Time Complexity

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Performance Analysis

The program we write should

1. meet the specification.
2. work correctly.
3. be documented properly.
4. run effectively
5. be readable.

6. **use the storage effectively – space**
7. **run timely – time**

space & time complexity

The **space complexity** of a program is the amount of **memory** that it needs to run to completion.

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

Space complexity:

1. Fixed space requirements : c
   - that do not depend on input size, simple or fixed-size variables
2. Variable space requirements : $S_p(I)$
   - that depend on the instance I, stack, variable

The total space requirement for the program P:

$$\boldsymbol{S(P) = c + S_p(I)}$$

where **c** is a constant for fixed space and variable space for the instance I.

We are concerned about only $\boldsymbol{S_p(I)}$, but not c. **Why?**

Because we usually **compare** the algorithms of the programs.

Performance Analysis

Space complexity: $S(P) = c + S_p(I)$

Example: $S_{sum}(n)$ = ?

Program1.11

```
float sum(float list[], int n) {
  float total = 0;
  for (int i=0; i<n; i++)
    total += list[i];
  return total;
}
```

$\boldsymbol{S_{sum}(n)} = 0$ since the C passes list[] by its address.

Performance Analysis

Space complexity: $S(P) = c + S_p(I)$

Example: S_{rsum}(n=MAX_SIZE) = ?

Program1.12

```
float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}
```

The variable space requirement are for **two** parameters and **one** return address are saved in the system stack **per recursive call**:

$$sizeof(n) + list[]\,address + return\ address = 12$$

$$\boldsymbol{S_{sum}(n) = 12 * n}$$

Performance Analysis

Time complexity: The time taken by the program P:

$$T(P) = \textit{compile time } c + \textit{execution time } T_p(n)$$

Similarly, we are concerned about only $T_p(n)$, but not c.

Example: $Tp(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$

where n – number of execution, c for constant time for operation

We are not concerned about this, but …

Program step: a meaningful program segment whose execution time is independent of the instance characteristics.

Example:

$a = 2;$ ⇨ *1 step!!*

$a = 2 * b + 3 * c/d - e + f/g/a/b/c\,;$ ⇨ *1 step!!*

Performance Analysis

Example: How many **program steps** required?

Program sum	2n+3
`float sum(float list[], int n) {`	
`  float total = 0;`	1
`  for (int i=0; i<n; i++)`	n+1
`    total += list[i];`	n
`    return total;`	1
`}`	

Performance Analysis

Exercise: How many **program steps** required?

Program rsum	2n + 2
`float rsum(float list[], int n) {`	
`  if (n)`	n + 1
`    return rsum(list, n-1) + list[n-1];`	n
`  return 0;`	1
`}`	

Performance Analysis

Comparison:

Program sum

```
float sum(float list[], int n) {
  float total = 0;
  for (int i=0; i<n; i++)
    total += list[i];
  return total;
}
```

Program rsum

```
float rsum(float list[], int n) {
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
}
```

2n + 3 > 2n + 2
sum > rsum
(iterative) > (recursive)
⇨ $T_{iterative} > T_{recursive}$

Performance Analysis

Example: How many **program steps** required?

Program sum of matrix	
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],	
int c[][MAX_SIZE}, int rows, int cols) {	
for(int i=0; i<rows; i++)	rows + 1
for(int j=0; j<cols; j++)	rows * (cols+1)
c[i][j] = a[i][j] + b[i][j];	rows * cols
}	

step count = **2 rows*cols + 2 rows + 1**

Why step count?

It is to compare the **time complexities** of two programs that compute the same function and also to predict the **growth rate** in run time.

Example: Let's compute the step count for three programs and compare their time complexities.

1. $T_{add}(n)$ – adding two numbers
2. $T_{sum}(n)$ – adding list of numbers
3. $T_{mtx}(n)$ – adding two matrix

Asymptotic notation (O,Ω,Θ) - 점근표기법

Program add	step count

`float add(int a, int b) {`
`  return    a + b;`
`}`
	
	1

Program sum of list	step count

`float sum(float list[], int n) {`
`  float total = 0;`
`  int i;`
`  for (i=0; i<n; i++)`
`    total += list[i];`
`  return total;`
`}`
	
	1
	
	n + 1
	n
	1

Program sum of matrix	step count

`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],`
`         int c[][MAX_SIZE}, int rows, int cols) {`
`  for(int i=0; i<rows; i++)`
`    for(iny j=0; j<cols; j++)`
`      c[i][j] = a[i][j] + b[i][j];`
`}`
	
	rows + 1
	rows * (cols+1)
	rows * cols

Asymptotic notation (O,Ω,Θ) - 점금표기법

What is the exact number of times sum++ executed?

	Step count	

`int sum = 0;`
`for (int i = 1; i <= n*n; i++)`
`    for (int j = 1; j <= i; j++)`
`        sum++;`
	1
	n * n + 1
	2 + 3 + … + n*n+1
	?	

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Asymptotic notation (O,Ω,Θ) - 점금표기법

What is the exact number of times sum++ executed?

	Step count
`int sum = 0;`	1
`for (int i = 1; i <= n; i++)`	n + 1
`    for (int j = n; j >= i; j--)`	(n + 1) + n + (n-1) + … + 2
`        sum++;`	?

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Asymptotic notation (Ο,Ω,Θ) - 점금표기법

What is the exact number of times sum++ executed?

	Step count

`int sum = 0;`
`while (n > 1) {`
`    sum++;`
`    n /= 2;`
`}`
	n / 2^k = 1

We have to find the smallest k such that n / 2^k = 1

```
n / 2^k = 1
n = 2^k
log(n) = log(2^k)
log(n) = k
```

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Asymptotic notation (O,Ω,Θ) - 점금표기법

Compute the following series:

a) $1 + 2 + 3 + \ldots + 9 + 10 =$
b) $1 + 2 + 3 + \ldots + (N - 1) + N =$

c) $1 + 2 + 4 + \ldots + 16 =$

Compute the following series and express the result in term of N but without log expression. (Hint: $N = 2^{\log N}$)
Then use the result and to compute the series shown above in c):

d) $1 + 2 + 4 + \ldots + N =$

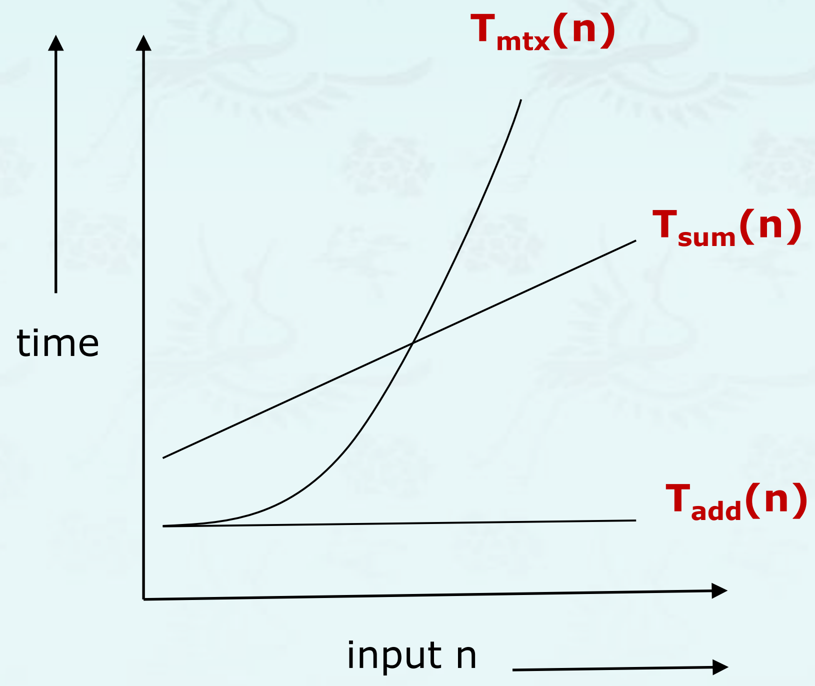
Useful formulas:
$1 + 2 + 3 + \ldots (N-1) + N = N(N+1)/2$
$1 + 2 + 4 + 8 + \ldots 2^{n-1} + 2^{n} = 2^{n+1} - 1$

Asymptotic notation (O,Ω,Θ) - 점금표기법

$$T_{add}(n) = 2$$ → O(1)

$$T_{sum}(n) = 1 + 2(n+1) + 2n + 1 = 4n + 4$$
$$= c * n + c'$$ → O(n)

$$T_{mtx}(n) = 2\,rows * cols + 2\,rows + 1$$
$$= a * n^2 + b * n + c$$ → O(n^2)

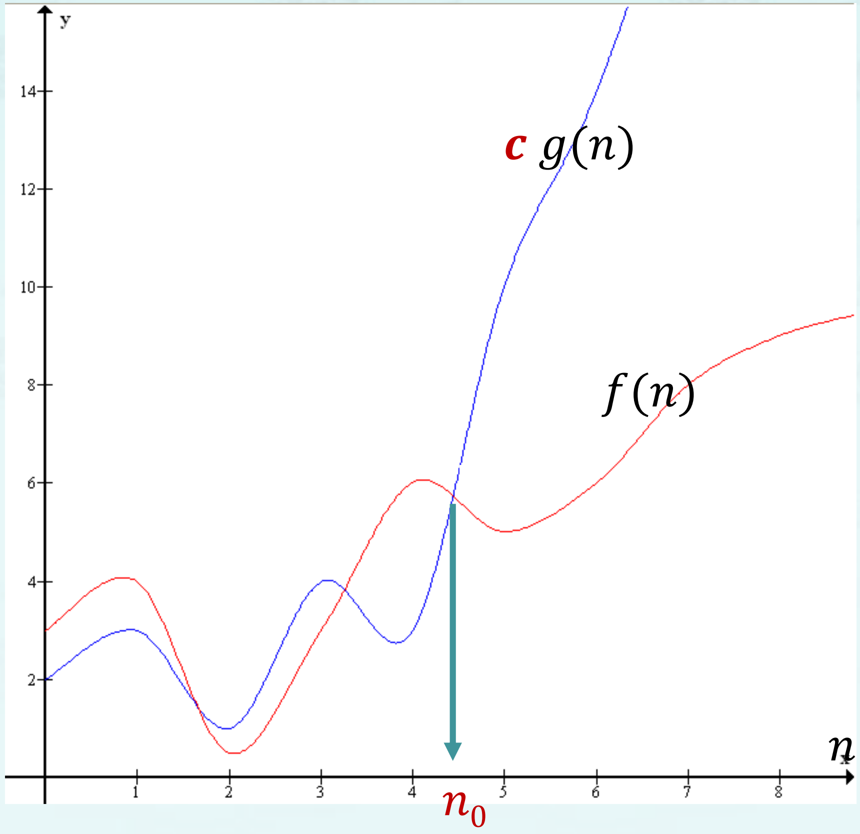


Asymptotic notation (O,Ω,Θ) - 점금표기법

The "Big-Oh" Notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that ***f(n) is O(g(n))*** iff there are positive constants **c** and n_0 such that

$$f(n) \le c\, g(n), for\ n \ge n_0.$$



f(n) is O(g(n))

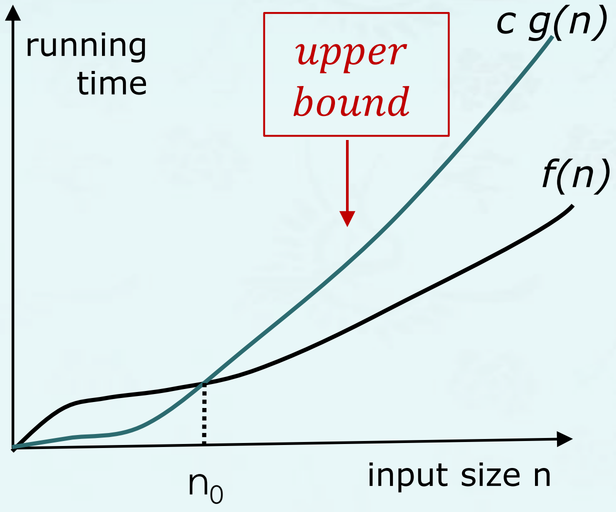
Asymptotic notation (O,Ω,Θ) - 점금표기법

The "Big-Oh" Notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers.
We say that ***f(n) is O(g(n))*** iff there are positive constants **c** and $\boldsymbol{n_0}$ such that

$$\boldsymbol{f(n)} \leq \boldsymbol{c}\, \boldsymbol{g(n)}, for\ n \geq \boldsymbol{n_0}.$$

Then it is pronounced as "$f(n)$ **is** $big\ Oh\ of\ g(n)\ or\ \boldsymbol{f(n)} = \boldsymbol{O(g(n))}$"



Example: Justify that the function $\mathbf{8}\boldsymbol{n} - \mathbf{2}$ **is** $\boldsymbol{O(n)}$.

Given $f(n) = 8n - 2, g(n) = n$,
we need to find c and n_0 such that
$8n - 2 \leq c\, n$ for every integer $n \geq n_0$.

An easy choice among many is $c = 8$ and $n_0 = 1$.
Therefore, $f(n) = 8n - 2\ is\ O(n)$.

$g(n) = n$

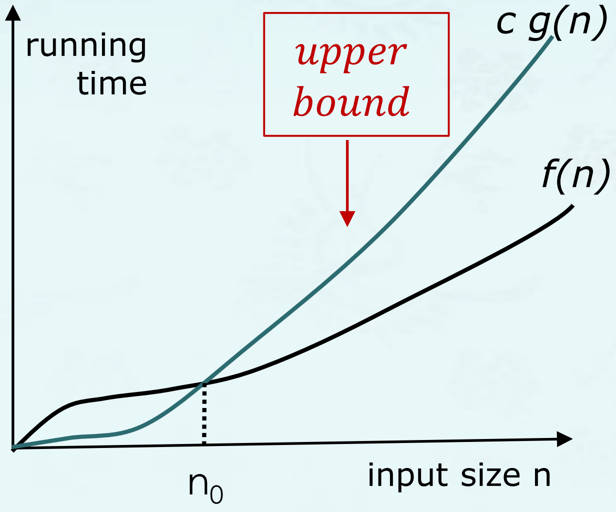
Asymptotic notation (O,Ω,Θ) - 점금표기법

The "Big-Oh" Notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that ***f(n) is O(g(n))*** iff there are positive constants **c** and n_0 such that

$$\boldsymbol{f(n)} \le \boldsymbol{c}\ \boldsymbol{g(n)}, for\ n \ge \boldsymbol{n_0}.$$

Then it is pronounced as "$f(n)$ **is** $big\ Oh\ of\ g(n)\ or\ \boldsymbol{f(n)} = \boldsymbol{O(g(n))}$"



Find $\boldsymbol{c}$ and $\boldsymbol{n_0}$ to justify that the function $7\boldsymbol{n} + 5$ **is** $\boldsymbol{O(n)}$.

7n + 5 is O(n), we have to find c and $\boldsymbol{n_0}$ such that

$7n + 5 \le c\ n\ for\ n \ge \boldsymbol{n_0}$

$7n + 5 \le 7\ n + n$

$7n + 5 \le 8\ n,\ for\ n \ge \boldsymbol{n_0} = 5$

Therefore, $7n + 5 \le c\ n$ for $c = 8$ and $\boldsymbol{n_0} = 5$

Asymptotic notation (O,Ω,Θ) - 점금표기법

Examples:

1) $3n + 2 =$
2) $3n + 3 =$
3) $100n + 6 =$
4) $10n^2 + 4n + 2 =$
5) $6 * 2^n + n^2 =$
6) $3n + 3 =$
7) $10n^2 + 4n + 2 =$
8) $3n + 2 \neq O(1)$
9) $10n^2 + 4n + 2 \neq O(n)$

Asymptotic notation (O, Ω, Θ) - 점금표기법

Preferred Big-Oh usage:

- **Pick the tightest bound.** If $f(N) = 5N$, then:

  $f(N) = O(N^5)$
  $f(N) = O(N^3)$
  $f(N) = O(N \log N)$
  $\mathbf{f(N) = O(N)}$ $\leftarrow$ preferred or right!

- **Ignore constant factors and low order terms:**

  $f(N) = \mathbf{O(N)}$, *not* $f(N) = O(5N)$
  $f(N) = \mathbf{O(N^3)}$, *not* $f(N) = O(N^3 + N^2 + 15)$

  - Wrong: $f(N) \leq O(g(N))$
  - Wrong: $f(N) \geq O(g(N))$
  - Right: $\mathbf{f(N) = O(g(N))}$

Asymptotic notation (O,Ω,Θ) - 점금표기법

Suppose two algorithms, A and B, solving the same problem have the running time of $O(n)$ and $O(n^2)$, respectively.
Then algorithm A is asymptotically better than algorithm B.

※ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

constant	logarithmic	linear	linearithmic	quadratic	cubic	exponential
정수함수	대수	선형함수	선형대수 loglinear	2/3승함수	2/3승함수	지수함수

여기서부터 boundary
여기서부터 사용되는 코드는 조금 쉽지 않음

Asymptotic notation (O,Ω,Θ) - 점금표기법

[Omega] ***f(n) = Ω (g(n))*** iff there exist positive constants c and n_0 such that

$$f(n) \geq c\, g(n), for\, n \geq n_0.$$

Example: Let's suppose we have

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

For all $n \geq 0$, this $(2n + 1)$ will be ≥ to 1, **if** we have $c = 5$ and $n_0 = 0$.

Then, $5\, n^2 \leq f(n)$, for all $n \geq 0$

Therefore, we can say that the time complexity of $f(n)$ is $\Omega\, (\boldsymbol{n^2})$;

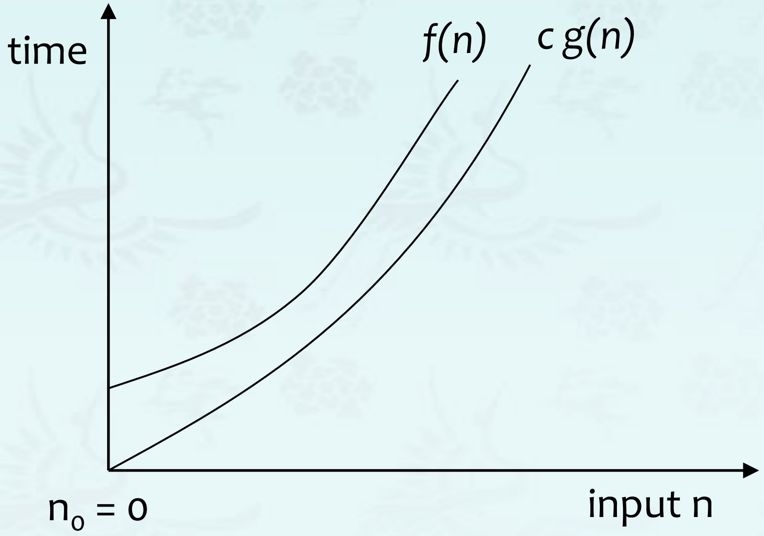
Asymptotic notation (O,Ω,Θ) - 점금표기법

[Omega] ***f(n) = Ω (g(n))*** iff there exist positive constants c and n_0 such that

$$f(n) \geq c\, g(n), for\ n \geq n_0.$$

Example: Let's suppose we have

$$f(n) = 5n^2 + 2n + 1$$
$$g(n) = n^2$$



❖ **Omega** notation gives us the **lower bound** of the growth rate of a function.

Asymptotic notation (O,Ω,Θ) - 점근표기법

[Omega] ***f(n) = Ω (g(n))*** iff there exist positive constants c and n_0 such that

$$f(n) \geq c\, g(n), for\ n \geq n_0.$$

Example:

1) $3n+2 = \Omega(n)$ *since* $\mathbf{3n+2} \geq \mathbf{3n}$ *for* $n \geq 1$

2) $3n+3 = \Omega(n)$ *since* $3n+3 \geq 3n$ *for* $n \geq 1$

3) $100n+6 = \Omega(n)$ *since* $100n+6 \geq 100n$ *for* $n \geq 1$

4) $100n^2+4n+2 = \Omega(n^2)$ *since* $100n^2+4n+2 \geq n^2$ *for* $n \geq 1$

5) $6*2^n+n^2 = \Omega(2^n)$ *since* $6*2^n+n^2 \geq 2^n$ *for* $n \geq 1$

Asymptotic notation (O,Ω,Θ) - 점금표기법

[Theta] *f(n)* = Θ (*g(n)*) iff there exist positive constants c and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example: Let's suppose we have

$$f(n) = 5n^2 + 2n + 1$$
$$g(n) = n^2$$

Then, we can choose $c_1 = 5, c_2 = 8$, and $n_0 = 1$; and our inequality will hold. Therefore we can say that the time complexity of

$$f(n) = 5n^2 + 2n + 1 = \Theta\,(n^2)$$

Asymptotic notation (O,Ω,Θ) - 점금표기법

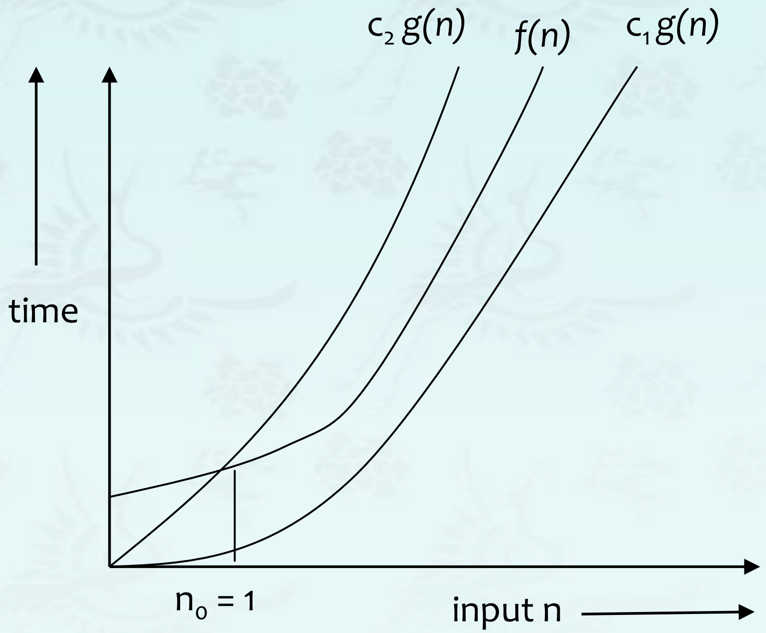
[Theta] ***f(n) = Θ (g(n))*** iff there exist positive constants c and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example: Let's suppose we have

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$



❖ **Θ notation** best describes or give the best idea about the growth rate of the function because it gives us a **tight bound** unlike ***O* and *Ω*** which give us **upper bound** and **lower bound**, respectively.

Asymptotic notation (O,Ω,Θ) - 점금표기법

[Theta] ***f(n)* = Θ *(g(n))*** iff there exist positive constants c and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example:

1) $3n + 2 = \Theta(n)$

since $\mathbf{3n} \le \mathbf{3n+2} \le \mathbf{4n}$ $for\ all\ n \ge 2, c_1 = 3, c_2 = 4, and\ n_0 = 2$

2) $3n + 3 = \Theta(n)$

3) $10n^2 + 4n + 2 = \Theta(n^2)$

4) $6 * 2^n + n^2 = \Theta(2^n)$

5) $10 * \log n + 4 = \Theta(\log n)$

Performance Analysis – Linear search

The time complexity of the linear search:

- **Best Case: Find at first place - one comparison**
- **Worst Case: Find at nth place or not at all - n comparisons**
- **Average Case: It is shown below that this case takes - (n+1)/2 comparisons**

- In considering the average case there are n cases that can occur, i.e. find at the first place, the second place, the third place and so on up to the *n*th place. If found at the *i*th place then *i* comparisons are required. Hence the average number of comparisons over these n cases is:

  average = (1 + 2 + 3 … + n) / n
  = n(n + 1)/2 / n,
  since (1 + 2 + 3 + … + n) is equal to n(n + 1)/2.

Hence linear search is $an\ order(n)\ process\ or\ T(n) = O(n)$.

Recurrence Relations

Recurrence Relations is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given: each further term of the sequence or array is defined as a function of the preceding terms.

For example:

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

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Performance Analysis – Linear search

We may describe that the time complexity of the linear search is

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

Recurrence equation

- **The cost of searching n elements is the cost of looking at 1 element, plus the cost of searching n – 1 elements.**
- Let's "**telescoping**" a few of these:

$$T(n) = T(n-1) + c$$
$$T(n-1) = T(n-2) + c$$
$$T(n-2) = T(n-3) + c$$
$$\ldots$$
$$T(2) = T(1) + c$$

- Then add each side,

$$T(n) = T(1) + (n-1)c$$
$$T(n) = c + nc - c$$
$$T(n) = O(n)$$

Performance Analysis – Selection sort

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

```
public static void selectionSort(int[]a) {        // Java syntax
  int min;
  for (int i = 0; i < a.length-1; i++)
    min = i;
    for (int j = i+1; j < a.length; j++)
      if (a[j] < a[min])
        min = j;
    swap(a[i], a[min]);     // exchange a[i] with a[min] found
  }
}
```

Performance Analysis – Selection sort

$$T(1) = 1$$

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

Recurrence equation

- **Unfolding** makes repeated substitutions applying the recursive rule until the base case is reached.

  Substitute n-1 everywhere we see an n in the recurrence relation:

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

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

Making this substitution one more time we get

$$T(n) = n + (n-1) + (n-2) + T(n-3)$$

We repeat this process until we reaches T(1), base case

$$T(n) = n + (n-1) + \ldots + (n-(n-2)) + \boxed{}$$

$$T(n) = n + (n-1) + \ldots + 2 + \boxed{}$$

$$= n + (n-1) + \ldots + 2 + 1$$

$$= \frac{n(n+1)}{2}$$

$$= O(n^2)$$

Performance Analysis – Selection sort

$$T(1) = 1$$

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

Recurrence equation

- **Unfolding** makes repeated substitutions applying the recursive rule until the base case is reached.

  Substitute n-1 everywhere we see an n in the recurrence relation:

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

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

Making this substitution one more time we get

$$T(n) = n + (n-1) + (n-2) + T(n-3)$$

We repeat this process until we reaches T(1), base case

$$T(n) = n + (n-1) + \dots + (n-(n-2)) + T(n-(n-1))$$

$$T(n) = n + (n-1) + \dots + 2 + T(1)$$

$$= n + (n-1) + \dots + 2 + 1$$

$$= \frac{n(n+1)}{2}$$

$$= O(n^2)$$

Performance Analysis – Selection sort

$T(1) = 1$

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

- **Telescoping**

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
T(n-1) &= n - 1 + T(n-2) \\
T(n-2) &= n - 2 + T(n-3) \\
&\cdots \\
T(2) &= 2 + T(1)
\end{aligned}
$$

- **Add all terms in each side and cancel the equal terms, then it becomes**

$$
\begin{aligned}
T(n) &= n + (n-1) + \ldots + 2 + T(1) \\
&= \frac{n(n+1)}{2} - 1 + T(1) \\
&= O(n^2)
\end{aligned}
$$

Performance Analysis – Binary search

Recurrence equation

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 +$) where n is $hi - lo$

- $O(\log n)$ where n is $array.length$
- Solve *recurrence equation* to know that…

```
// returns whether k is in array, array a is sorted
boolean binarySearch(int *a, int k, size){
   return _binarySearch(a,k,0,size-1);
}

boolean _binarySearch(int *a, int k, int lo, int hi) {
   int mid = (hi+lo)/2;
   if (lo==hi)     return false;
   if (a[mid]==k) return true;
   if (a[mid]< k) return _binarySearch(a,k,mid+1,hi);
   else            return _binarySearch(a,k,lo,mid-1);
}
```

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 + T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T(n/2)$$

telescoping

$$T(2) = 1 + T(1)$$

2. Sum up the left and right sides of the equations above:

$$T(n) += (\_\_\_\_\_\_\_\_\_\_\_\_) + T(1)$$

3. Cross out the equal terms to simplify. How many 1's on the right side?

$$T(n) =$$

$$=$$

Therefore the time complexity of binary search is $T(n)$ is $O(\log n)$

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 + T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

telescoping

$$
\begin{aligned}
T(n) &= 1 + T(n/2) \\
T(n/2) &= 1 + T(n/4) \\
T(n/4) &= 1 + T(n/8) \\
&\dots \\
T(4) &= 1 + T(2) \\
T(2) &= 1 + T(1)
\end{aligned}
$$

2. Sum up the left and right sides of the equations above:

$$T(n) += (\underline{\hspace{4cm}}) + T(1)$$

3. Cross out the equal terms to simplify. How many 1's on the right side?

$$T(n) =$$

$$=$$

Therefore the time complexity of binary search is $T(n)$ is $O(\log n)$

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) \ = 1 \ + \ T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 \ + \ T(n/2)$$
$$T(n/2) = 1 + T(n/4)$$
$$T(n/4) = 1 + T(n/8)$$
$$\ldots$$
$$T(4) = 1 + T(2)$$
$$T(2) = 1 + T(1)$$

telescoping

2. Sum up the left and right sides of the equations above:

$$T(n) += (1 + 1 +..+1) + \ T(1)$$

3. Cross out the equal terms to simplify. How many 1's on the right side?

$$T(n) =$$
$$=$$

Therefore the time complexity of binary search is $T(n)$ is $O(\log n)$

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 + T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

telescoping →

$$T(n) = 1 + T(n/2)$$
$$T(n/2) = 1 + T(n/4)$$
$$T(n/4) = 1 + T(n/8)$$
$$\ldots$$
$$T(4) = 1 + T(2)$$
$$T(2) = 1 + T(1)$$

2. Sum up the left and right sides of the equations above:

$$T(n) += (1 + 1+..+1) + T(1)$$

3. Cross out the equal terms to simplify. How many 1's on the right side?

$$T(n) = \log_2 n + T(1)$$
$$= \log_2 n + 1$$

Therefore the time complexity of binary search is $T(n)$ is $O(\log n)$

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 + T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T\left(\frac{n}{2}\right) \qquad\qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions*.

$$\begin{aligned} T(n) &= 1 + 1 + T(n/4) \\ &= 1 + 1 + 1 + T(n/8) \\ &= 1 + 1 + 1 + 1 + T(n/16) \\ &= 1 + \ \dots + 1 + T(n/n) \end{aligned}$$

How many 1's here?

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) = 1 + T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T\left(\frac{n}{2}\right) \qquad\qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions*.

$$\begin{aligned} T(n) &= 1 + 1 + T(n/4) && 2^2 \\ &= 1 + 1 + 1 + T(n/8) && 2^3 \\ &= 1 + 1 + 1 + 1 + T(n/8) && 2^4 \\ &= 1 + \ldots + 1 + T(n/n) && 2^n \\ &= \mathbf{1}k + T\left(\frac{\boldsymbol{n}}{2^k}\right) && \mathbf{2^k} \end{aligned}$$

number of 1's

Performance Analysis – Binary search

Base case: $\mathrm{T}(1) = O(1) = 1$

Recurrence: Let suppose that $T(n) \ = 1 \ + \ T(\frac{n}{2})$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 \ + \ T\left(\frac{n}{2}\right) \qquad\qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions*.

$$\begin{aligned} T(n) \ &= 1 \ + 1 \ + \ T(n/4) \\ &= 1 \ + 1 \ + 1 \ + \ T(n/8) \\ &= 1 + \ \ldots \ + 1 + T(n/n) \\ &= \mathbf{1}k \ + \ T\left(\frac{\boldsymbol{n}}{2^k}\right) \end{aligned}$$

Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case

$n/(2^k) \ = \ 1$ means $n \ = \ 2^k \rightarrow \ k \ = \ \log_2 n$

So $T(n) \ = 1 \log_2 n \ + 1$ (get to base case and do it)

So $T(n)$ is $O(\log n)$

Asymptotic notation (*O*,*Ω*,*Θ*) - 점금표기법

Asymptotic Analysis:

Suppose that two algorithms, A and B, solving the same problem have the running time of O(n) and O(n^2), respectively. Then this implies that algorithm A is **asymptotically better** than algorithm B.

We can use the **big-Oh** notation to order classes of functions by **asymptotic growth rate**.

Seven functions below are often used and ordered by increasing growth rate.

※ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

n	log n	n	n log n	n^2	n^3	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84 x 10^19
128	7	128	896	16,384	2,097,152	3.40 x 10^38
256	8	256	2,048	65,536	16,777,216	1.15 x 10^77

※ Even if we achieve a dramatic speed-up in hardware, we still cannot overcome the handicap of an asymptotically slow program.

Asymptotic notation (O, Ω, Θ) - 점금표기법

Example: Running time estimates - empirical analysis

- Laptop executes 10^8 compares/second
- Supercomputer executes 10^{12} compares/second

use a reasonable time unit

	Insertion sort (N^2)			Merge sort (N $\log_2$ N)		
N	Thousand	Million	Billion	Thousand	Million	Billion
Laptop	Instant	2.8 hours		Instant	1 sec	
Super Com	Instant	1 sec		Instant	Instant	Instant

$$log_{10}2 \cong 0.3$$
$$86{,}400\text{sec}/day$$

※ **Bottom line:** Good algorithms are better than supercomputers.

Data Structures

- *performance analysis - time complexity*