Chapter 2 - Analysis of Algorithm Efficiency

- time complexity: how fast an algorithm runs, with input size n
  space complexity: how much space an algorithm requires

  $T(n) \approx c_{op}\, C(n)$
  T(n): running time of algorithm
  c_{op}: unit time to execute one instruction (clock speed)
  C(n): # of times/steps the basic operation is executed (a line of code operated most # of times)

- time complexity focuses on the ***order of growth***, not multiplicative constants

  order of growth in *increasing* order:
  $logn,\ \sqrt{n},\ n,\ nlogn,\ n^2,\ n^3,\ 2^n,\ n!$

  2^n and n! are referred to as "exponential-growth functions", only practical for small n.
  - examples of large n (and basically uncomputable result):
    2^n: wheat-chessboard problem
    n!: travel salesman problem (TSP)

- C_{worst}(n) is an input for which the algorithm runs the *longest* among all possible inputs of size n.
  C_{best}(n) is an input for which the algorithm runs the *fastest* among all possible inputs of size n.
  $C_{average}$(n) tells an algorithm's behavior on a typical / random input of size n.

- Example: Sequential Search

  Algorithm *SequentialSearch(A, K):*
  // Input: Array A with size n, search key K
  // Output: Index of the first element of A that matches K. or -1 if not found

```
i = 0
while i < n:
    if A[i] == K:
        return i
    i += 1
return -1
```

  C_{worst}(n) = n if K is not in the array or target is the last item of array
  C_{best}(n) = 1 if K is the first element of the array

  Suppose the probability that K is in the array is p, the probability that K is at each location i is $\frac{p}{n}$ and the probability that K is not in array is (1-p). Then:
  $C_{average} =$ time when K is in array + time when K is not in array
  $$C_{average} = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$
  $$= \frac{p}{n} \cdot (1 + 2 + 3 + \ldots + n) + n \cdot (1-p) = \frac{(1+n) \cdot n}{2} \cdot \frac{p}{n} + n \cdot (1-p) = \frac{np+p}{2} + n - np$$

  when p = 1 (K is in array), $C_{average}$(n) = $\frac{n+1}{2}$; when p = 0 (K is not in array), $C_{average}$(n) = n

- O defines the *upper* bound complexity of an algorithm. $n \in O(n^2)$
  > $O(g(n))$ contains the set of functions with a smaller/same order of growth of g(n)
  > $O(g(n)) \leq g(n)$

  Ω defines the *lower* bound complexity of an algorithm. $n \in \Omega(1)$
  > $\Omega(g(n))$ contains the set of functions with a higher/same order of growth of g(n)
  > $\Omega(g(n)) \geq g(n)$

  Θ defines the *exact* complexity of an algorithm. $n \in \Theta(n)$
  > $\Theta(g(n))$ contains the set of functions with a same order of growth of g(n)
  > $\Theta(g(n)) = g(n)$

- If $t(n) \in O(g(n))$, then, there exists some positive constant *c* and same non-negative integer n_0 such that $t(n) \leq c \cdot g(n)$ for all $n \geq n_0$

  If $t(n) \in \Omega(g(n))$,then, there exists some positive constant *c* and same non-negative integer n_0 such that $t(n) \geq c \cdot g(n)$ for all $n \geq n_0$

  If $t(n) \in \Theta(g(n))$, then, there exists some positive constant c_1 and c_2 and same non-negative int n_0 such that $c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$ (prove both $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$)

- *** If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$ TRUE**
  - proof:

  Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

  $$t_1(n) \leq c_1 g_1(n) \quad \text{for all } n \geq n_1.$$

  Similarly, since $t_2(n) \in O(g_2(n))$,

  $$t_2(n) \leq c_2 g_2(n) \quad \text{for all } n \geq n_2.$$

  Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

  $$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

  Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

  *** If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(min\{g_1(n), g_2(n)\})$ FALSE**
  - counter example:
  $n \in O(n)$, $n^2 \in O(n^2)$; $n + n^2 \in O(n^2)$ not *min(n,n²)* which is n.

*** If** $t_1(n) \in \Omega(g_1(n))$ **and** $t_2(n) \in \Omega(g_2(n))$**, then** $t_1(n) + t_2(n) \in \Omega(min\{g_1(n), g_2(n)\})$ **TRUE**

\- proof:

Since $t_1(n) \in \Omega(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that $t_1(n) \geq c_1 g_1(n)$ for all $n \geq n_1$.
Since $t_2(n) \in \Omega(g_2(n))$, there exist some positive constant c_2 and some nonnegative integer n_2 such that $t_2(n) \geq c_2 g_2(n)$ for all $n \geq n_2$.

a) Let us denote $c = c_1 + c_2$ and consider $n \geq n_0 = \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\geq c_1 g_1(n) + c_2 g_2(n) \\ &\geq c_1 \min\{g_1(n), g_2(n)\} + c_2 \min\{g_1(n), g_2(n)\} \\ &= (c_1 + c_2) \min\{g_1(n), g_2(n)\} \\ &\geq c \min\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence $t_1(n) + t_2(n) \in \Omega(\min\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the Ω definition being $c_1 + c_2$ and $\max\{n_1, n_2\}$, respectively.

*** If** $t_1(n) \in \Omega(g_1(n))$ **and** $t_2(n) \in \Omega(g_2(n))$**, then** $t_1(n) + t_2(n) \in \Omega(max\{g_1(n), g_2(n)\})$ **TRUE**

\- proof:

b) Let us denote $c = \min\{c_1, c_2\}$ and consider $n \geq n_0 = \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\geq c_1 g_1(n) + c_2 g_2(n) \\ &\geq c g_1(n) + c g_2(n) \\ &= c[g_1(n) + g_2(n)] \\ &\geq c \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence $t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the Ω definition being $\min\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

*** If** $t_1(n) \in \Theta(g_1(n))$ **and** $t_2(n) \in \Theta(g_2(n))$**, then** $t_1(n) + t_2(n) \in \Theta(max\{g_1(n), g_2(n)\})$ **TRUE**

\- proof:

Since both O and Ω are proved above, Θ holds as well.

*** If** $t_1(n) \in \Theta(g_1(n))$ **and** $t_2(n) \in \Theta(g_2(n))$**, then** $t_1(n) + t_2(n) \in \Theta(min\{g_1(n), g_2(n)\})$ **FALSE**

\- counter example:

$n \in \Theta(n)$, $n^2 \in \Theta(n^2)$; $n + n^2 \in \Theta(n^2)$ not *min(n,n²)* which is n.

- To compare order of growth, use division:
  - n^2 is faster than n, because $n^2/n = n$
  - n! is faster than (n-1)!, because n!/(n-1)! = n
  - 2^n is the same as 2^{n-1}, because $2^n/2^{n-1} = 2$ (constant)
  - 3^n is faster than 2^n, because $3^n/2^n = 1.5^n$
  - $log_2^2 n$ is faster than $log_2 n^2$, because $log_2^2 n = log_2 n \cdot log_2 n$, while $log_2 n^2 = 2 log_2 n$

- Mathematical analysis of **non-recursive** algorithms

Algorithm *MaxElement(A[0...n-1])*:

```
maxval = A[0]
for i from [1,n-1]:
    if A[i]>maxval:   # basic operation
        maxval = A[i]
return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = 1 + 1 + 1 + ... + 1 \ = n - 1 \in \Theta(n)$$

Algorithm *UniqueElement(A[0...n-1])*:

```
for i from [0,n-2]:
    for j from [i+1,n-1]:
        if A[i]==A[j]:
            return False
return True
```

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = (n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Algorithm *Binary(n)*:

 # find the number of binary digits in n

```
count = 1
while n>1:    # basic operation, operated for log2(n) times
    count += 1
    n //= 2
return count
```

- Mathematical analysis of **recursive** algorithms

Algorithm *Factorial(n)*:

```
if n==0: return 1
return F(n-1)*n
```

Initial: M(0) = 0

Recurrent: M(n) = M(n-1)+1 for n>0

M(n) = M(n-1) + 1, M(n-1) = M(n-2) + 1, M(n-2) = M(n-3) + 1 …

M(n) = M(n-1) + 1 = (M(n-2) + 1) + 1 = M(n-2) + 2 = M(n-3) + 3 = … = M(n-i) + i

Then, let i = n,

M(n) = M(n-n) + n = M(0) + n = 0+n = n

Algorithm *TowerofHanoi(n)*:

```
1- move all disk except for last disk to col2
2- move last disk to col3
3- move all disk except for last disk on top of last disk in col3
```

Initial: M(1) = 1 (only have 1 disk, takes 1 move to move it to another col)

Recurrent: M(n) = M(n-1) + 1 + M(n-1) (move blob to col2, move big disk to col3, move blob to col3)

$M(n) = 2M(n-1) + 1$, $M(n-1) = 2M(n-2) + 1$, $M(n-2) = 2M(n-3) + 1 \ldots$

$M(n) = 2M(n-1) + 1 = 2(2M(n-2) + 1) + 1 = 4M(n-2) + 2 = 4(2M(n-3) + 1) + 2 = 8M(n-3)+1+2+4 \ldots$

$M(n) = 2^i M(n-i) + 1 + 2 + 4 + \ldots + 2^{i-1}$

Then, let $i = n-1$,

$M(n) = 1 + 2 + 4 + \ldots + 2^{n-1} = \frac{2^n-1}{2-1} = 2^n-1 \in \Theta(2^n)$

Algorithm *BinRec(n)*:

 # find the number of binary digits in n

```
if n==1:
    return 1
return BinRec(n//2)+1
```

Initial: A(1) = 0

Recurrent: A(n) = A(n/2)+1 for n>1

Let's assume that $n = 2^k$, so we have:

Initial: A(2^0) = 0

Recurrent: A(2^k) = A(2^{k-1})+1 for n>1

$A(2^k) = A(2^{k-1})+1$, $A(2^{k-1}) = A(2^{k-2})+1$, $A(2^{k-2}) = A(2^{k-3})+1 \ldots$

$A(2^k) = A(2^{k-1}) + 1 = A(2^{k-2}) + 2 = A(2^{k-3}) + 3 = A(2^{k-i})+i$

Then, let $i = k$,

$A(2^k) = A(2^{k-k})+k = 0+k = log_2 n \in \Theta(logn)$