



제6장

알고리즘의 분석: 시간복잡도

- 알고리즘의 **자원(resource)** 사용량을 분석
- 자원이란 실행 시간, 메모리, 저장장치, 통신 등
- 여기서는 **실행시간의 분석**에 대해서 다룸

시간복잡도(time complexity)

- 실행시간은 실행환경에 따라 달라짐
 - 하드웨어, 운영체제, 언어, 컴파일러 등
- 실행 시간을 측정하는 대신 연산의 실행 횟수를 카운트
- 연산의 실행 횟수는 입력 데이터의 크기에 관한 함수로 표현
- 데이터의 크기가 같더라도 실제 데이터에 따라서 달라짐
 - 최악의 경우 시간복잡도 (worst-case analysis)
 - 평균 시간복잡도 (average-case analysis)

점근적(Asymptotic) 분석

- 점근적 표기법을 사용

- 데이터의 개수 $n \rightarrow \infty$ 일때 수행시간이 증가하는 growth rate로 시간복잡도를 표현하는 기법
- Θ -표기, O -표기 등을 사용

- 유일한 분석법도 아니고 가장 좋은 분석법도 아님

- 다만 (상대적으로) 가장 간단하며
- 알고리즘의 실행환경에 비의존적임
- 그래서 가장 광범위하게 사용됨

점근적 분석의 예: 상수 시간복잡도

입력으로 n 개의 데이터가 저장된 배열 `data`가 주어지고,
그 중 $n/2$ 번째 데이터를 반환한다.

```
int sample( int data[], int n )  
{  
    int k = n/2 ;  
    return data[k] ;  
}
```

n 에 관계없이 상수 시간이 소요된다.
이 경우 알고리즘의 시간복잡도는 $O(1)$ 이다.

점근적 분석의 예: 선형 시간복잡도

입력으로 n 개의 데이터가 저장된 배열 `data`가 주어지고,
그 합을 구하여 반환한다.

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

이 알고리즘에서 가장 자주 실행되는 문장이며,
실행 횟수는 항상 n 번이다.

가장 자주 실행되는 문장의 실행횟수가 n 번이라면
모든 문장의 실행 횟수의 합은 n 에 선형적으로 비례하며,
모든 연산들의 실행횟수의 합도 역시 n 에 선형적으로 비례한다.

**선형 시간복잡도를 가진다고 말하고
 $O(n)$ 이라고 표기한다.**

선형 시간복잡도: 순차탐색

배열 data에 정수 target이 있는지 검색한다.

```
int search(int n, int data[], int target)
{
    for (int i=0; i<n; i++) {
        if (data[i] == target)
            return i;
    }
    return -1;
}
```

이 알고리즘에서 가장 자주 실행되는 문장이며,
실행 횟수는 최악의 경우 n 번이다.

최악의 경우 시간복잡도는 $O(n)$ 이다.

Quadratic

배열 x에 중복된 원소가 있는지 검사하는 함수이다.



```
bool is_distinct( int n, int x[] )
{
    for (int i=0; i<n-1; i++)
        for (int j=i+1; j<n; j++)
            if (x[i]==x[j])
                return false;
    return true;
}
```

이 알고리즘에서 가장 자주 실행되는 문장이며,
최악의 경우 실행 횟수는 $n(n-1)/2$ 번이다.

**최악의 경우 배열에 저장된 모든 원소 쌍을 비교 하므로
비교 연산의 횟수는 $n(n-1)/2$ 이다.
최악의 경우 시간복잡도는 $O(n^2)$ 으로 나타낸다.**


```
for (i=1; i<n; i*=2)
{
    // Do something
}
```

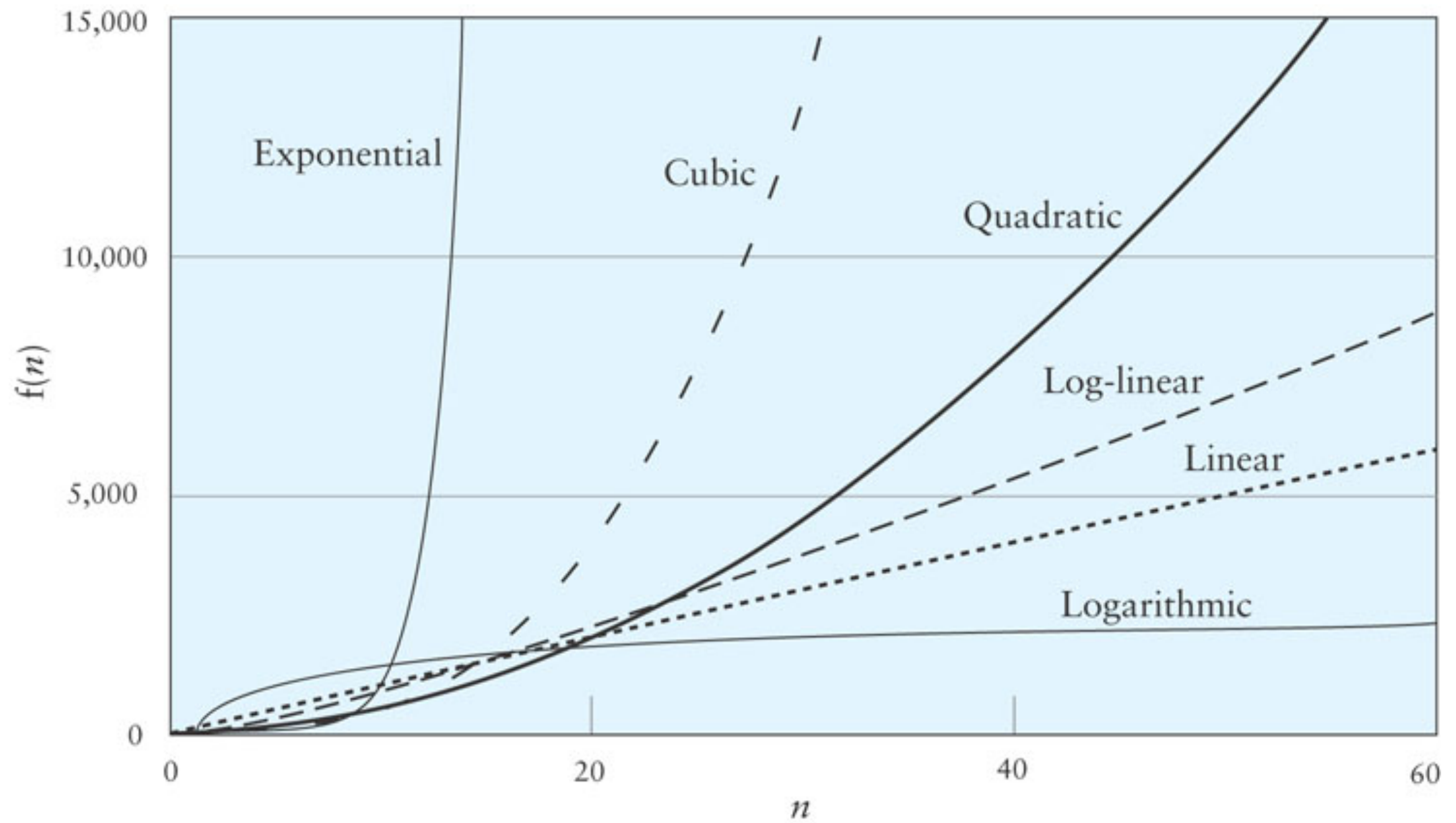
← 이 문장의 실행 횟수는?

- 알고리즘에 포함된 연산들의 실행 횟수를 표기하는 하나의 기법
- 최고차항의 차수만으로 표시
- 따라서 가장 자주 실행되는 연산 혹은 문장의 실행횟수를 고려하는 것으로 충분

Common Growth Rate

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Common Growth Rate



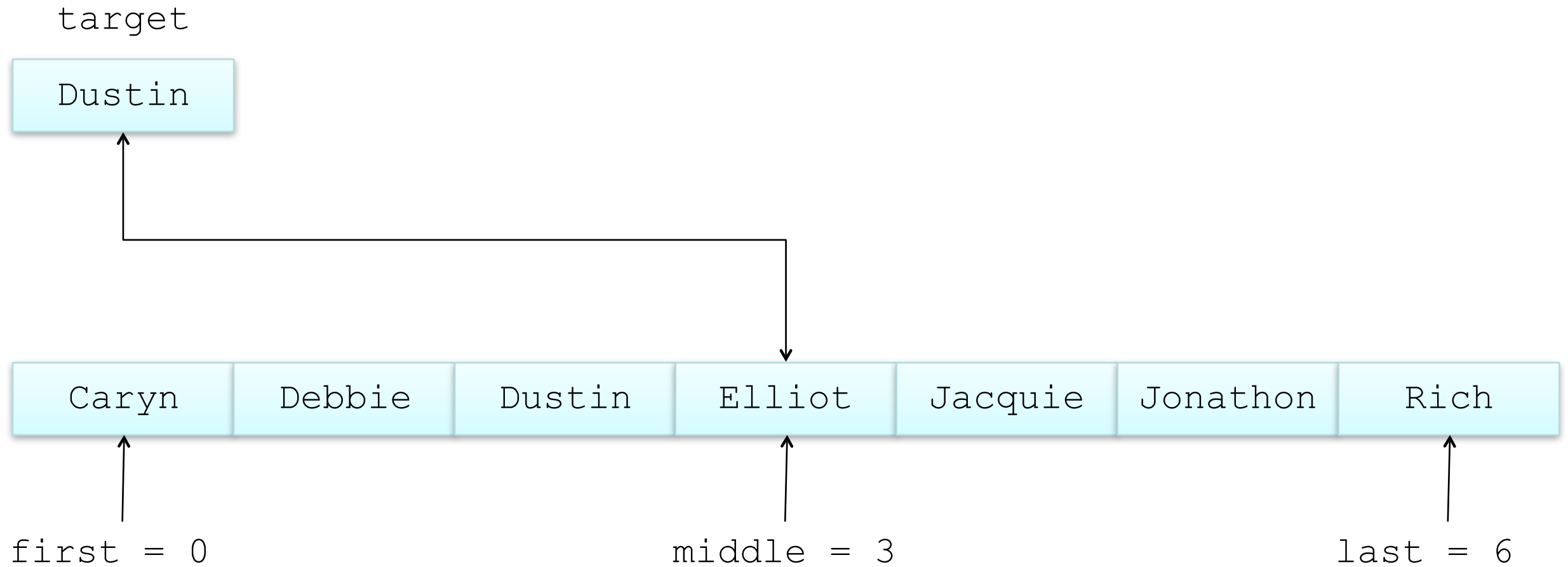
Common Growth Rate

$O(f(n))$	$f(50)$	$f(100)$	$f(100)/f(50)$
$O(1)$	1	1	1
$O(\log n)$	5.64	6.64	1.18
$O(n)$	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

이진검색과 정렬

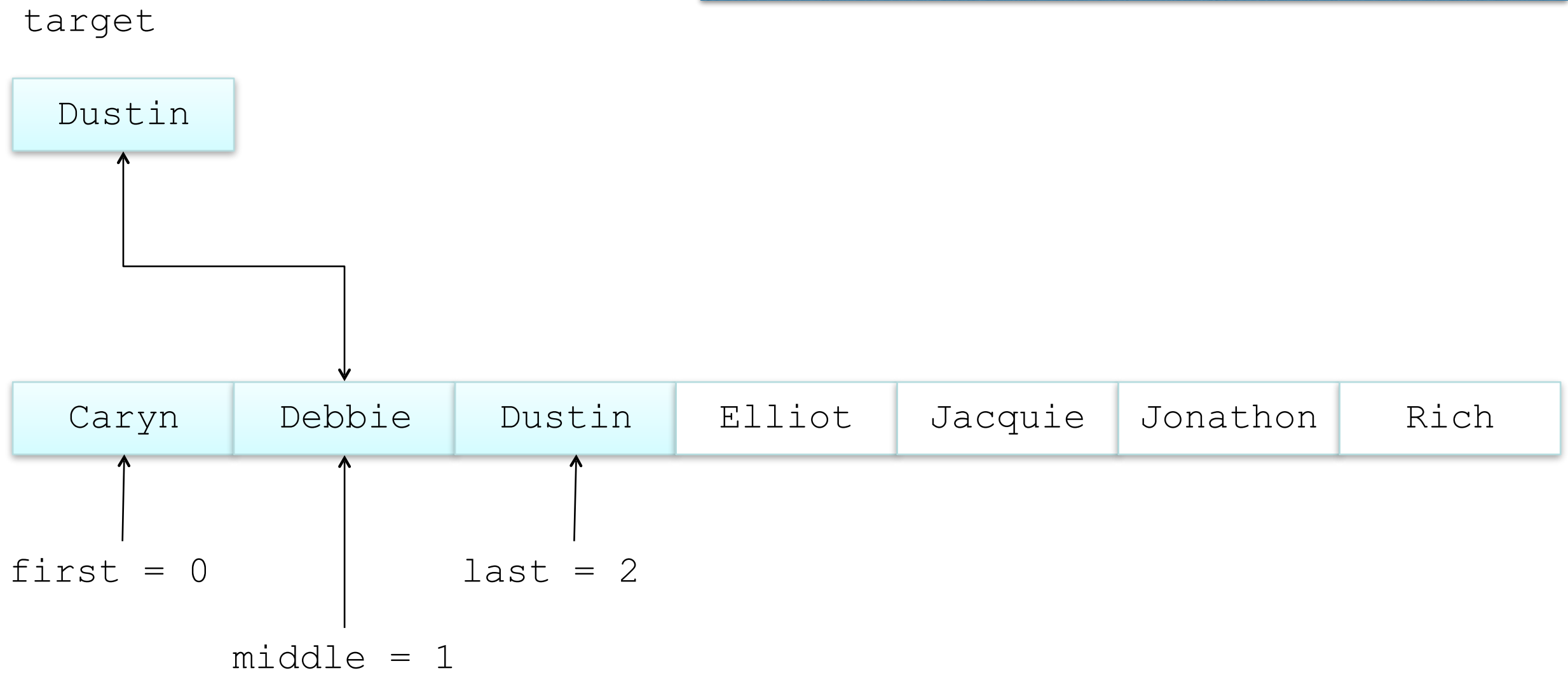
이진검색 (Binary Search)

배열에 데이터들이 오름차순으로 정렬되어 저장되어 있다.



이진검색 (Binary Search)

배열에 데이터들이 오름차순으로 정렬되어 저장되어 있다.



이진검색 (Binary Search)

배열에 데이터들이 오름차순으로 정렬되어 저장되어 있다.

target

Dustin

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich
-------	--------	--------	--------	---------	----------	------

first= middle = last = 2

이진검색

배열 data에 n개의 문자열이 오름차순으로 정렬되어 있다.

```
int binarySearch(int n, char *data[], char *target) {  
    int begin = 0, end = n-1;  
    while(begin <= end) {  
        int middle = (begin + end)/2;  
        int result = strcmp(data[middle], target);  
        if (result == 0)  
            return middle;  
        else if (result < 0)  
            begin = middle+1;  
        else  
            end = middle-1;  
    }  
    return -1;  
}
```

한 번 비교할 때마다 남아있는 데이터가 절반으로 줄어든다.
따라서 시간복잡도는 $O(\log_2 n)$ 이다.

- 데이터가 연결리스트에 오름차순으로 정렬되어 있다면?
 - 연결리스트에서는 가운데(middle) 데이터를 $O(1)$ 시간에 읽을 수 없음
 - 따라서 이진검색을 할 수 없다.
- 순차검색의 시간복잡도는 $O(n)$ 이고 이진검색은 $O(\log_2 n)$ 이다. 둘의 차이는 매우 크다.

버블 정렬 (bubble sort)

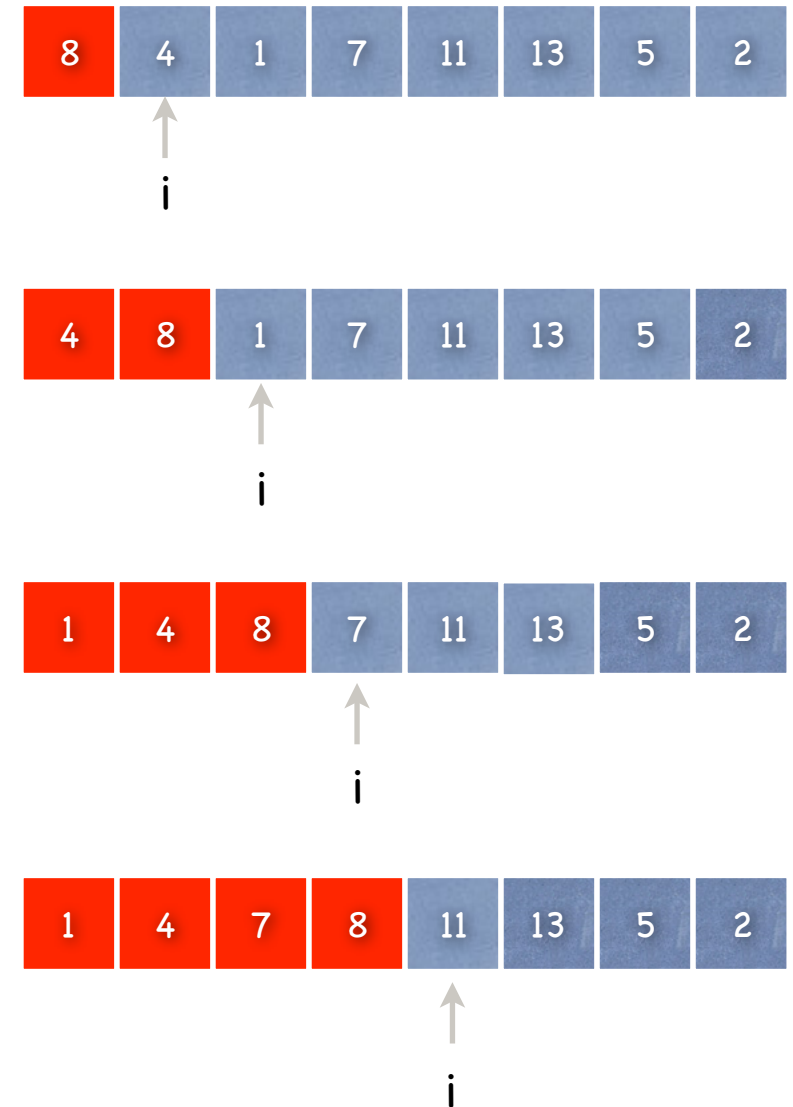
```
void bubbleSort(int data[], int n) {  
    for ( int i=n-1; i>0; i--) {  
        for ( int j=0; j<i; j++ ) {  
            if (data[j] > data[j+1]) {  
                int tmp = data[j];  
                data[j] = data[j+1];  
                data[j+1] = tmp;  
            }  
        }  
    }  
}
```

시간복잡도는 ?

삽입정렬 (Insertion sort)

```
void insertion_sort(int n, int data[]) {  
    for (int i=1; i<n; i++) {  
        int tmp = data[i];  
        int j = i-1;  
        while (j>=0 && data[j]>data[i]) {  
            data[j+1] = data[j];  
            j-;  
        }  
        data[j+1] = tmp;  
    }  
}
```

data[i]를 data[0] ~
data[i-1] 중에 제자리를
찾아 삽입하는 일



시간복잡도는 ?

- 퀵소트(quick sort) 알고리즘
 - 최악의 경우 $O(n^2)$, 하지만 평균 시간복잡도는 $O(n \log_2 n)$
- 최악의 경우 $O(n \log_2 n)$ 의 시간복잡도를 가지는 정렬 알고리즘
 - 합병정렬(merge sort)
 - 힙 정렬(heap sort) 등
- 데이터가 배열이 아닌 연결리스트에 저장되어 있다면?

Code Review

push and pop: 배열로 구현한 경우

```
void push(Stack s, Item i)
{
    if (is_full(s))
        reallocate(s);
    s->top++;
    s->contents[s->top] = i;
}
```

스택에 저장된 데이터의 개수를 n 이라고 하면
reallocate함수의 시간복잡도는 $O(n)$ 이다.

reallocate함수를 제외한
나머지 부분의 시간복잡도는 $O(1)$ 이다.

```
Item pop(Stack s)
{
    if (is_empty(s))
        terminate("Error in pop: stack is empty.");
    s->top--;
    return s->contents[s->top+1];
}
```

pop()함수의 시간복잡도는 $O(1)$ 이다.

push and pop: 연결리스트로 구현한 경우

```
void push(Stack s, Item i)
{
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL)
        terminate("Error in push: stack is full.");

    new_node->data = i;
    new_node->next = s->top;
    s->top = new_node;
}
```

```
Item pop(Stack s)
{
    struct node *old_top;
    Item i;

    if (is_empty(s))
        terminate("Error in pop: stack is empty.");

    old_top = s->top;
    i = old_top->data;
    s->top = old_top->next;
    free(old_top);
    return i;
}
```

스택을 연결리스트로 구현한 경우 push와 pop의
시간복잡도는 $O(1)$ 이다.

enqueue and dequeue


- 배열로 구현 한 경우 역시 **reallocate**를 제외하면 **enqueue**와 **dequeue**의 시간복잡도는 **$O(1)$** 이다.
- 연결리스트로 구현한 경우 **enqueue**와 **dequeue**의 시간복잡도는 **$O(1)$** 이다.

정렬된 리스트(ordered list)에 삽입하기: 배열의 경우

```
void insert_to_ordered_array(int n, int data[], int item) {  
    int i = n-1;  
    for (; i >= 0 && data[i] > item; i--)  
        data[i+1] = data[i];  
    data[i+1] = item;  
  
}
```

최악의 경우 시간복잡도는 $O(n)$ 이다.

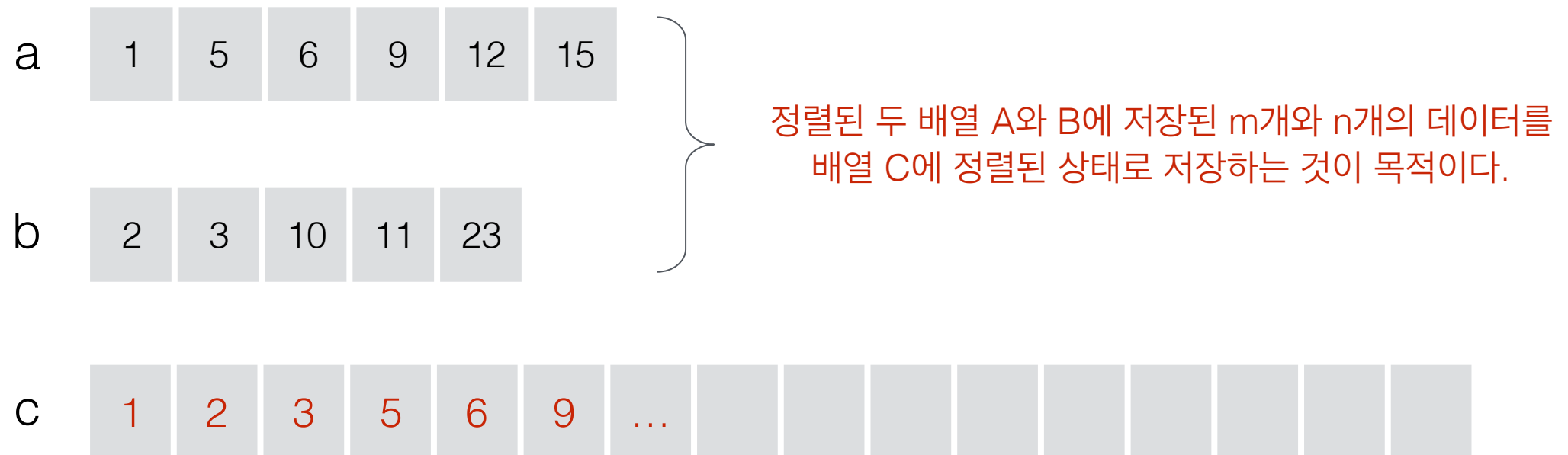
정렬된 리스트(ordered list)에 삽입하기: 연결리스트의 경우

```
Node *insert_to_ordered_linked_list(Node *head, int item) {  
    Node *new_node = (Node *)malloc(sizeof(Node));  
    new_node->data = item;  
    new_node->next = NULL;  
  
    Node *p = head, *q=NULL;  
    while (p!=NULL && p->data < item) {  
        q = p;  
        p = p->next;  
    }  
  
    if (q==NULL) {  
        new_node->next = head;  
        head = new_node;  
    }  
    else {  
        new_node->next = p;  
        q->next = new_node;  
    }  
    return head;   
}
```

최악의 경우 시간복잡도는 $O(n)$ 이다.

연결리스트의 첫 번째 노드의 주소를 저장하는 변수가 전역변수가 아니라고 가정해보았다. 그런 경우 첫 번째 노드의 주소를 변경할 가능성이 있는 함수들은 항상 첫 번째 노드의 주소를 반환하도록 구현하는 것이 일반적이다.

정렬된 두 배열 합치기



```
void merge_sorted_arrays(int m, int a[], int n, int b[], int c[])
{
    for (int i=0; i<m; i++) ← 먼저 배열 a의 데이터를 그대로 배열 c로 복사하고
        c[i] = a[i];

    for (int j=0; j<n; j++) ← 그런 다음 배열 b의 데이터 각각을 배열 c에 insert한다.
        insert_to_ordered_array(m+j, c, b[j]);
}
```

이렇게 한다면 최악의 경우 시간복잡도는 $O(mn)$ 이다.

정렬된 두 배열 합치기

a 1 5 6 9 12 15

b 2 3 10 11 23

c 

```
void merge_sorted_arrays_linear(int m, int a[], int n, int b[], int c[])
{
    int i=0, j=0, k=0;
    while (i<m && j<n) {
        if (a[i] <= b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while(i<m) c[k++] = a[i++];
    while(j<n) c[k++] = b[j++];
}
```

최악의 경우 시간복잡도는 $O(m+n)$ 이다.

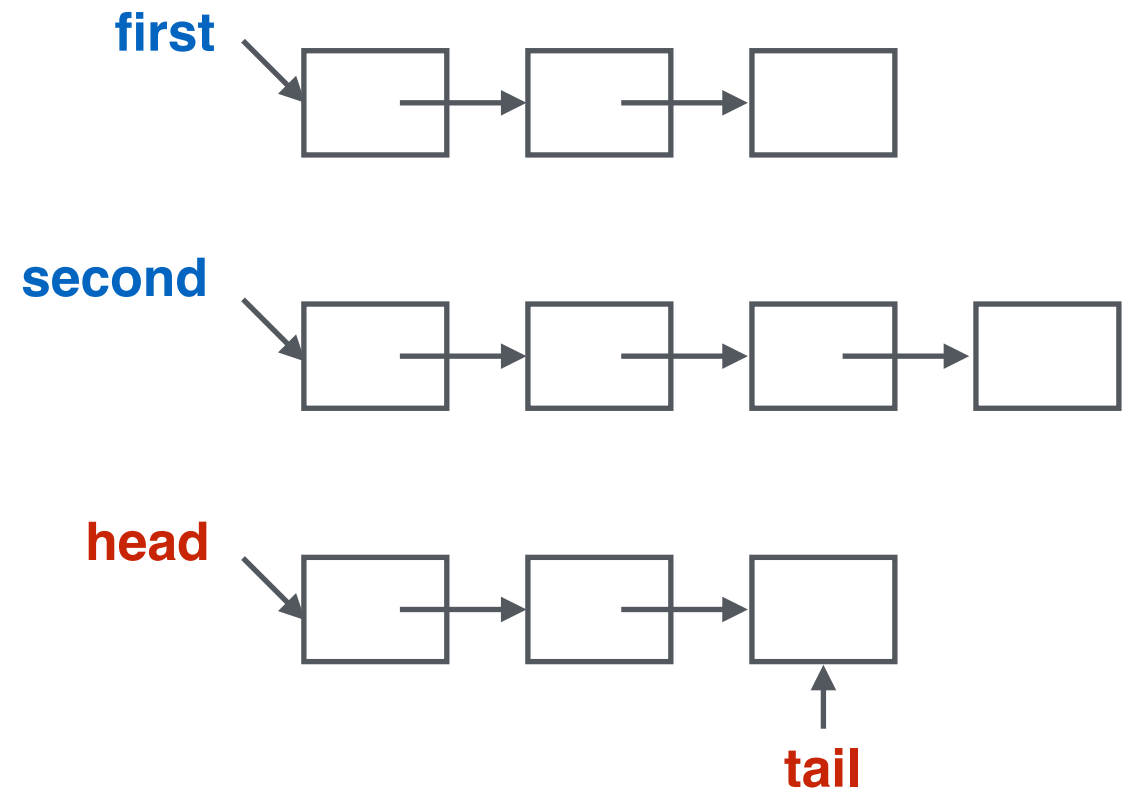
정렬된 두 연결리스트 합치기

```
Node *merge_two_ordered_list(Node *first, Node *second) {  
  
    /* insert nodes of the second list into the first list */  
    Node *p = second;  
    Node *q = first;  
  
    while (p!=NULL) {  
  
        Node *the_node = p;  
        p = p->next;  
  
        first = insert_to_ordered_list(first);  
    }  
  
    return first;  
}
```

최악의 경우 시간복잡도는 $O(mn)$ 이다.

정렬된 두 연결리스트 합치기2

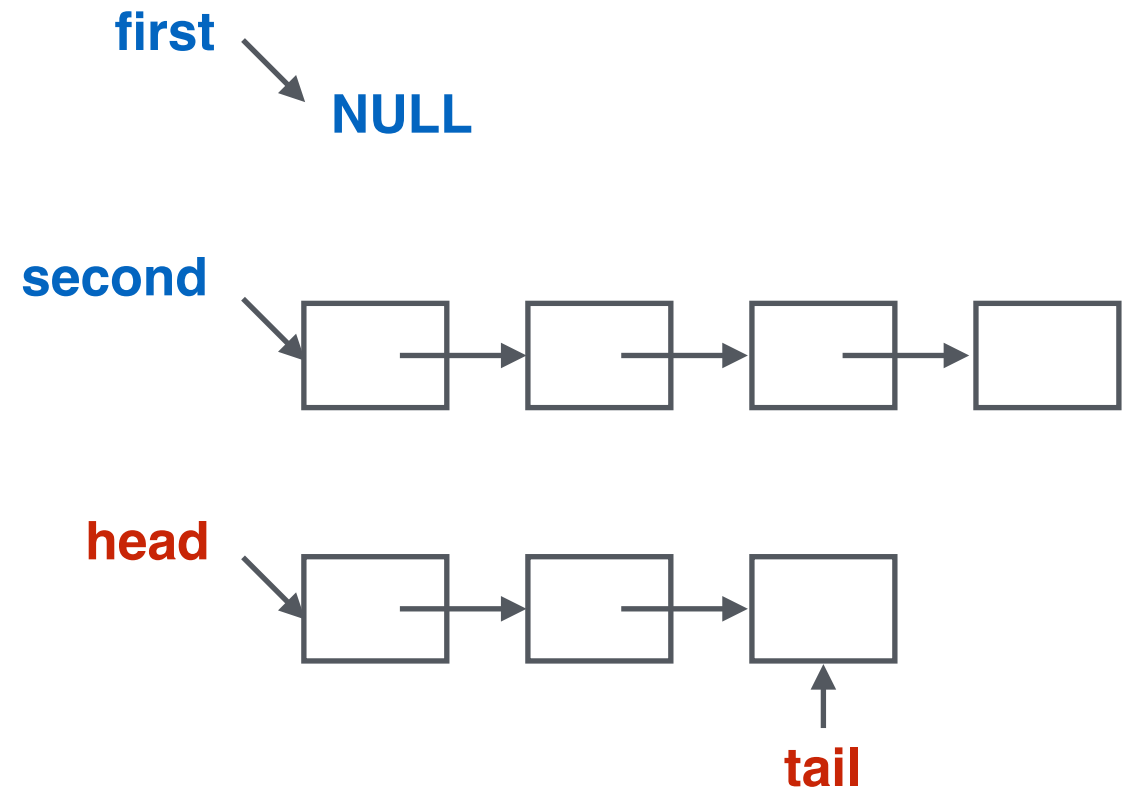
```
Node *merge_two_ordered_lists2(Node *first, Node *second) {  
    Node *head = NULL, *tail = NULL;  
    Node *tmp;  
    while (first != NULL && second != NULL) {  
        if (first->data <= second->data) {  
            tmp = first;  
            first = first->next;  
        }  
        else {  
            tmp = second;  
            second = second->next;  
        }  
        tmp->next = NULL;  
  
        if (tail == NULL) {  
            head = tail = tmp;  
        }  
        else {  
            tail->next = tmp;  
            tail = tmp;  
        }  
    }  
}
```



정렬된 두 연결리스트 합치기2

```
if (first != NULL)
    tail->next = first;
else if (second != NULL)
    tail->next = second;

return head;
}
```



최악의 경우 시간복잡도는 $O(m+n)$ 이다.

정렬된 두 연결리스트 합치기3

```
Node *merge_two_ordered_list3(Node *first, Node *second) {
    /* insert nodes of second into first */
    Node *p = second;
    Node *q = first, *pre_q = NULL;
    while (p!=NULL) {
        Node *the_node = p;
        p = p->next;

        while(q!=NULL && q->data < the_node->data) {
            pre_q = q;
            q = q->next;
        }
        if (pre_q == NULL) { /* add p at the front */
            the_node->next = first;
            first = the_node;
        }
        else { /* add after pre_q */
            the_node->next = q;
            pre_q->next = the_node;
        }
        pre_q = the_node;
    }
    return first;
}
```

최악의 경우 시간복잡도는 $O(m+n)$ 이다.

연결리스트 뒤집기

```
Node *inverse_list(Node *head) {  
    if (head == NULL || head->next == NULL)  
        return head;  
    Node *p = head;  
    Node *q = NULL;           /* before p */  
    Node *r = p->next;        /* next to p */  
  
    while (p != NULL) {  
        p->next = q;  
        q = p;  
        p = r;  
        if (r != NULL)  
            r = r->next;  
    }  
    return q;  
}
```

시간복잡도는 $O(n)$ 이다.

연결리스트에서 특정 조건을 만족하는 모든 노드 삭제하기

```
Node *remove_all_divisible(Node *head, int divisor) {
    Node *p = head;
    Node *q = NULL, *deleted = NULL;
    while(p!=NULL) {
        if (p->data%divisor == 0) {
            if (q==NULL)
                head = p->next;
            else
                q->next = p->next;
            deleted = p;
            p = p->next;
            free(deleted);
        }
        else {
            q = p;
            p = p->next;
        }
    }
    return head;
}
```

시간복잡도는 $O(n)$ 이다.

스택을 이용한 미로 찾기

```
int maze()  
{  
    Stack s = create();  
    Position cur;  
    cur.x = 0; cur.y = 0;  
    int init_dir = 0; /* 어떤 위치에 도착했을 때 처음으로 시도해 볼 이동 방향 */  
  
    while(1) {  
        maze[cur.x][cur.y] = VISITED; /* visited */  
        if (cur.x == n-1 && cur.y == n-1) {  
            break;  
        }  
  
        bool forwarded = false;  
        for (int dir = init_dir; dir<4; dir++) {  
            if (movable(cur, dir)) {  
                push(s, dir);  
                cur = move_to(cur, dir);  
                forwarded = true;  
                init_dir = 0;  
                break;  
            }  
        }  
        if (!forwarded) {  
            maze[cur.x][cur.y] = BACKTRACKED; /* backtracked */  
            if (is_empty(s))  
                break;  
            int d = pop(s);  
            cur = move_to(cur, (d+2)%4);  
            init_dir = d+1;  
        }  
    }  
}
```

어떤 셀도 2번 방문되지 않으며
이 문장에 의해서 4번 이상 검사되지 않는다.

시간복잡도는 $O(n^2)$ 이며 이는 선형시간이다.

큐를 이용한 미로 찾기

```
Queue queue = create_queue();
Position cur;
cur.x = 0;
cur.y = 0;

enqueue(queue, cur);

maze[0][0] = -1;
bool found = false;

while(!is_empty(queue)) {
    Position cur = dequeue(queue);
    for (int dir=0; dir<4; dir++) {
        if (movable(cur, dir)) {
            Position p = move_to(cur, dir);
            maze[p.x][p.y] = maze[cur.x][cur.y] - 1;
            if (p.x == n-1 && p.y == n-1) {
                printf("Found the path.\n");
                found = true;
                break;
            }
            enqueue(queue, p); ← 어떤 셀도 2번 이상 큐에 들어가지 않는다.
        }
    }
}
```

시간복잡도는 $O(n^2)$ 이며 이는 선형시간이다.