^2) for all n >= 5

  - f(n) = 6*(2^n)+n^2, O(2^n) because f(n) <= 7*(2^n) for all n >= 4

- You may think of f(n) is the running time of program, n is the input size of the program and g(n) is the approximate upper bound of f(n), i.e. worst case

- The following identities hold for Big Oh notation:

  $O(k\,f(n)) = O(f(n))$
  $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  $O(f(n))\,O(g(n)) = O(f(n)\,g(n))$

- Example 1: For matrix addition, we have

O(rows)	for i for-loop statement
O(rows*cols)	for j for loop statement
O(rows*cols)	for the statement in j for loop

  Total : O(rows*cols)

- Example 2 : factorial

```
res=1;
for (i=1; i<=n; i++)
        res=res*i
return res
```

  each iteration, constant c time,
  each time we reduce the number n by 1
  we have n iterations
  total time = c * n = O(n)

- Example 3 : Three algorithms for computing
  1 + 2 + … n for an integer n > 0

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
`for i = 1 to n`
`   sum = sum + i`
	`sum = 0`
	`for i = 1 to n`
	`{ for j = 1 to i`
	`      sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

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

- Main Problem : constant numbers and lower terms are eliminated. They may be a very large number.

- Here are the list of common time complexities:

$$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n) \text{ etc}$$

Time	*Name* $n \rightarrow$	*1*	*2*	*4*	*8*	*16*	*32*
1	Constant	1	1	1	1	1	1
log	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
n log n	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
n!	Factorial	1	2	24	40326	2E13	4E47

2. Efficiency of Implementations of ADT Bag (as present in Chapter 4 & 5)

- For array-based implementation (ArrayBag)
  Add to end of array O(1)
  Search array (getIndex()) O(1) best case, O(n) worst case

- For linked implementation (LinkedBag)
  Add to the beginning of linked list O(1)
  Search an entry O(1) best case, O(n) worst case

  Note: In general, you should consider worst case scenario, so, in the following table, you should focus on slower big-O numbers :

Operation	Fixed-Size Array	Linked
`add(newEntry)`	$O(1)$	$O(1)$
`remove()`	$O(1)$	$O(1)$
`remove(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`clear()`	$O(n)$	$O(n)$
`getFrequencyOf(anEntry)`	$O(n)$	$O(n)$
`contains(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`toArray()`	$O(n)$	$O(n)$
`getCurrentSize(), isEmpty(), isFull()`	$O(1)$	$O(1)$