Order

Most Common Complexity Categories

- $\Theta(\lg n)$
- $\Theta(n)$: linear
- $\Theta(n \lg n)$
- $\Theta(n^2)$: quadratic
- $\Theta(n^3)$: cubic
- $\Theta(2^n)$: exponential
- $\Theta(n!)$
- $\Theta(n^n)$

The quadratic term eventually dominates.

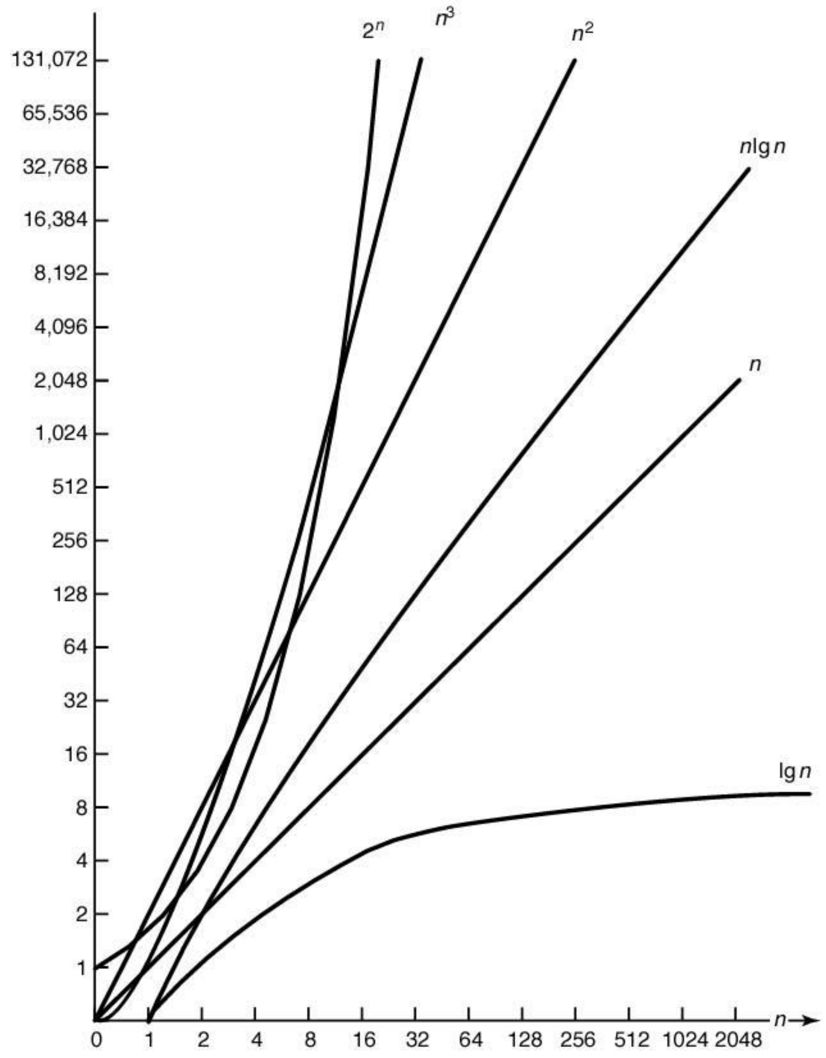
n	$0.1n^2$	$0.1n^2+n+100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

The higher order term eventually dominates.

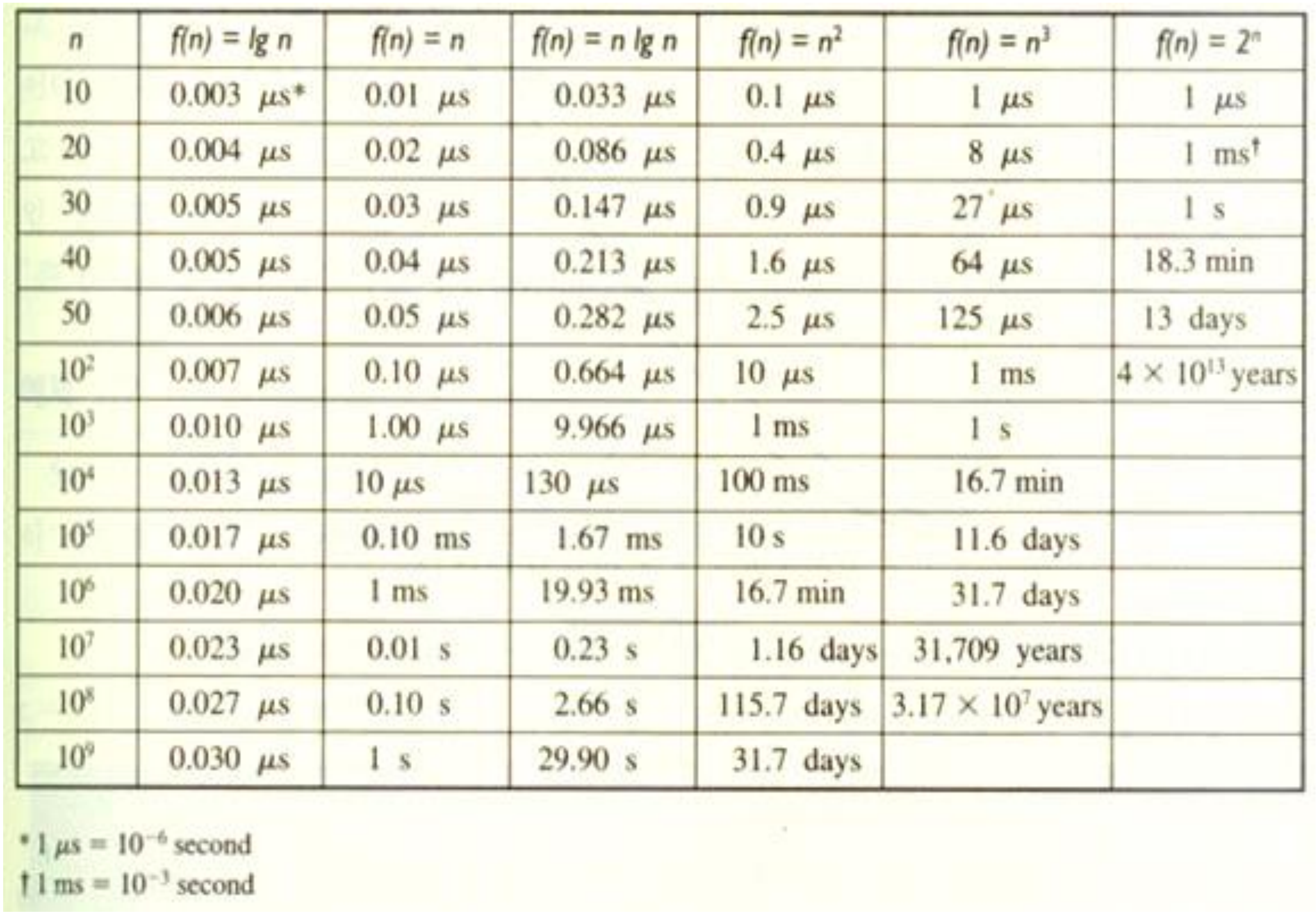
n	$0.01n^2$	$100n$
10	1	1,000
100	100	10,000
1,000	10,000	100,000
10,000	1,000,000	1,000,000
100,000	100,000,000	10,000,000

For $n > 10{,}000$, $0.01n^2 > 100n$.

Growth rates of some common complexity functions



Execution times for algorithms with the given time complexities



n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 days	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 days		

* 1 μs = 10^{-6} second

† 1 ms = 10^{-3} second

Asymptotic Behavior

- asymptotic behavior of $f(n)$ is behavior of $f(n)$ for a large value of n.
- (ex) $f(n) = 1/n$

$$\lim_{n \to \infty} \frac{1}{n} = 0$$

Notations

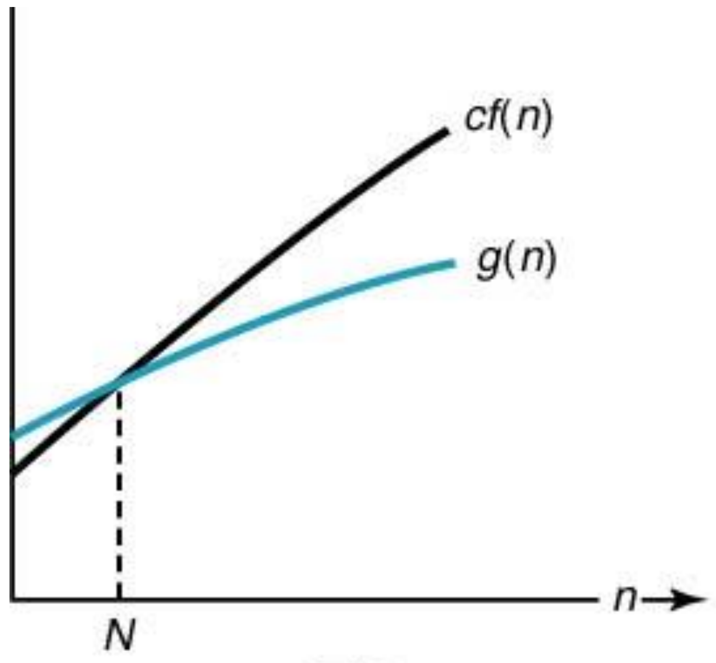
- O( ) - big oh: asymptotic upper bound
- o( ) - small oh: upper bound that is not asymptotically tight
- Ω( ) - omega: asymptotic lower bound
- ω( ) - small omega: lower bound that is not asymptotically tight
- Θ( ) - theta: asymptotic tight bound

Big O Notation

- **Definition : asymptotic upper bound**
  - ✓ For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ ($\in O(f(n))$) for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$, $0 \leq g(n) \leq c \times f(n)$.
  - ✓ $O(f(n)) = \{g(n)$: there exists positive constants c and N such that $0 \leq g(n) \leq c \times f(n)$ for all $n \geq N\}$

- $g(n) \in O(f(n))$ is said that
  $g(n)$ is *big* O of $f(n)$.

Big O Notation



(a) $g(n) \in O(f(n))$

Big O Notation

- If $g(n)$ is in $O(n^2)$,
  - ✓ $g(n)$ lies beneath some pure quadratic function cn^2 on a graph as n increases(i.e, for all $n \geq N$).
  - ✓ this means that if $g(n)$ is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast as quadratic.

- If $g(n)$ is in $O(f(n))$,
  - ✓ eventually $g(n)$ is at least as good as $f(n)$.
  - ✓ $f(n)$ is asymptotic upper bound.
  - ✓ $g(n)$ is at least as fast as $f(n)$(never slower than $f(n)$).

Big O Notation Examples

- $n^2+10n \in O(n^2)$?

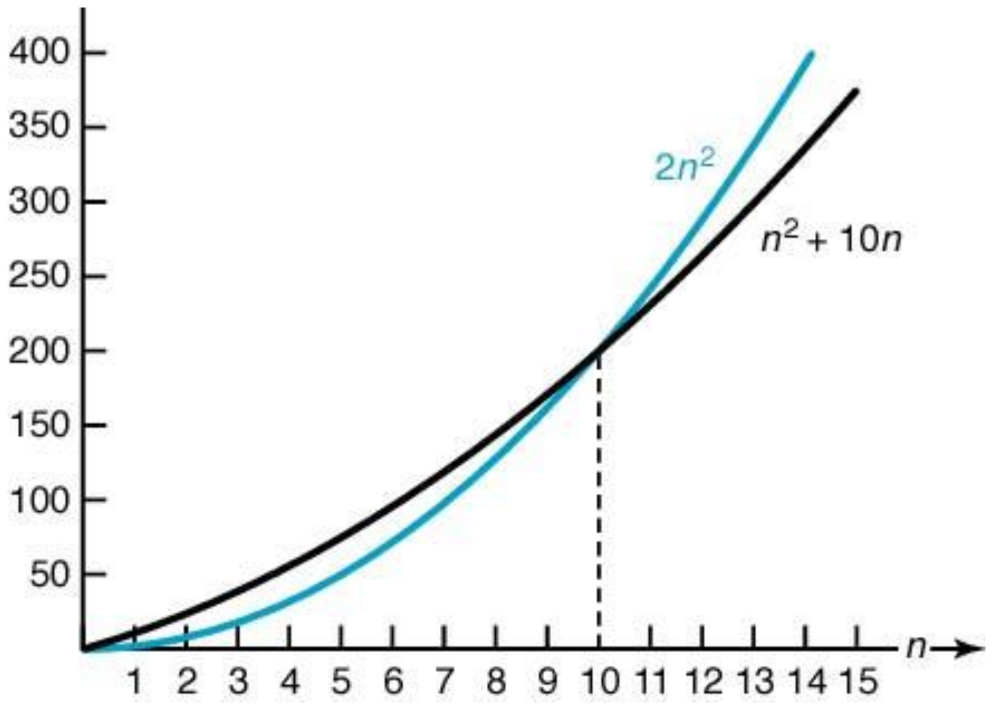
  (1) For $n \geq 10$, $n^2+10n \leq 2n^2$.

  Therefore, we can take $c = 2$ and $N = 10$ in the definition of "big O ", $n^2+10n \in O(n^2)$.

  (2) For $n \geq 1$, $n^2+10n \leq n^2+10n^2 = 11n^2$.

  Therefore, if we select $c = 11$ and $N = 1$, according to the definition of "big O ", $n^2+10n \in O(n^2)$.

$n^2 + 10$ eventually stays beneath $2n^2$



- $5n^2 \in O(n^2)$?

  If we select c=5 and N=0, for all $n \geq 0$, $5n^2 \leq 5n^2$.

- $T(n) = \frac{n(n-1)}{2}$?

  For all $n \geq 0$, $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$.

  Hence, we select $c = \frac{1}{2}$ and N=0, $T(n) \in O(n^2)$.

- $n^2 \in O(n^2+10n)$?

  For all $n \geq 0$, $n^2 \leq 1\times (n^2+10n)$. Therefore, for c=1 and N=0, $n^2 \in O(n^2+10n)$.

- $n \in O(n^2)$?

  For all $n \geq 1$, $n \leq 1 \times n^2$.

  Therefore, for c=1 and N=1, $n \in O(n^2)$.

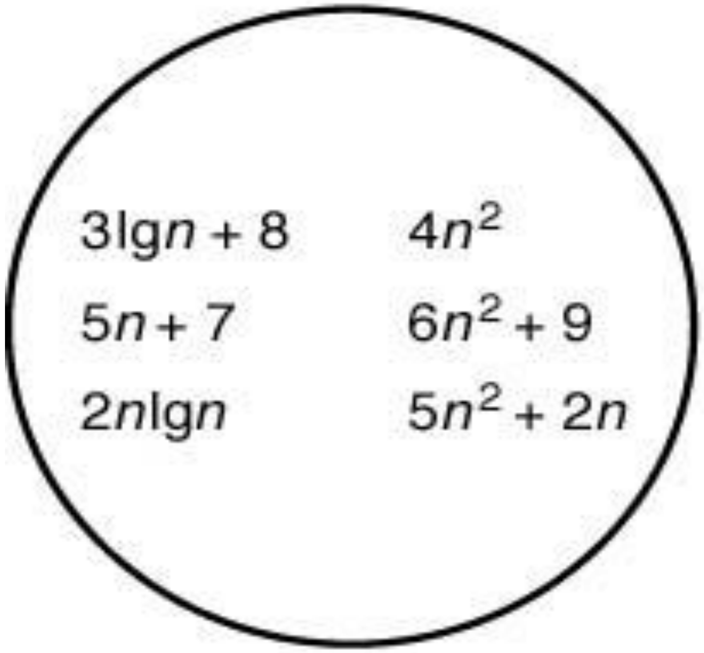
- $n^3 \in O(n^2)$?

  For all $n \geq N$, we can not select c and N such that $n^3 \leq c \times n^2$.

  i.e, if we divide both sides of the inequality by n^2, $n \leq c$.

  Even if we take a large value of c, there always exists a value of n larger than c.

$O(n^2)$



(a) $O(n^2)$

Ω Notation

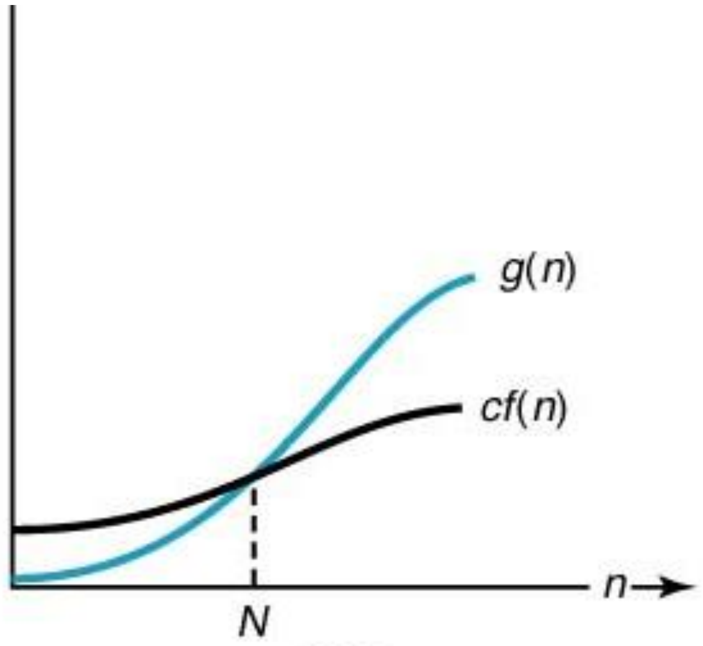
- **Definition : asymptotic lower bound**
  - ✓ For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ ($\in \Omega(f(n))$) for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$, $g(n) \geq c \times f(n)$.
  - ✓ $\Omega(f(n)) = \{g(n)$: there exists positive constants c and N such that

    $g(n) \geq c \times f(n) \geq 0$ for all $n \geq N\}$

- $g(n) \in \Omega(f(n))$ is said that

  $g(n)$ is omega of $f(n)$.

Ω Notation



(b) $g(n) \in \Omega(f(n))$

Ω Notation

- If $g(n)$ is in $\Omega(n^2)$,
  - ✓ $g(n)$ lies above some pure quadratic function cn^2 on a graph as n increases(i.e, for all $n \geq N$).
  - ✓ this means that if $g(n)$ is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as bad as quadratic.

- If $g(n)$ is in $\Omega(f(n))$,
  - ✓ eventually $g(n)$ is at least as bad as $f(n)$.
  - ✓ $f(n)$ is asymptotic lower bound.
  - ✓ $g(n)$ is at least as slow as $f(n)$(never faster than $f(n)$).

Ω Notation Examples

- $n^2 + 10n \in \Omega(n^2)$?

  For all $n \geq 0$, $n^2 + 10n \geq n^2$.

  Therefore, if we select $c = 1$ and $N = 0$, $n^2 + 10n \in \Omega(n^2)$.

- $5n^2 \in \Omega(n^2)$?

  For all $n \geq 0$, $5n^2 \geq 1 \times n^2$.

  Therefore, for $c = 1$ and $N = 0$, $5n^2 \in \Omega(n^2)$.

- $T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$?

  For all $n \geq 2$, $n - 1 \geq \frac{n}{2}$.

  Therefore, for all $n \geq 2$, $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4} n^2$.

  For $c = \frac{1}{4}$ and $N = 2$, $T(n) \in \Omega\left(n^2\right)$

- $n^3 \in \Omega\left(n^2\right)$?

  For all $n \geq 1$, $n^3 \geq 1 \times n^2$.
  Therefore, for $c = 1$ and $N = 1$, $n^3 \in \Omega\left(n^2\right)$

- $n \in \Omega(n^2)$?

  proof by contradiction

  Assume that $n \in \Omega(n^2)$. For all $n \geq N$, there exists a real number $c > 0$ and a positive integer N such that $n \geq c \times n^2$.

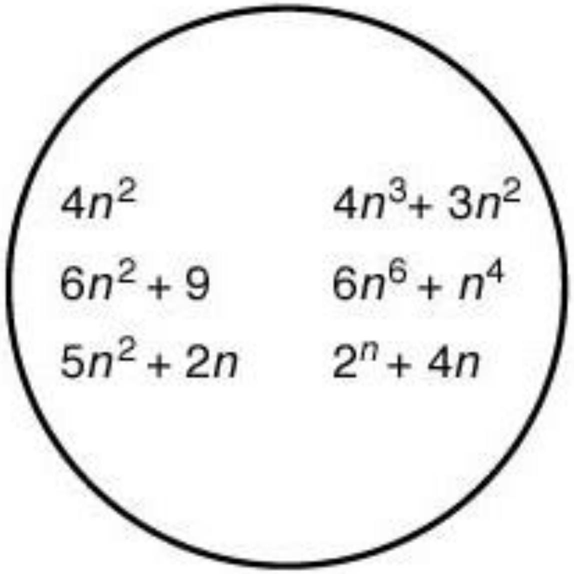
  If we divide both sides of this inequality by cn , $\frac{1}{c} \geq n$.

  But, this inequality is impossible since we assumed that ‘For all $n \geq N$’ and n must be less than $1/c$.

  Contradiction.

  Hence, $n \notin \Omega(n^2)$

$\Omega(n^2)$



(b) $\Omega(n^2)$

Θ Notation

- **Definition: asymptotic tight bound**
  - ✓ For a given complexity function $f(n)$, $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.
  - ✓ $\Theta(f(n))$ is the set of complexity function $g(n)$ for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$, $c \times f(n) \leq g(n) \leq d \times f(n)$.
  - ✓ $\Theta(f(n)) = \{g(n)$: there exist positive constants c, d, and N such that $c \times f(n) \leq g(n) \leq d \times f(n)$ for all $n \geq N\}$

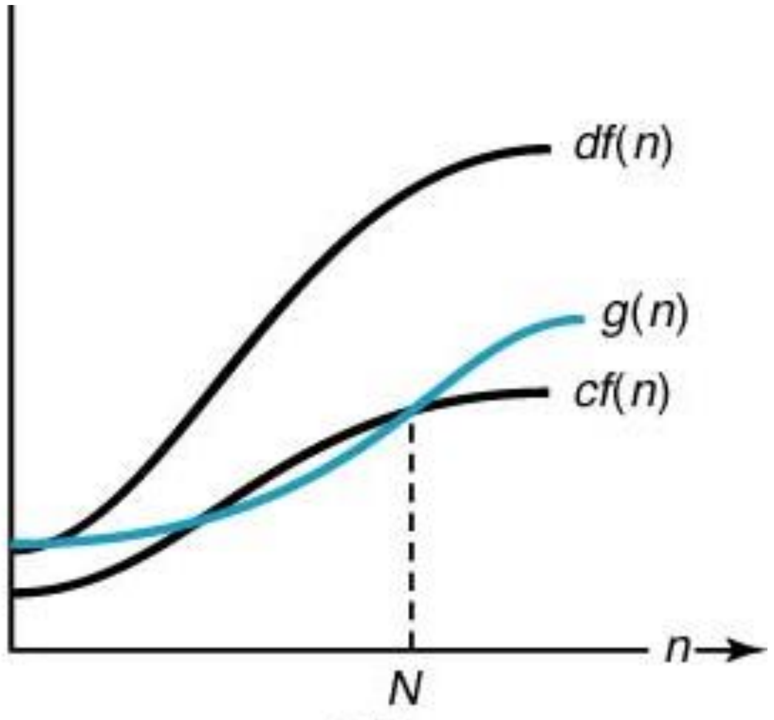
- If $g(n) \in \Theta(f(n))$, we say that "$g(n)$ is order of $f(n)$.

- Ex) : $T(n) = \dfrac{n(n-1)}{2}$ is in both $O(n^2)$ and $\Omega(n^2)$.

  Hence,

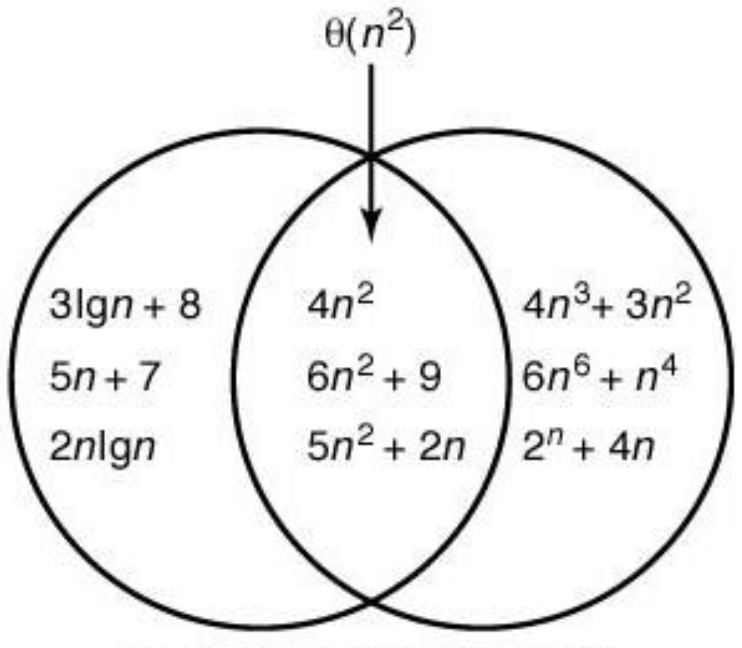
$$T(n) = \Theta(n^2)$$

Θ Notation



(c) $g(n) \in \theta(f(n))$

$\Theta(n^2)$



(c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

Small o Notation

- **Definition : small o**
  - ✓ For a given complexity function $f(n)$, o($f(n)$) is the set of complexity functions $g(n)$ ($\in$o($f(n)$)) satisfying the following:
  - ✓ For every positive real constant c there exists a nonnegative integer N such that for all $n \geq N$, $0 \leq g(n) \leq c \times f(n)$.
  - ✓ o($f(n)$) = {$g(n)$: For every $c>0$, there exists an N such that $0\leq g(n) \leq c \times f(n)$ for all $n \geq N$}

- If $g(n) \in o(f(n))$, "$g(n)$ is small o of $f(n)$.

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Big O vs. small o

- Difference
  - Big O - there must be <u>some</u> real positive constant c for which the bound holds.
  - Small o – the bound must hold for <u>every</u> real positive constant c.

- If $g(n) \in o(f(n))$, $g(n)$ is eventually much better(faster) than functions such as $f(n)$.

Small o Notation Examples

- $n \in o(n^2)$?

Proof:

Let $c>0$ be given. We need to find an N such that, for $n \geq N$,

$$n \leq cn^2 .$$

If we divide both sides of this inequality by cn, we get

$$1/c \leq n.$$

Therefore, it suffices to choose any $N \geq 1/c$.

Notice that the value of N depends on the constant c.

For example, if $c=0.0001$, we must take N equal to at least 10,000.

That is, for $n > 10{,}000$,

$$n \leq 0.0001n^2 .$$

- *n* is not in o(5*n*)?

Proof by contradiction:

Let c=1/6. If $n \in$ o(5n), then there must exist some N such that, for $n \geq N$, $n \leq 1/6 \times 5n = 5/6n$.

This contradiction proves that *n* is not in o(5*n*).

ω Notation

- By analogy, ω –notation is to Ω-notation as o-notation is to O-notation.
- It denotes a lower bound that is not asymptotically tight.
  - ✓ $g(n) \in \omega(f(n))$ if and only if $f(n) \in o(g(n))$
  - ✓ $\omega(f(n))$: small omega of f of n
  - ✓ $\omega(f(n))=\{ g(n)$: for any positive constant $c > 0$, there exists a constant $N > 0$ such that $0 \le c \times f(n) \le g(n)$ for all $n \ge N\}$
- $n^2/2 = \omega(n)$. But $n^2/2 \neq \omega(n^2)$.
- The relation $g(n)=\omega(f(n))$ implies that

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

Properties of Order

1. $g(n) \in O(f(n))$ if and only if $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ if and only if $f(n) \in \Theta(g(n))$

3. If $b > 1$ and $a > 1$, $\log_a n \in \Theta(\log_b n)$. This implies that all logarithmic complexity functions are in the same complexity category. We will represent this category by $\Theta(\lg n)$.

4. If $b > a > 0$, $a^n \in o(b^n)$. This implies that all exponential complexity functions are not in the same complexity category.

5. For all $a > 0$, $a^n \in o(n!)$. $n!$ is worse than any exponential complexity function.

6. Consider the following ordering.

$$\Theta(\lg n), \Theta(n), \Theta(n \lg n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$

where $k>j>2$ and $b>a>1$. If a complexity function $g(n)$ is in a category that is to the left of the category containing $f(n)$, then

$$g(n) \in o(f(n))$$

7. If $c \geq 0$, $d \geq 0$, $g(n) \in O(f(n))$, and $h(n) \in \Theta(f(n))$, then

$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

Using a limit to Determine Order

- Theorem 1.3

$$\lim_{n\to\infty}\frac{g(n)}{f(n)} = \begin{cases} c & \text{implies} \quad g(n)\in\Theta(f(n)) \text{ if } c>0 \\ 0 & \text{implies} \quad g(n)\in o(f(n)) \\ \infty & \text{implies} \quad f(n)\in o(g(n)) \end{cases}$$

- Ex 1.24 :

  Prove that $\frac{n^2}{2}\in o(n^3)$.

$$\lim_{n\to\infty}\frac{n^2/2}{n^3} = \lim_{n\to\infty}\frac{1}{2n} = 0$$

- Ex 1.25 : Prove that for $b>a>0$, $a^n \in o(b^n)$.

$$\lim_{n\to\infty}\frac{a^n}{b^n} = \lim_{n\to\infty}\left(\frac{a}{b}\right)^n = 0 \quad \text{because } 0<\frac{a}{b}<1$$

- Theorem 1.3 implies that, for $a > 0$,

$$a^n \in o(n!)$$

(1) if $a \leq 1$, it is trivial.

(2) if $a > 1$, for a large n, $\left\lceil \frac{n}{2} \right\rceil > a^4$.

$$n! > \overbrace{\left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{2} \right\rceil \cdots \left\lceil \frac{n}{2} \right\rceil}^{n/2}$$

$$> a^4 a^4 \cdots a^4$$

$$\therefore \frac{a^n}{n!} < \frac{a^n}{a^4 a^4 \cdots a^4} = \frac{a^n}{(a^4)^{n/2}} = \frac{a^n}{a^{2n}} = (\frac{1}{a})^n$$

$$\lim_{n\to\infty} \frac{a^n}{n!} = 0$$

Theorem 1.4 : L'Hopital's Rule

If $f(x)$ and $g(x)$ are both differentiable with derivatives $f'(x)$ and $g'(x)$, respectively, and if

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = \infty$$

then

$$\lim_{x\to\infty} \frac{g(x)}{f(x)} = \lim_{x\to\infty}\left(\frac{g'(x)}{f'(x)}\right) .$$

[Ex 1.27]

$$\lg n \in o(n) \qquad \lim_{n\to\infty}\frac{\lg n}{n} = \lim_{n\to\infty}\left(\frac{\frac{1}{n\ln 2}}{1}\right) = 0$$

[Ex 1.28]

$$\log_a n \in \Theta(\log_b n)$$

$$\lim_{n\to\infty}\frac{\log_a n}{\log_b n} = \lim_{n\to\infty}\left(\frac{\frac{1}{n\ln a}}{\frac{1}{n\ln b}}\right) = \frac{\log b}{\log a} > 0$$

알고리즘 복잡도와 컴퓨터 능력

- 알고리즘의 복잡도가 (A)일 때 t시간 동안
  - ✓ 현재의 기계가 문제크기 m개의 문제 해결
  - ✓ 기계의 처리 속도가 <u>(B)배</u> 된다면 문제크기 <u>(C)</u>개의 문제를 t시간에 해결할 수 있다.

A	**B**	**C**
O(n)	2	2m
$O(n^2)$	4	2m
$O(n^3)$	8	2m
$O(2^n)$	2	m+1
$O(2^n)$	100	?

Ch 2. Divide-and-Conquer

Textbook: Foundations of Algorithms Using C++ Pseudocode by R. Neapolitan and K. Naimipour
This is prepared only for the class.

Design Strategy of Divide-and-Conquer

- Divide an instance of a problem into two or more smaller instances. The smaller instances are usually instances of the original problems.
- Conquer(solve) each of the smaller instances.
- If necessary, combine the solutions to the smaller instances to obtain the solution to the original instance.

- top-down approach

Binary Search(recursive)

- problem: Determine whether *x* is in the sorted array *S* of size *n*.
- inputs: positive integer *n*, sorted(nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*
- outputs: *location* - the location of *x* in *S(*o if *x* is not in *S)*
- design strategy:
  - ✓ If *x* equals the middle item, quit. Otherwise,
  - ✓ Divide the array into two subarrays about half as large. If *x* is smaller than the middle item, choose the left subarray. If *x* is larger than the middle item, choose the right subarray.
  - ✓ Conquer(solve) the subarray by determining whether *x* is in that subarray. Unless the subarray is sufficient small, use recursion to do this.

```
index location (index low, index high) {
   index mid;

   if (low > high)
       return 0;                                   // not found
   else {
       mid = (low + high) / 2    // division by an integer
       if (x == S[mid])
        return mid;                      //  found
       else if (x < S[mid])
        return location(low, mid-1);  // select left half
       else
        return location(mid+1, high) ;// select right half
   }
}
…
locationout = location(1, n);
...
```

Discussion

1. *n*, *S*, *x* are not parameters to function *location*. Because they remain unchanged in each recursive call, there is no need to make them parameters.
2. Only the variables, whose values can change in the recursive calls, are made parameters to recursive calls.
3. Make *n*, *S*, *x* global variables.

4. No operations are done after the recursive all.

    - tail recursion

    - easy to produce an iterative version

5. Recursion clearly illustrates the divide-and-conquer process of dividing an instance into smaller instances.

6. advantageous to replace tail-recursion by iteration.

    - ✓ a substantial amount of memory can be saved by eliminating the stack developed in the recursive calls.
    - ✓ when a routine calls another routine it is necessary to save the first routine's pending results by pushing them onto the stack of activation records.
    - ✓ iterative algorithm will execute faster (but only by a constant multiplicative factor) than the recursive version because no stack needs to be maintained.

Worst Case Time Complexity Analysis of Binary Search(recursive)

- **basic operation**: the comparison of *x* with *S*[mid].
- **input size**: *n*, the number of items in the array.(*high*- *low* + 1)
- There are two comparisons of *x* with *S*[*mid*] in any call to function location in which *x* does not equal to *S*[*mid*].
  - ✓ we can assume that there is only one comparison, because
    - (1) this would be the case in an efficient assembler language implementation.
    - (2) we ordinarily assume that the basic operation is implemented as efficiently as possible.

- **case 1: *n* is a power of 2.**

$$W(n) = W(n/2) + 1 \,,\ n > 1 \,,\ n = 2^k, \text{ for some } k \geq 1$$
$$W(1) = 1$$

We can obtain the following:

$$\begin{aligned}
&W(1) = 1\\
&W(2) = W(1) + 1 = 2\\
&W(4) = W(2) + 1 = 3\\
&W(8) = W(4) + 1 = 4\\
&W(16) = W(8) + 1 = 5\\
&\cdots\\
&W(2^k) = k + 1\\
&\cdots\\
&W(n) = \lg n + 1
\end{aligned}$$

iterative substitution method or iteration method

$$\boxed{\begin{aligned}
W(n) &= W(n/2) + 1\\
&= (W(n/2^2) + 1) + 1\\
&= W(n/2^2) + 2\\
&= \bullet\bullet\\
&= \bullet\bullet\\
&= W(n/2^k) + k \,, (n = 2^k \text{가정})\\
&= 1 + k\\
&= \lg n + 1
\end{aligned}}$$

Substitution Method

Use mathematical induction

1. induction basis : For $n = 1$, $W(1) = 1 = \lg 1 + 1$.
2. induction hypothesis : For n a power of 2, assume that $W(n) = \lg n + 1$.
3. induction step : We have to prove that $W(2n) = \lg(2n) + 1$.

By the recursion,

$$
\begin{aligned}
W(2n) &= W(n) + 1 && (\textit{by}\ \text{recursion}) \\
&= \lg n + 1 + 1 && (\textit{by}\ \text{assumption}) \\
&= \lg n + \lg 2 + 1 \\
&= \lg(2n) + 1
\end{aligned}
$$

- **case 2: general case - the size of one subinstance will be $\left\lfloor \frac{n}{2} \right\rfloor$.**

$\lfloor y \rfloor$ means the greatest integer less than or equal to *y*.

For any *n*, $mid = \left\lfloor \frac{1+n}{2} \right\rfloor$

n	Size of left half	mid	Size of right half
even	*n*/2 - 1	1	*n*/2
odd	(*n*-1)/2	1	(*n*-1)/2

According to the above table, the size of the next subinstance will be at most $\left\lfloor \frac{n}{2} \right\rfloor$. Hence,

$$W(n) = 1 + W(\lfloor \tfrac{n}{2} \rfloor) \quad for\ n > 1$$

$$W(1) = 1$$

- Prove that $W(n) = \lfloor \lg n \rfloor + 1$ by mathematical induction.

1. induction basis : For $n = 1$,

$$\lfloor \lg n \rfloor + 1 = \lfloor \lg 1 \rfloor + 1 = 0 + 1 = 1 = W(1)$$

2. induction hypothesis : For $n > 1$ and $1 < k < n$, assume that $W(k) = \lfloor \lg k \rfloor + 1$.

3. induction step : (1) n is even (i.e., $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$),

$$\begin{aligned}
W(n) &= 1 + W(\lfloor \tfrac{n}{2} \rfloor) && \text{(by recursion)} \\
&= 1 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor + 1 && \text{(by assumption)} \\
&= 2 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor \\
&= 2 + \lfloor \lg \tfrac{n}{2} \rfloor && \text{(since } n \text{ is even)} \\
&= 2 + \lfloor \lg n - 1 \rfloor \\
&= 2 + \lfloor \lg n \rfloor - 1 \\
&= 1 + \lfloor \lg n \rfloor
\end{aligned}$$

- (2) n is odd (i.e., $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2}$),

$$
\begin{aligned}
W(n) &= 1 + W(\lfloor \tfrac{n}{2} \rfloor) && \text{(by recursion)} \\
&= 1 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor + 1 && \text{(by assumption)} \\
&= 2 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor \\
&= 2 + \lfloor \lg \tfrac{n-1}{2} \rfloor && \text{(since } n \text{ is odd)} \\
&= 2 + \lfloor \lg(n-1) - 1 \rfloor \\
&= 2 + \lfloor \lg(n-1) \rfloor - 1 \\
&= 1 + \lfloor \lg(n-1) \rfloor \\
&= 1 + \lfloor \lg n \rfloor && \text{(since } n \text{ is odd)}
\end{aligned}
$$

Hence, $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$.

Mergesort

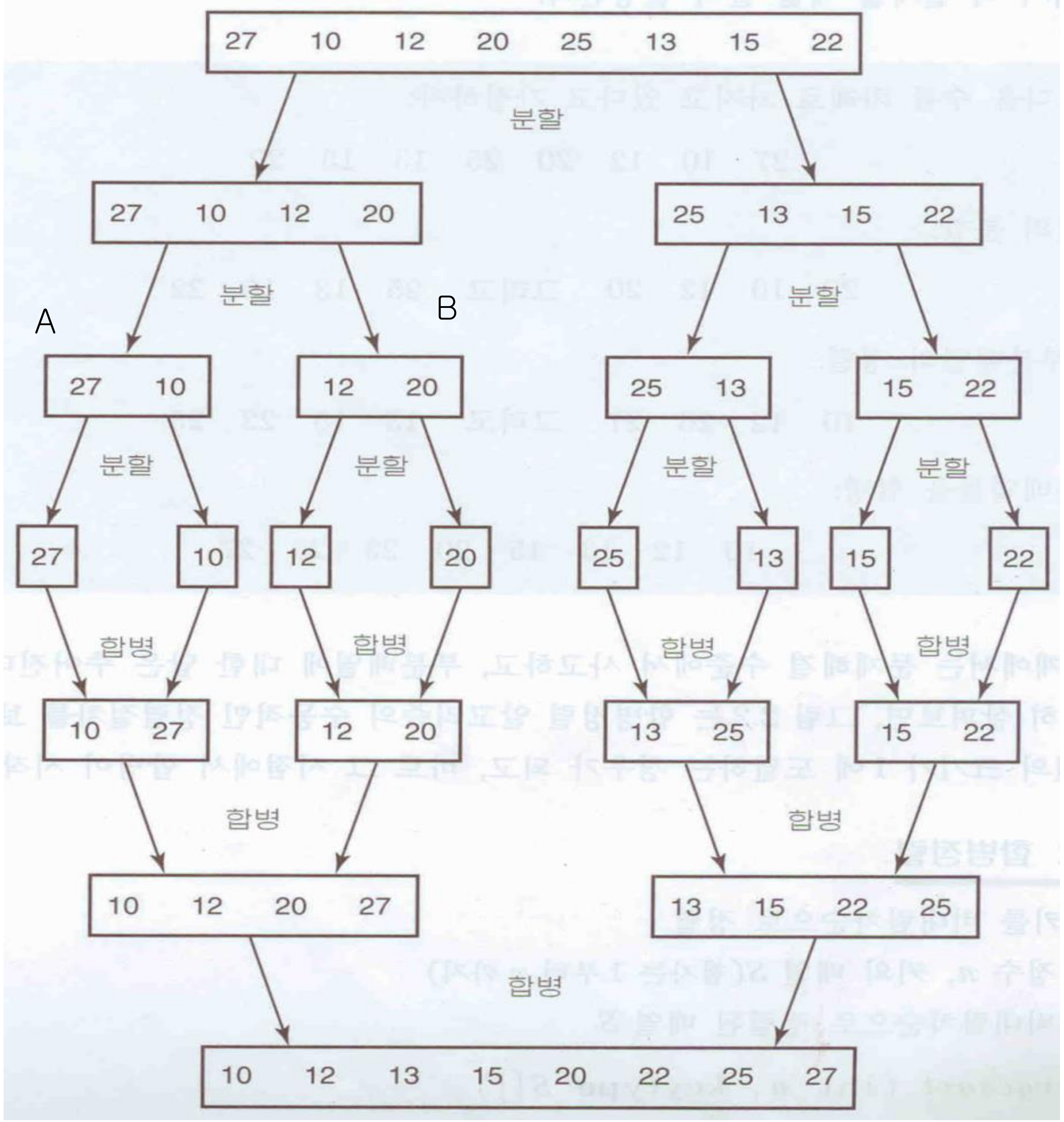
- problem: Sort *n* keys in nondecreasing sequence.
- inputs: positive integer *n*, *S*[1..*n*]
- outputs: *S*[1..*n*] containing the keys in nondecreasing order
- ex): 27, 10, 12, 20, 25, 13, 15, 22

- algorithm:

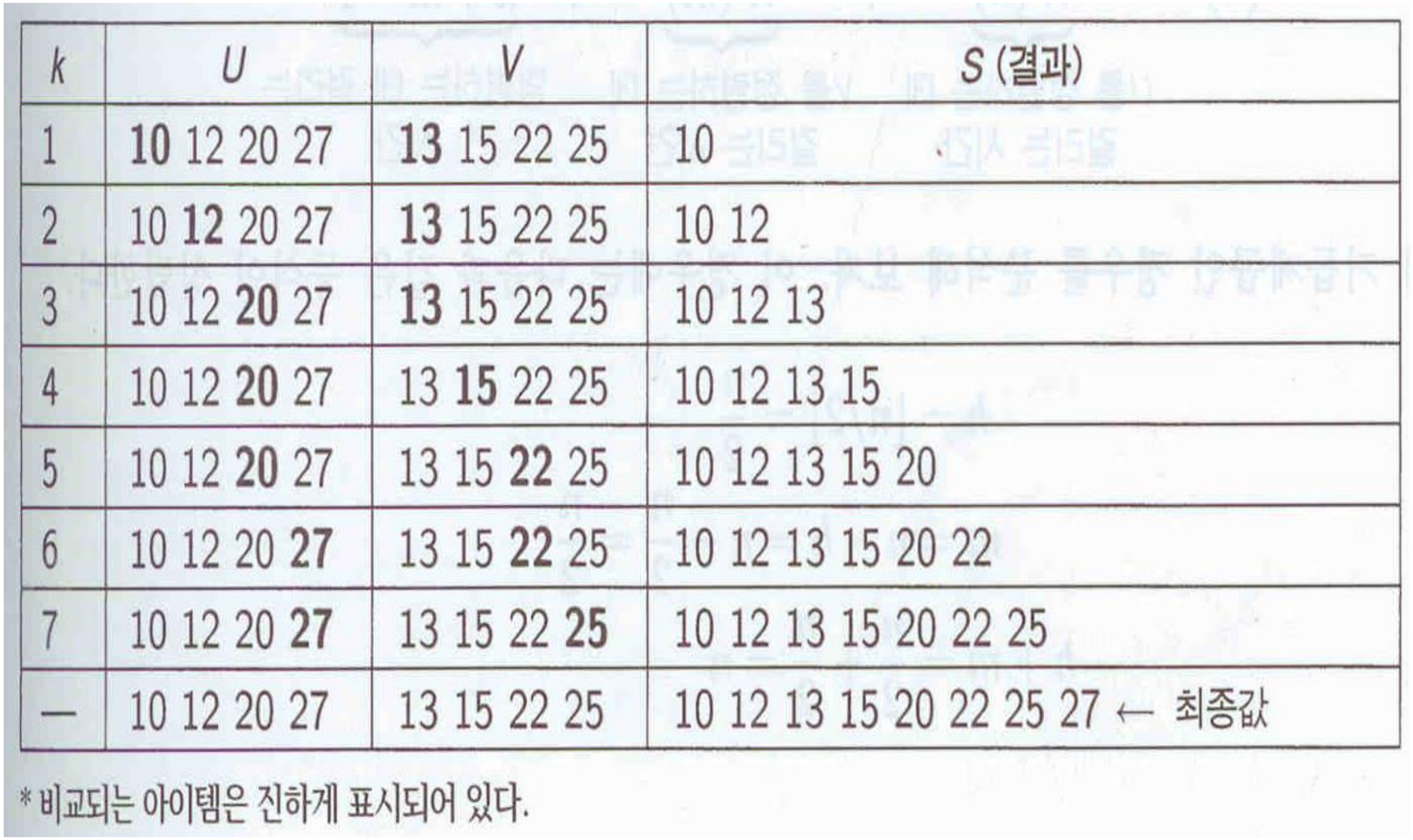
```
void mergesort (int n, keytype S[]) {
     const int h = n / 2, m = n - h;
     keytype U[1..h], V[1..m];

     if (n > 1) {
          copy S[1] through S[h] to U[1] through U[h];
          copy S[h+1] through S[n] to V[1] through V[m];
          mergesort(h,U);
          mergesort(m,V);
          merge(h,m,U,V,S);
     }
}
```

Merge

- problem: merge two sorted arrays into one sorted array.
- inputs: (1) positive integers h, m, (2) sorted keys $U[1..h]$, $V[1..m]$
- outputs: sorted $S[1..h+m]$ containing the keys in U and V.



- Fig 2.2 The steps done by a human when sorting with Mergesort



k	U	V	S (결과)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← 최종값

* 비교되는 아이템은 진하게 표시되어 있다.

- Table 2.1 An example of merging two arrays *U* and *V* into one array *S*.

```
void  merge(int h, int m, const keytype U[], const keytype V[],
            const keytype S[]) {
    index i, j, k;
    i = 1; j = 1; k = 1;
    while (i <= h && j <= m) {
          if (U[i] < V[j]) {
               S[k] = U[i];
               i++;}
          else {
               S[k] = V[j];
               j++;}
          k++;
    }
    if (i > h)
          copy V[j] through V[m] to S[k] through S[h+m];
    else
          copy U[i] through U[h] to S[k] through S[h+m];
}
```

Worst Case Time Complexity(merge)

- Worst-Case Time Complexity of *merge*
  - ✓ **basic operation**: the comparison that takes place in *merge*, i.e., comparison of U[*i*] with V[*j*].
  - ✓ **input size**: the number of items in *U* and *V*, *h* and *m*, respectively.
  - ✓ **analysis**: the worst case occurs when the loop is exited with $i = h+1$ and $j = m$ (more cases are possible). the first $m - 1$ items in *V* are placed first in *S*, followed by all *h* items in *U*, at which time the loop is exited because *i* equals $h+1$. Therefore, $W(h,m) = h + m - 1$.
  - ✓ (ex) U: 4 5 6 7   V: 1 2 3 8

Worst Case Time Complexity(mergesort)

- Worst Case Time Complexity of mergesort
  - ✓ **basic operation**: the comparison that takes place in *merge*, i.e., comparison of U[*i*] with V[*j*].
  - ✓ **input size**: size *n*, the number of items in *S*.
  - ✓ **analysis**: the worst time complexity of merge is $W(h,m) = W(h) + W(m) + h + m - 1$, where $W(h)$ and $W(m)$ are times needed to sort *U* and *V*, respectively, $h = \frac{n}{2}, m = \frac{n}{2}$ and $h + m - 1$ is needed to merge *U* and *V*. Let $n = 2^k (k \geq 1)$,

$$W(n) = 2W(\tfrac{n}{2}) + n - 1 \quad \text{for } n > 1,\ n = 2^k (k \geq 1)$$
$$W(1) = 0$$

  - ✓ Hence,

$$W(n) = \Theta(n \lg n)$$

According to the ex B.19 in Appendix B(also Theorem B.14), the solution of the above recursion is