C – 알고리즘 2

정렬, 탐색 알고리즘

정렬 알고리즘

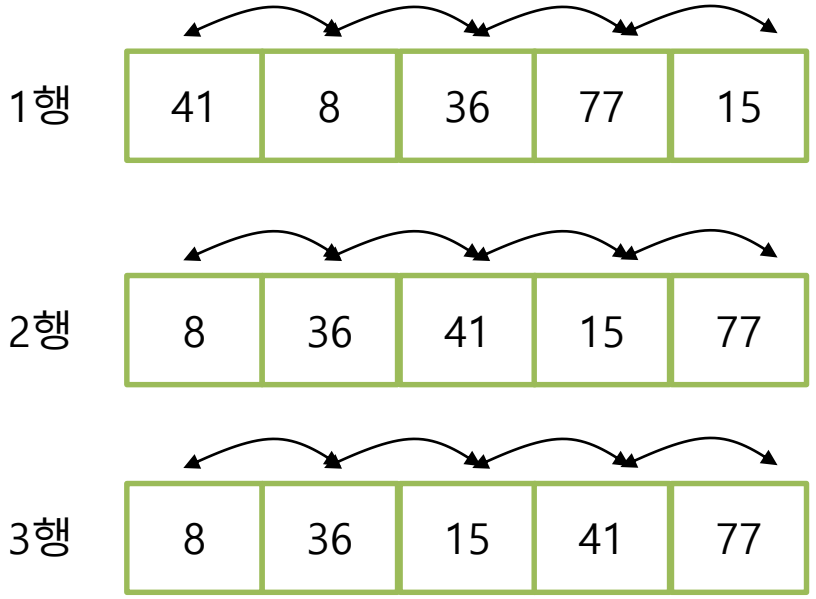
- 정렬(sorting)

  정렬(sorting)은 이름, 학번, 키 등 핵심 항목(key)의 대소 관계에 따라 데이터 집합을 일정한 순서로 줄지어 늘어서도록 바꾸는 작업을 말한다.
  이 알고리즘을 이용해 데이터를 정렬하면 검색을 더 쉽게 할수 있다.

  키값이 작은 값을 앞쪽에 놓으면 오름차순(ascending order) 정렬, 그 반대로 놓으면 내림차순(descending order) 정렬이라고 부른다.

정렬 알고리즘

- 버블 정렬(bubble sorting)
  - 리스트에서 인접한 두 개의 요소를 비교하여 자리를 바꾸는 방식
  - 요소의 개수가 n개인 배열에서 n-1회 비교 교환함



정렬 결과

[8, 36, 41, 15, 77]

[8, 36, 15, 41, 77]

[8, 15, 36, 41, 77] – 완료!

정렬 알고리즘

- 버블 정렬(bubble sorting)

```
int arr[] = { 41, 8, 36, 77, 15 };
int i, j, temp;

//비교와 교환 반복
for (i = 0; i < 5; i++) {
    for (j = 0; j < 4 - i; j++) { //열의 요소 비교
        if (arr[j] > arr[j + 1]) { //앞요소가 뒤요소보다 크면
            temp = arr[j];          //자리 바꿈 - 오름차순 정렬
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

정렬 알고리즘

- 버블 정렬(bubble sorting)

```
/*
   { 41, 8, 36, 77, 15 }
   i=0, j=0, 41>8,  { 8, 41, 36, 77, 15 }
        j=1, 41>36, { 8, 36, 41, 77, 15 }
        j=2,
        j=3, 77>15, { 8, 36, 41, 15, 77 }
   i=1, j=0,
        j=1,
        j=2, 41>15, { 8, 36, 15, 41, 77 }
   i=2, j=0,
        j=1, 36>15, { 8, 15, 36, 41, 77 } - 오름차순
   i=3, j=0
   i=4,
*/

//출력
for (i = 0; i < 5; i++) {
    printf("%d ", arr[i]); // 8 15 36 41 77
}
```

```
8 15 36 41 77
```

정렬 알고리즘

- 버블 정렬(bubble sorting) – 함수로 정의

```
void bubbleSorting(int a[], int n) {
    int i, j, temp;

    //비교와 교환 반복
    for (i = 0; i < n; i++) {
        for (j = 0; j < n-1-i; j++) { //열의 요소 비교
            if (a[j] > a[j + 1]) { //앞요소가 뒤요소보다 크면
                temp = a[j];          //자리 바꿈 - 오름차순 정렬
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

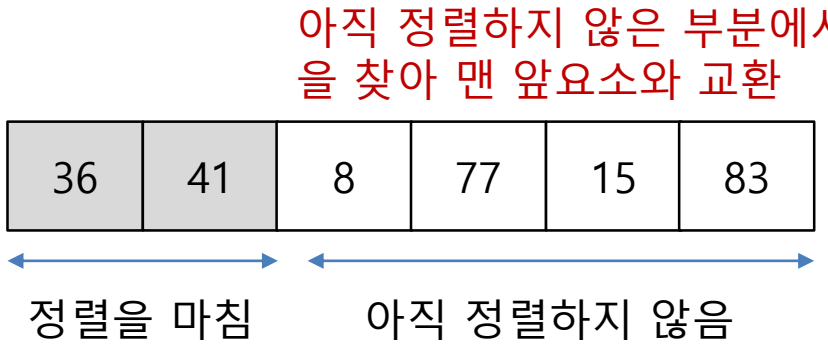
정렬 알고리즘

- 버블 정렬(bubble sorting) – 함수로 정의

```
int arr[] = { 41, 8, 36, 77, 15, 85 };
int i;
int size;  //배열의 크기

size = sizeof(arr) / sizeof(arr[0]);

//버블 정렬 함수 호출
bubbleSorting(arr, size);

//출력
for (i = 0; i < size; i++) {
    printf("%d ", arr[i]); // 8 15 36 41 77 85
}
```

정렬 알고리즘

- 선택 정렬(selection sorting)
  - 리스트에서 가장 작은 값을 찾아 맨 앞으로 보내는 방식



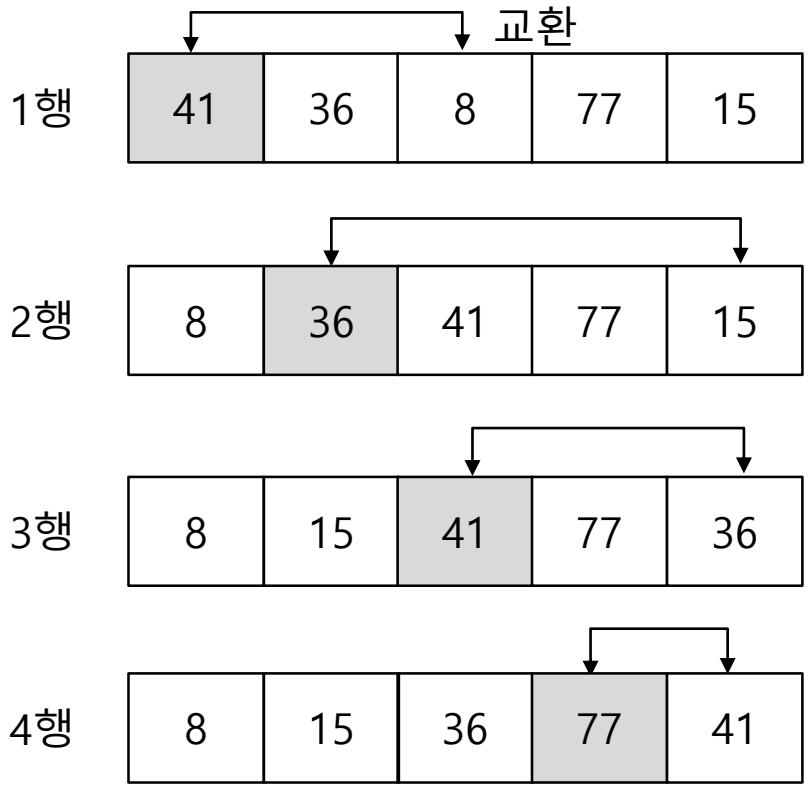
시간 복잡도: 항상 $O(n^2)$ (데이터 상태와 관계없이 비교 횟수가 일정)

장점: 교환 횟수가 최대 (n-1)번으로 적음

단점: 비교 횟수가 많아, 데이터가 많으면 느림

정렬 알고리즘

- 선택 정렬(selection sorting)



i = 0 (첫번째 위치 고정)
최소값 8 찾음. (min_idx=2)
[8, 36, 41, 77, 15]

i = 1 (첫번째 위치 고정)
최소값 15 찾음. (min_idx=4)
[8, 15, 41, 77, 36]

i = 2 (첫번째 위치 고정)
최소값 36 찾음. (min_idx=4)
[8, 15, 36, 77, 41]

i = 3 (첫번째 위치 고정)
최소값 41 찾음. (min_idx=4)
[8, 15, 36, 41, 77] - 완료

정렬 알고리즘

- 선택 정렬(selection sorting)

```c
int arr[5] = { 41, 36, 8, 77, 15 };
int i, j, temp;

for (i = 0; i < 4; i++) {
    int minIdx = i;  //현재 위치(행)를 최소값으로 설정
    for (j = i + 1; j < 5; j++) {
        if (arr[j] < arr[minIdx])
            minIdx = j; //비교후 최소값 위치 변경
    }
    //교환 처리
    temp = arr[i];
    arr[i] = arr[minIdx];
    arr[minIdx] = temp;
}

//정렬 후 출력
for (i = 0; i < 5; i++)
    printf("%d ", arr[i]);
```

정렬 알고리즘

- 선택 정렬(selection sorting)

```
{41, 36, 8, 77, 15}
1회전 (i=0)
minIdx = 0 (41)
뒤에서 최소값 탐색 → 8이 가장 작음 (minIdx = 2)
교환 → {8, 36, 41, 77, 15}

2회전 (i=1)
minIdx = 1 (36)
뒤에서 최소값 탐색 → 15 (minIdx = 4)
교환 → {8, 15, 41, 77, 36}

3회전 (i=2)
minIdx = 2 (41)
뒤에서 최소값 탐색 → 36 (minIdx = 4)
교환 → {8, 15, 36, 77, 41}

4회전 (i=3)
minIdx = 3 (77)
뒤에서 최소값 탐색 → 41 (minIdx = 4)
교환 → {8, 15, 36, 41, 77}
```

정렬 알고리즘

- 선택 정렬(selection sorting) – 함수로 구현

```
void selectionSorting(int a[], int n) {
    int i, j, temp;

    //비교와 교환 반복
    for (i = 0; i < n - 1; i++) {
        int minIdx = i;  //현재 위치(행)를 최소값으로 설정
        for (j = i + 1; j < n; j++) {
            if (a[j] < a[minIdx])
                minIdx = j; //비교후 최소값 위치 변경
        }

        temp = a[i];
        a[i] = a[minIdx];
        a[minIdx] = temp;
    }
}
```

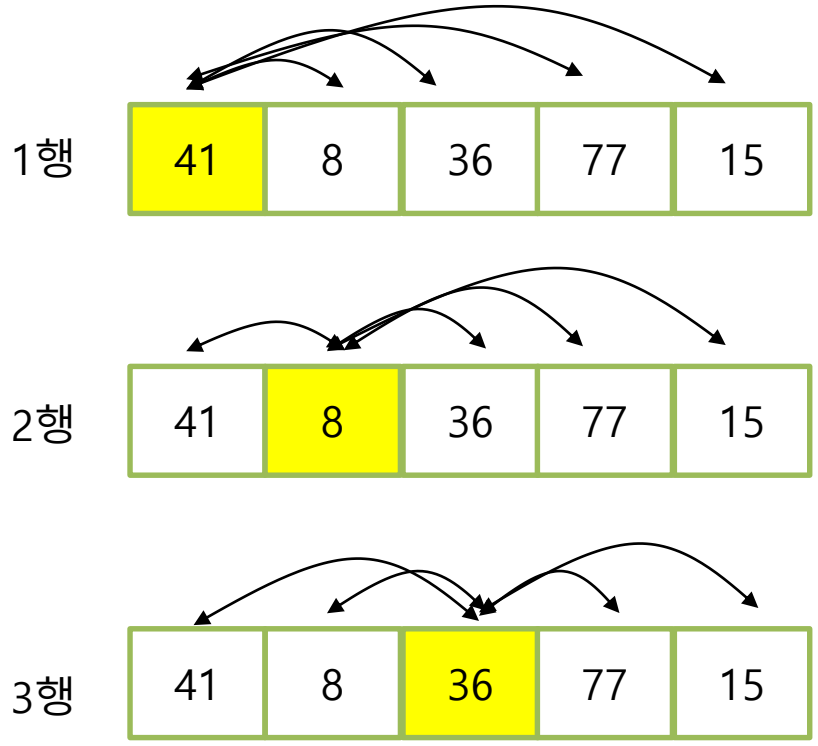
정렬 알고리즘

- 선택 정렬(selection sorting) – 함수로 구현

```
int size;  //배열의 크기
int* arr;  //배열(동적 할당)
int i;

puts("===== 선택 정렬 =====");
printf("요소의 개수 입력: ");
scanf("%d", &size);
arr = (int*)malloc(sizeof(int) * size);

for (i = 0; i < size; i++) {
    printf("arr[%d]: ", i);
    scanf("%d", &arr[i]);
}

selectionSorting(arr, size); //선택 정렬 함수 호출

for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
}
free(arr);
```

순위 정하기

- 순위 정하기



비교 결과

[2, 1, 1, 1, 1]

[2, 5, 1, 1, 1]

[2, 5, 3, 1, 1]

순위 정하기

- 순위 정하기

```
int arr[] = { 41, 8, 36, 77, 15 };
int size, i, j;
//int rank[] = { 1, 1, 1, 1, 1 };
int* rank;

size = sizeof(arr) / sizeof(arr[0]);
rank = (int*)malloc(sizeof(int) * size);

//비교후 순위 결정
for (i = 0; i < size; i++){
    rank[i] = 1;
    for (j = 0; j < size; j++) {
        if (arr[i] < arr[j]) {
            //rank[i] = rank[i] + 1;
            rank[i]++;
        }
    }
}
```

순위 정하기

- 순위 정하기

```
/*
   arr[i]   더 큰 값 개수     rank[i]
    41       1(77)               2
    8        4(41,36,77,15)      5
    36       2(41,77)            3
    77       0                   1
    15       3(41,36,77)         4
*/

//순위 출력
for (i = 0; i < size; i++) {
    printf("%d ", rank[i]); //2 5 3 1 4
}

free(rank);
```

삽입 정렬

- 순위 정하기 – 함수로 구현

```
void calcRank(int arr[], int rank[], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        rank[i] = 1; // 초기 순위
        for (j = 0; j < n; j++) {
            if (arr[i] < arr[j])
                rank[i]++;
        }
    }
}
```

삽입 정렬

- 순위 정하기 – 함수로 구현

```
int arr[] = { 41, 8, 36, 77, 15, 60};
int size = sizeof(arr) / sizeof(arr[0]);
int* rank = (int*)malloc(sizeof(int) * size);

calcRank(arr, rank, size); //순위 함수 호출

// 순위 결과 출력
for (int i = 0; i < size; i++) {
    printf("%d ", rank[i]); //3 6 4 1 5 2
}
printf("\n");

free(rank);
```

삽입 정렬

- 삽입 정렬(insert sorting)
  - 선택한 요소를 그보다 더 앞쪽의 알맞은 위치에 삽입하는 작업을 반복하여 정렬하는 알고리즘이다.

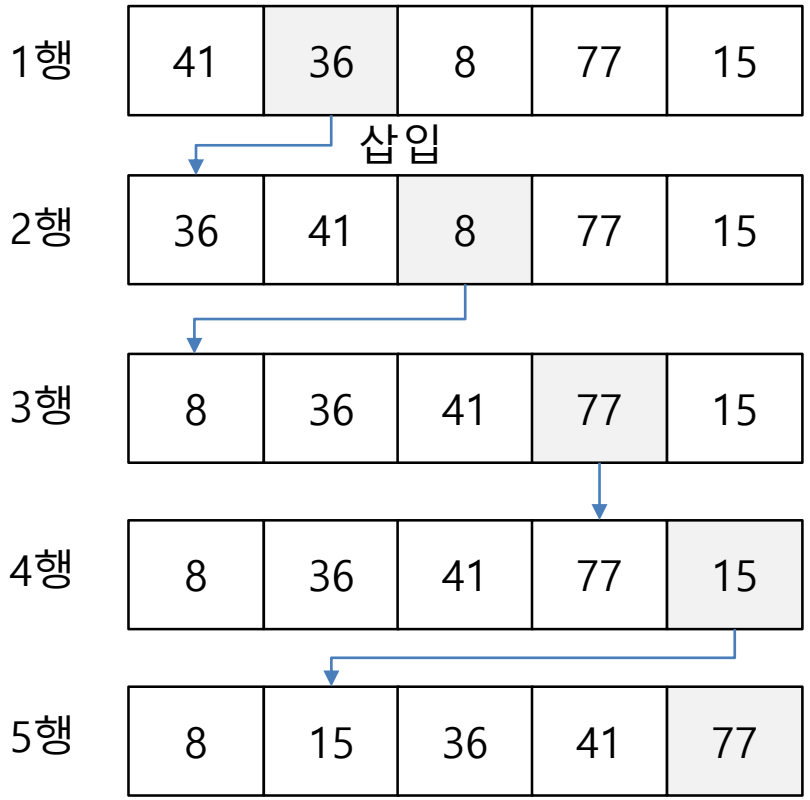
정렬되지 않은 부분의 첫 번째 요소를 정렬된 열의 알맞은 위치에 삽입하는 작업을 n-1회 반복함

장점: 구현이 간단하고, 거의 정렬된 데이터에 매우 빠름 (O(n))

단점: 데이터 개수가 많으면 비효율적

삽입 정렬

- 삽입 정렬(insert sorting



i = 1 (삽입할 요소 지정)
41은 정렬된 요소로 간주함

i = 2 (삽입할 요소 지정)
36을 41앞에 삽입

i = 3 (삽입할 요소 지정)
8을 36앞에 삽입

i = 4 (삽입할 요소 지정)
77을 그대로 유지

15를 8과 36사이에 삽입 – 정렬 완료!

정렬 알고리즘

- 선택 정렬(selection sorting)

i = 1 (36 저장)
41>36, 한칸 뒤로 밀림
[41, 41, 8, 77, 15]
36 삽입
[36, 41, 8, 77, 15]

i = 2 (8 저장)
41>8, 한칸 뒤로 밀림
[36, 41, 41, 77, 15]
36>8, 한칸 뒤로 밀림
[36, 36, 41, 77, 15]
8 삽입
[8, 36, 41, 77, 15]

삽입 정렬

- 삽입 정렬(insert sorting)

```c
int arr[5] = { 41, 36, 8, 77, 15 };
int i, j, tmp;

//삽입 정렬
for (i = 1; i < 5; i++) {
    tmp = arr[i]; //삽입할 요소 지정
    for (j = i; j > 0 && arr[j - 1] > tmp; j--) {
        arr[j] = arr[j - 1]; //한 칸씩 뒤로 밀기
    }
    //printf("%d\n", j); //0 0 3 1
    arr[j] = tmp;  //tmp를 제자리 삽입
}

//정렬 후 출력
for (i = 0; i < 5; i++)
    printf("%d ", arr[i]); //8 15 36 41 77
```

삽입 정렬

- 삽입 정렬(insert sorting)

```
{41, 36, 8, 77, 15}
i = 1
tmp = 36
j=1, 41>36, 뒤로 한 칸 밀기, {41, 41, 8, 77, 15}
j=0, 36 삽입, {36, 41, 8, 77, 15}

i = 2
tmp = 8
j=2, 41>8, {36, 41, 41, 77, 15}
j=1, 36>8, {36, 36, 41, 77, 15}
j=0, 8 삽입, {8, 36, 41, 77, 15}

i = 3
tmp = 77
j=3, 41<77, {8, 36, 41, 77, 15}

i = 4
tmp = 15
j=4, 77>15, {8, 36, 41, 77, 77}
j=3, 41>15, {8, 36, 41, 41, 77}
j=2, 36>15, {8, 36, 36, 41, 77}
j=1,
j=0, 15 삽입, {8, 15, 36, 41, 77}
```

삽입 정렬

- 삽입 정렬(insert sorting) – 함수로 구현

```
void insertSorting(int a[], int n) {
    int i, j, tmp;

    for (i = 1; i < n; i++) {
        tmp = a[i];
        for (j = i; j > 0 && a[j - 1] > tmp; j--) {
            a[j] = a[j - 1];
        }
        a[j] = tmp;
    }
}
```

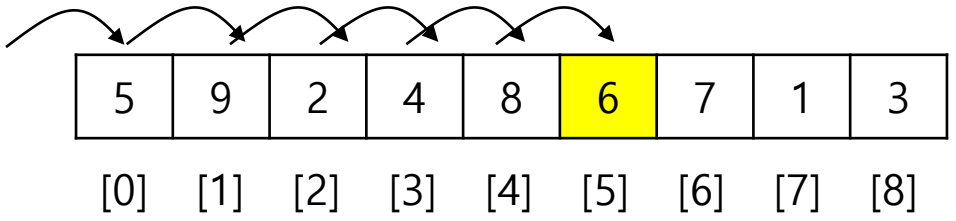
삽입 정렬

- 삽입 정렬(insert sorting) – 함수로 구현

```
int size;
int* arr; //배열(동적 할당)

puts("----- 삽입 정렬 -----");
printf("요소 개수 입력: ");
scanf("%d", &size);
arr = (int*)malloc(sizeof(int) * size);

//사용자 입력
for (int i = 0; i < size; i++) {
    printf("arr[%d]: ", i);
    scanf("%d", &arr[i]);
}

insertSorting(arr, size); //삽입 정렬 함수 호출
puts("오름차순으로 정렬했습니다.");
for (int i = 0; i < size; i++) {
    printf("arr[%d] = %d\n", i, arr[i]);
}
free(arr);
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search)
  - 동작 원리
    1. 첫번째 요소부터 하나씩 검사
    2. 찾는 값과 같으면 위치 출력
    3. 찾았으면 종료
    4. 끝까지 못찾으면 "없음" 출력
  - 특징
    1. 구현이 매우 간단하다.
    2. 데이터가 많아지면 속도가 느려진다. – 시간 복잡도 O(n)
    3. 불필요한 비교 – 값을 찾았어도 반복문이 끝가지 돎



검색 알고리즘

- 배열에서 값 찾기 1

```
int a[] = { 9, 8, 7, 6, 7 };
int i;
int count = 0;

//7이 몇 개인지 세기
for (i = 0; i < 5; i++){
    if (a[i] == 7) {
        printf("7 발견!\n");
        count++;
    }
}
printf("7을 %d개 발견!", count);
```

검색 알고리즘

- 배열에서 값 찾기 2

```
//7을 하나 발견하면 종료
int sw = 0;  //상태(토글) 변수
for (i = 0; i < 7; i++) {
    if (a[i] == 7) {
        printf("7 발견!\n");
        sw = 1;
        break;
    }
}

if(sw == 0)
    printf("7을 발견 못함!\n");
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search)

```c
int a[9] = { 5, 9, 2, 4, 8, 6, 7, 1, 3 };
int i;
int x = 6;  //찾을 값
int found = 0; //상태(찾음, 못찾음)

for (i = 0; i < 9; i++) {
    if (a[i] == x) {
        printf("%d은 a[%d]에 있습니다.\n", x, i);
        found = 1; //찾음
        break;  //더 이상 찾을 필요 없음!!
    }
}

if (!found) {
    printf("%d은 없습니다.\n");
}
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 함수로 구현

```
void sequentialSearch(int a[], int n, int x) {
    int i, found = 0;

    for (i = 0; i < n; i++) {
        if (a[i] == x) {
            printf("%d은 a[%d]에 있습니다.\n", x, i);
            found = 1; //찾음
            break;  //더 이상 찾을 필요 없음!!
        }
    }

    if (!found) {
        printf("%d은 없습니다.\n");
    }
}
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 함수로 구현

```
int arr[9] = { 5, 9, 2, 4, 8, 6, 7, 1, 3 };
int size;  //배열의 크기
int x = 6; //찾을 값

size = sizeof(arr) / sizeof(arr[0]);

//순차 탐색 함수 호출
sequentialSearch(arr, size, x);
```

```
6은 a[5]에 있습니다.
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 함수로 구현

```
int search(int a[], int n, int x) {
    int i = 0;
    while (1) {
        if (i == n)
            return -1;  //검색 실패
        if (a[i] == x)
            return i;   //검색 성공
        i++;
    }
}
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 함수로 구현

```
int arr[9] = { 5, 9, 2, 4, 8, 6, 7, 1, 3 };
int size;  //배열의 크기
int x = 10; //찾을 값

size = sizeof(arr) / sizeof(arr[0]);

//순차 탐색 함수 호출
int idx = search(arr, size, x);
if (idx == -1)
    puts("검색에 실패했습니다.");
else
    printf("%d은 a[%d]에 있습니다.\n", x, idx);
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 중복값 찾기

```
int searchAll(int a[], int n, int x, int idxs[]) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            idxs[count++] = i; // 발견한 인덱스를 저장
        }
    }
    return count; // 찾은 개수 반환
}
```

검색 알고리즘 – 순차 검색

- 순차 검색(sequential search) – 중복값 찾기

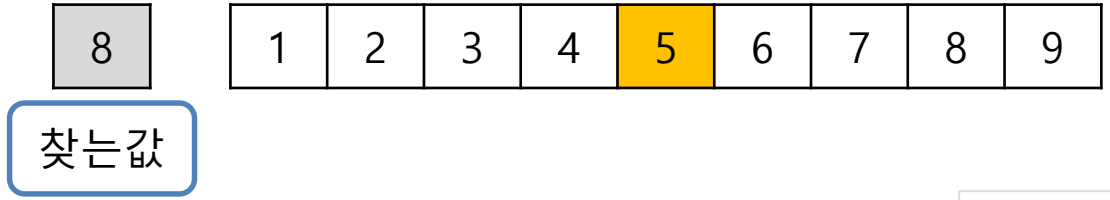
```
int arr[9] = { 5, 9, 2, 4, 8, 6, 7, 1, 6 }; // 6이 두 번 있음
int size = sizeof(arr) / sizeof(arr[0]);
int x = 6; // 찾을 값
int* idxs; // 찾은 인덱스 저장
int count; // 검색 값의 개수

idxs = (int*) malloc(sizeof(int) * size);

count = searchAll(arr, size, x, idxs); //순차 탐색 호출
if (count == 0) {
    puts("검색에 실패했습니다.");
}
else {
    printf("%d은 총 %d개 발견되었습니다.\n", x, count);
    printf("위치: ");
    for (int i = 0; i < count; i++) {
        printf("a[%d] ", idxs[i]);
    }
    printf("\n");
}
free(idxs);
```

검색 알고리즘 – 이분 검색

- 이분 검색(binary search)

  정렬된 데이터를 좌우 둘로 나눠서 찾는 값의 검색 범위를 좁혀가는 방식

  - 동작 원리
    - 찾을 값 < 가운데 요소 -> 오른쪽 반을 검색 범위에서 제외시킴(8 < 5)
    - 찾을 값 > 가운데 요소 -> 왼쪽 반을 검색 범위에서 제외시킴(8 > 5)
    - 찾을 값 = 가운데 요소 -> 검색을 완료함



$\log_{10}(1,000,000) = 6$ (10^6 = 1,000,000)

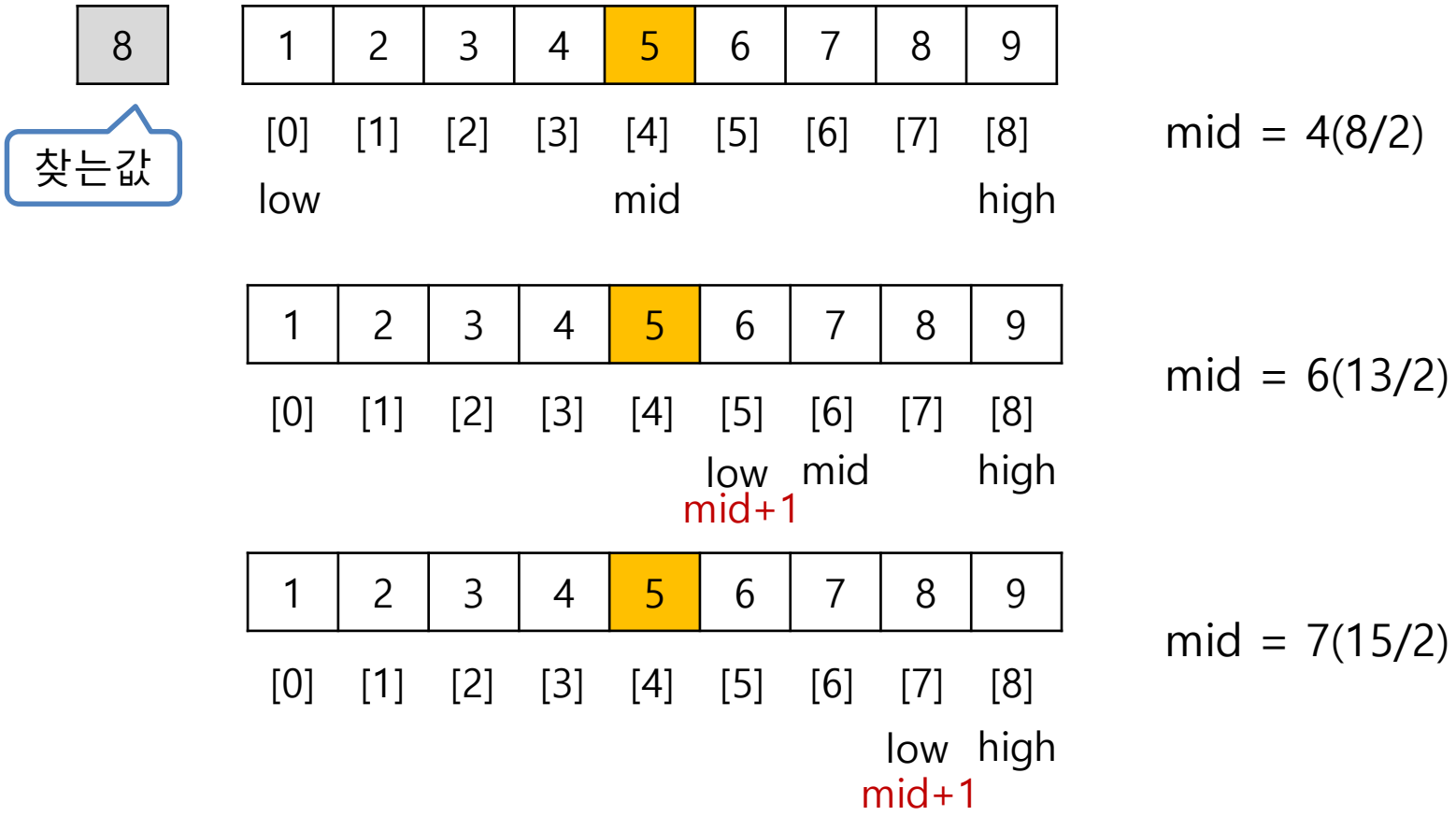
$\log_{10}(2) \approx 0.30102999566$

$$\frac{6}{0.30102999566} \approx 19.9315685693$$

- 특징
  1. 검색 속도가 빠르다.
  2. 먼저 정렬이 되어 있어야 하는 제약이 있음
  3. 시간 복잡도 O(log n) – 예) n=1000000이면 20회 정도

검색 알고리즘 – 이분 검색

- 이분 검색(binary search)



검색 알고리즘 – 이분 검색

- 이분 검색(binary search)

```
int low, high, mid;
int x, found;

//정렬된 배열
int arr[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

low = 0;  //첫 인덱스
high = 8; //마지막 인덱스
x = 8;    //찾을 값
found = 0; //상태(찾음/못찾음)

while (low <= high) {
    mid = (low + high) / 2;  //중간값의 위치
    //printf("%d\n", mid); //4 -> 6 -> 7

    if (arr[mid] == x) {
        printf("%d은 a[%d]에 있습니다.", x, mid);
        found = 1; //찾음
        break;
    }
```

검색 알고리즘 – 이분 검색

- 이분 검색(binary search)

```
        else if (arr[mid] < x) {
            low = mid + 1;
        }
        else { //a[mid] > x
            high = mid - 1;
        }
        /*
            mid=4, 5<8, low=5, high=8, mid=6(13/2)
            mid=6, 7<8, low=7, high=8, mid=7(15/2)
            mid=7, 8=8, 찾음
        */
    }

    if (!found) //찾지 못함
        printf("%d은 없습니다.", x);

    return 0;
}
```

검색 알고리즘 – 이분 검색

- 이분 검색(binary search) – 함수로 구현

```
int binarySearch(int a[], int n, int x) {
    int low = 0;
    int high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (a[mid] == x)
            return mid; // 찾은 위치 반환
        else if (a[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // 못 찾음
}
```

검색 알고리즘 – 이분 검색

- 이분 검색(binary search) – 함수로 구현

```
int arr[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int size = sizeof(arr) / sizeof(arr[0]);
int x = 8;

int idx = binarySearch(arr, size, x);
if (idx == -1)
    printf("%d은 없습니다.\n", x);
else
    printf("%d은 a[%d]에 있습니다.\n", x, idx);
```

분할 정복 방식 – 알고리즘

- 분할 정복 방식(Divide & Conquer)

  알고리즘 설계 기법 중 하나로, 큰 문제를 여러 개의 작은 문제로 나누어 해결하고,
  그 결과를 합쳐서 전체 문제를 해결하는 방식입니다.

  - 분할(Divide) → 문제를 더 작은 부분 문제로 나눔
  - 정복(Conquer) → 작은 문제들을 재귀적으로 해결
  - 결합(Combine) → 해결된 작은 문제들을 합쳐서 원래 문제의 답을 구함

  ◆ 대표적인 예시

  1. 이진 탐색 (Binary Search)
     분할: 배열의 중앙값과 찾는 값을 비교
     정복: 중앙값보다 크면 오른쪽, 작으면 왼쪽 절반에서 탐색
     결합: 필요 없음 (이미 한쪽에서 답을 찾음)

  2. 퀵 정렬 (Quick Sort)
     분할: 피벗을 기준으로 작은 값과 큰 값으로 배열을 나눔
     정복: 각각의 부분 배열을 재귀적으로 정렬
     결합: 분할된 배열이 모두 정렬되면 전체 배열이 정렬 완료

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬

퀵 정렬(Quick Sort)은 분할 정복(divide and conquer) 방식으로 동작하는 매우 효율적인 정렬 알고리즘입니다.

➢ 퀵 정렬(Quick Sort) 동작 원리

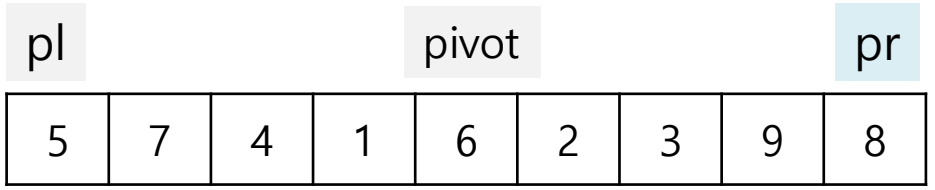
1. 피벗(Pivot) 선택
배열에서 하나의 원소를 피벗(pivot)으로 선택합니다.
보통은 가운데 값, 랜덤 값등을 사용합니다.

2. 분할(Partition)
피벗보다 작은 값들은 배열의 왼쪽에, 피벗보다 큰 값들은 배열의 오른쪽에 배치합니다.
이 과정을 "분할"이라고 합니다.

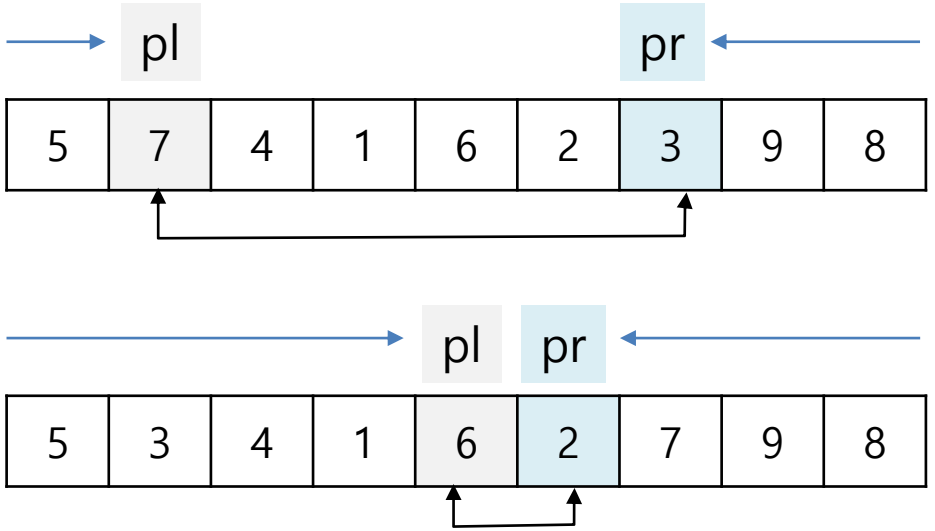
3. 재귀 호출(Recursion)
피벗을 기준으로 나뉜 왼쪽 부분 배열과 오른쪽 부분 배열에 대해 같은 과정을 반복합니다.
각 부분 배열이 길이가 1 이하가 되면 정렬이 끝납니다.

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 동작 원리

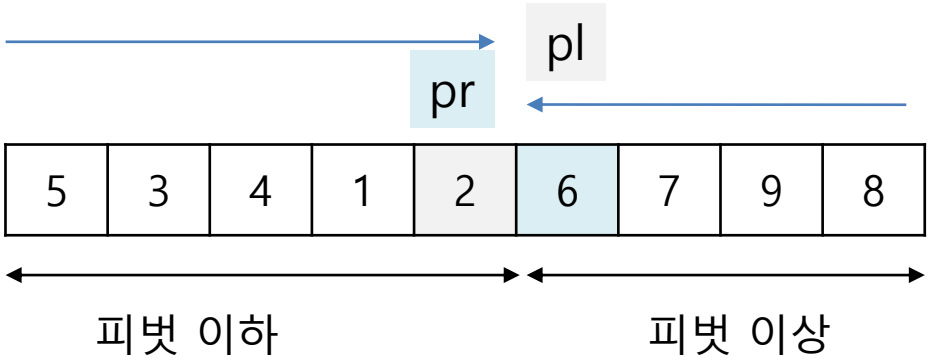


a[pl] >= pivot 요소를 찾을때까지 pl(커서)을 오른쪽으로 이동함
a[pr] <= pivot 요소를 찾을때까지 pr(커서)을 왼쪽으로 이동함



정렬 알고리즘 – 퀵 정렬

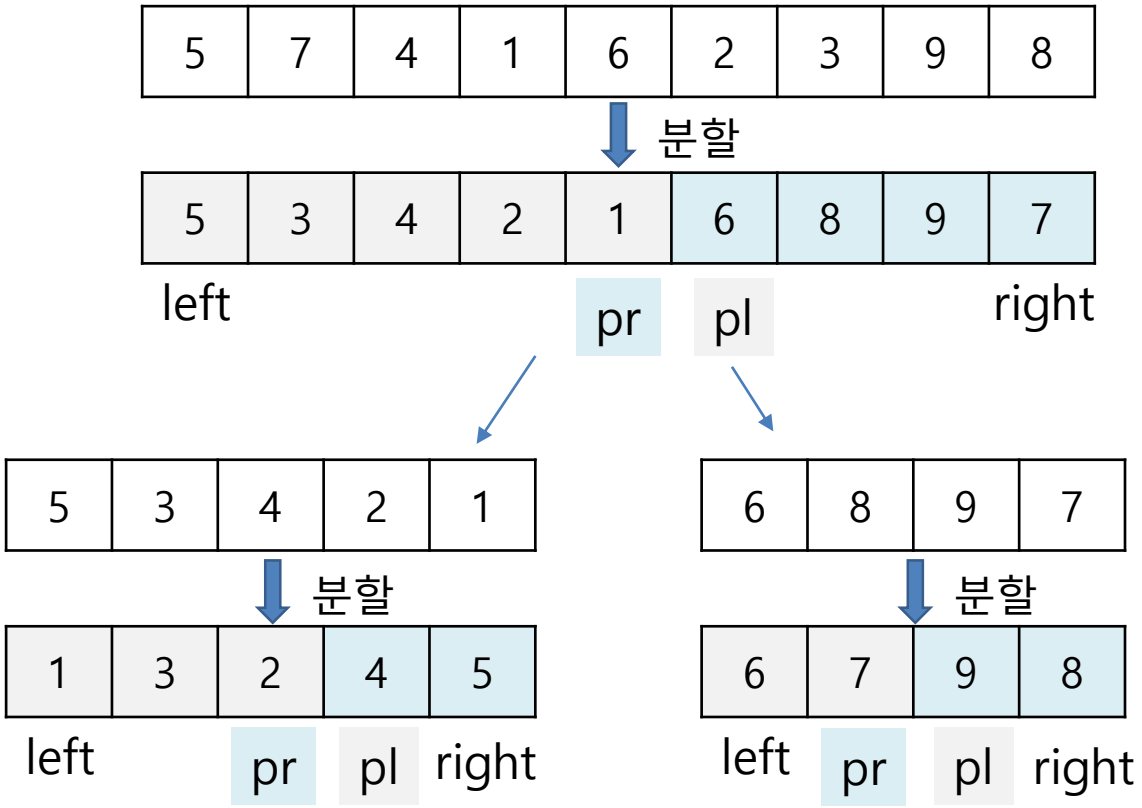
- 퀵 정렬 동작 원리



두 커서(pl, pr)이 교차함

피벗 이하의 그룹: a[0], …. a[pl-1]
피벗 이상의 그룹: a[pr+1], …. a[n-1]

정렬 알고리즘 – 퀵 정렬



pr이 a[0]보다 오른쪽에 있으면(left < pr) 왼쪽 그룹을 나눈다.
Pl이 a[8]보다 왼쪽에 있으면(pl < right) 오른쪽 그룹을 나눈다.
요소의 개수가 1개인 그룹은 더 이상 나눌수 없다.

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 - 분할

```
// 배열의 두 원소 교환
void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

//배열(a)를 피벗(pivot)을 기준으로 나눔
void partition(int a[], int n) {
    int pl = 0;     //왼쪽 커서
    int pr = n - 1; //오른쪽 커서
    int pivot = a[n / 2]; //피벗은 가운데 요소
    int temp;  //교환을 위한 임시 변수
```

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 - 분할

```
    while (pl <= pr){
        while (a[pl] < pivot)
            pl++;  //피벗보다 큰 값 나올 때까지 이동
        while (a[pr] > pivot)
            pr--;  //피벗보다 작은 값 나올 때까지 이동
        if (pl <= pr) {
            //교환
            swap(&a[pl], &a[pr]);
            pl++;
            pr--;
        }
    }
    printf("피벗의 값은 %d입니다.\n", pivot);
    printf("피벗 이하의 그룹\n");
    for (int i = 0; i <= pl - 1; i++)
        printf("%d ", a[i]); //a[0]~a[pl-1]
    putchar('\n');
    printf("피벗 이상의 그룹\n");
    for (int i = pr + 1; i < n; i++)
        printf("%d ", a[i]); //a[pr+1]~a[n-1]
    putchar('\n');
}
```

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 - 분할

```
int main()
{
    int arr[] = { 1, 8, 7, 4, 5, 2, 6, 3, 9 };
    int size = sizeof(arr) / sizeof(arr[0]);

    partition(arr, size);

    return 0;
}
```

```
피벗의 값은 5입니다.
피벗 이하의 그룹
1 3 2 4 5
피벗 이상의 그룹
5 7 6 8 9
```

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 – 재귀 호출

```
// 배열의 두 원소 교환
void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

// 퀵 정렬의 분할 함수
void quickSorting(int a[], int left, int right) {
    int pl = left;          // 왼쪽 커서
    int pr = right;         // 오른쪽 커서
    int pivot = a[(pl + pr) / 2]; // 가운데 값 피벗
```

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 – 재귀 호출

```
    while (pl <= pr) {
        while (a[pl] < pivot) pl++;  // 피벗보다 큰 값 나올 때까지 이동
        while (a[pr] > pivot) pr--;  // 피벗보다 작은 값 나올 때까지 이동
        if (pl <= pr) {
            swap(&a[pl], &a[pr]);    // 교환
            pl++;
            pr--;
        }
    }
    if (left < pr)
        quickSorting(a, left, pr);  //왼쪽부분 재귀 호출
    if (pl < right)
        quickSorting(a, pl, right); //오른쪽 부분 재귀 호출
}
```

정렬 알고리즘 – 퀵 정렬

- 퀵 정렬 – 재귀 호출

```
int size; //배열 요소의 수
int* arr; //배열(동적 할당)

puts(">>> 퀵 정렬 >>>");
printf("요소의 개수: ");
scanf("%d", &size);
arr = (int*)malloc(sizeof(int) * size);

//사용자 입력
for (int i = 0; i < size; i++) {
    printf("arr[%d]: ", i);
    scanf("%d", &arr[i]);
}

quickSorting(arr, 0, size - 1); //퀵 정렬 함수 호출

printf("정렬 후: ");
for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
free(arr); //메모리 반납
```

```
>>> 퀵 정렬 >>>
요소의 개수: 9
arr[0]: 8
arr[1]: 1
arr[2]: 5
arr[3]: 3
arr[4]: 6
arr[5]: 7
arr[6]: 4
arr[7]: 2
arr[8]: 9
정렬 후: 1 2 3 4 5 6 7 8 9
```

문자열 검색

- 문자열 검색

  문자열 안에 들어있는 부분 문자열 검색하는 알고리즘 학습

  - 문자열 정의하기

    문자열(string)은 문자의 나열이라 할 수 있는데, 문자가 하나만 있어도 되고, 심지어 비어 있는 빈 문자열도 문자열이다.

    "apple"처럼 2개의 큰 따옴표로 묶은 것을 문자열 리터럴(literal)이라 한다.

    'a', 'p', 'p', 'l', 'e', '\0' -> 문자열 리터럴의 끝을 나타내기 위해 널문자(null character)를 자동으로 추가한다.

문자열 검색

- 문자열 역순으로 읽기

```
char a1[] = "DOG";
char a2[10];  //충분한 크기 확보
int i;

//a1을 a2에 거꾸로 복사
for (i = 0; i < 4; i++) {
    a2[i] = a1[2 - i];
}
a2[3] = '\0'; //문자열 끝에 널문자 추가

printf("%s를 거꾸로 읽으면 %s\n", a1, a2);
```

```
DOG를 거꾸로 읽으면 GOD
```

문자열 검색

- 문자열 역순으로 읽기

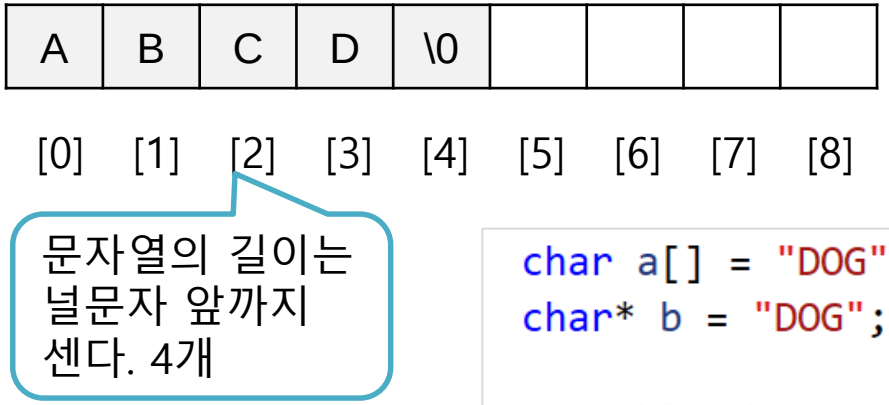
```
int n;
char a[] = "DOG";
char b[10];

n = strlen(a); //3 - a의 개수

for (i = n-1; i >= 0; i--) {
    b[n-1-i] = a[i];
}
b[n] = '\0';

printf("%s를 거꾸로 읽으면 %s\n", a, b);
```

문자열 검색

- 문자열의 길이 구하기

문자열의 길이를 구하는 알고리즘

문자열의 첫 문자부터 널 문자까지 선형(순차) 검색을 하면 된다.




```c
char a[] = "DOG";
char* b = "DOG";

//문자열 길이
printf("%s %d\n", a, strlen(a)); //DOG 3
printf("%s %d\n", b, strlen(b)); //DOG 3
```

문자열 검색

- 문자열의 길이 구하기

```
int str_len(const char* s) {
    int len = 0;

    while (s[len] != '\0') //s[len]도 가능
        len++;
    return len;
}

int str_len2(const char* s) {
    int len = 0;

    while (*s++) //*s++ != '\0'도 가능
        len++;
    return len;
}
```

const 키워드는 문자열이 상수임을 표시함

문자열 검색

- 문자열의 길이 구하기

```c
char a[] = "DOG";
char* b = "DOG";

//문자열 길이
printf("%s %d\n", a, strlen(a));
printf("%s %d\n", b, strlen(b));

//인덱스 검색
printf("%c\n", b[0]);
printf("%c\n", b[1]);
printf("%c\n", b[2]);
```

```c
char str[256];

printf("문자열: ");
scanf("%s", str);

printf("이 문자열의 길이는 %d입니다.\n", str_len(str));
```

```
문자열: computer
이 문자열의 길이는 8입니다.
```

문자열 검색

- 문자열에서 한 문자 검색

```
int str_char(const char* s, int c) {
    int i = 0;
    c = (char)c; //코드값을 문자로 형변환
    while (s[i] != c) { //s[i] == c, 반복종료
        if (s[i] == '\0')
            return -1;  //검색 실패
        i++;
    }
    return i;  //검색 성공
}
```

문자열 검색

- 문자열에서 한 문자 검색

```c
char str[128]; //이 문자열에서 검색
char tmp[128]; //검색할 문자 지정
int ch;        //검색할 문자(코드값)
int idx;       //문자의 위치

printf("문자열: ");
scanf("%s", str);  //문자열 입력

printf("검색할 문자: ");
scanf("%s", tmp); //검색할 문자 입력
ch = tmp[0];      //첫 번째 문자를 검색할 문자로 지정

if ((idx = str_char(str, ch)) == -1)
    printf("문자 '%c'는(은) 문자열에 없습니다.\n", ch);
else
    printf("문자 '%c'는(은) %d번째에 있습니다.\n", ch, idx+1);
```

```
문자열: computer
검색할 문자: p
문자 'p'는(은) 4번째에 있습니다.
```

문자열 검색

- 문자열에서 한 문자 검색 – 공백 문자 허용

```
char str[128]; // 이 문자열에서 검색
char tmp[128];
int ch;        // 검색할 문자
int idx;

printf("문자열: ");
fgets(str, sizeof(str), stdin);  //공백문자 허용

printf("검색할 문자: ");
fgets(tmp, sizeof(tmp), stdin);

ch = tmp[0]; // 첫 번째 문자를 검색할 문자로 지정

if ((idx = str_chr(str, ch)) == -1)
    printf("문자 '%c'는(은) 문자열에 없습니다.\n", ch);
else
    printf("문자 '%c'는(은) %d번째에 있습니다.\n", ch, idx + 1);
```

```
문자열: computer science
검색할 문자: s
문자 's'는(은) 10번째에 있습니다.
```

문자열 검색

- 문자열에서 중복 문자 검색

```c
int str_chr_all(const char* s, int c) {
    int count = 0; // 찾은 개수
    int i;

    for (i = 0; s[i] != '\0'; i++) {
        if (s[i] == (char)c) {
            printf("%d번째 ", i + 1); // 1부터 시작하는 위치 출력
            count++;
        }
    }

    if (count == 0) {
        printf("문자 '%c'는(은) 문자열에 없습니다.", c);
    }
    putchar('\n');

    return count; // 찾은 개수 반환
}
```

문자열 검색

- 문자열에서 중복 문자 검색

```c
char str[128]; // 검색 대상 문자열
char tmp[128];
int ch;        // 검색할 문자
int found;     // 반환값 저장

printf("문자열: ");
fgets(str, sizeof(str), stdin);

printf("검색할 문자: ");
fgets(tmp, sizeof(tmp), stdin);

ch = tmp[0]; // 첫 번째 문자만 검색

printf("문자 '%c'의 위치: ", ch);
found = str_chr_all(str, ch);

printf("총 %d번 등장했습니다.\n", found);
```

```
문자열: hello world
검색할 문자: l
문자 'l'의 위치: 3번째 4번째 10번째
총 3번 등장했습니다.
```

문자열 검색

- 두 문자열 연결하기

```
/*
  - 두 개의 문자열을 연결하여 하나의 문자열로 만들기
    a 배열은 충분히 커야 합니다.
    "smart"(5자) + "phone"(5자) + '\0' = 총 11바이트가 필요
*/
```

```
int i = 0, j = 0;
char a[12] = "smart";
char b[] = "phone";

printf("%s+%s=", a, b);
while (a[i] != '\0')
    i++;
//printf("\ni=%d\n", i); //5
```

문자열 검색

- 두 문자열 연결하기

```
while (b[j] != '\0') {
    a[i] = b[j];
    i++;
    j++;
}
a[i] = '\0';
printf("%s\n", a);

/*
   i=5, a[5]=b[0], smartp
   i=6, a[6]=b[1], smartph
   i=7, a[7]=b[2], smartpho
   i=8, a[8]=b[3], smartphon
   i=9, a[9]=b[4], smartphone
   i=10, a[10] = '\0'
*/
```

```
smart+phone=smartphone
smartphone
```

문자열 검색

- 두 문자열 연결하기 – strcat() 사용

```
/*
  strcart() 함수
  문자열 끝('\0')을 찾아 자동으로 뒤에 붙여주며,
  복사 후 마지막에 '\0'도 추가합니다.
*/
char str1[20] = "smart";
char str2[] = "phone";

strcat(str1, str2);
printf("%s\n", str1);
```