3주차 : 탐색 알고리즘 학습 (선형 탐색, 이진 탐색, DFS, BFS 등)

선형 탐색

> 맨 앞이나, 맨 뒤부터 순서대로 하나하나 찾아보는 알고리즘

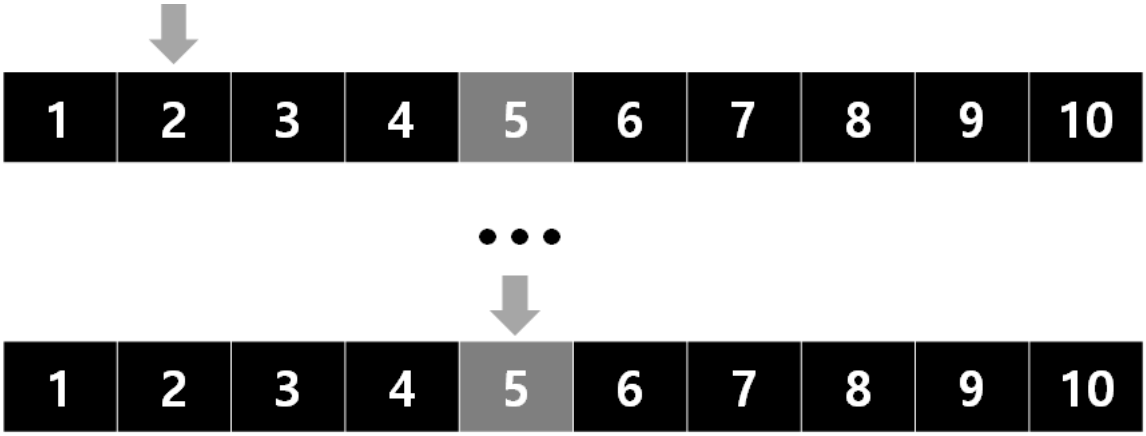


사진 출처 : https://bba-dda.tistory.com/21

- 가장 단순하고 간단한 탐색 알고리즘

과정

1. 맨 끝부터 하나하나 원하는 값을 찾아본다.
2. 원하는 값을 찾으면, 탐색을 종료한다.

시간 복잡도

길이 n짜리의 리스트를 탐색할 때,

최선의 경우

- 리스트의 첫 번째 원소가 정답인 경우 : 1번

최악의 경우

- 리스트의 맨 마지막 원소가 정답이거나, 리스트에 정답이 없을 때 : n번

→ **O(n)**의 시간복잡도

이분 탐색(이진 탐색)

> 이미 정렬되어 있는 배열에서 특정한 값의 위치를 찾는 알고리즘

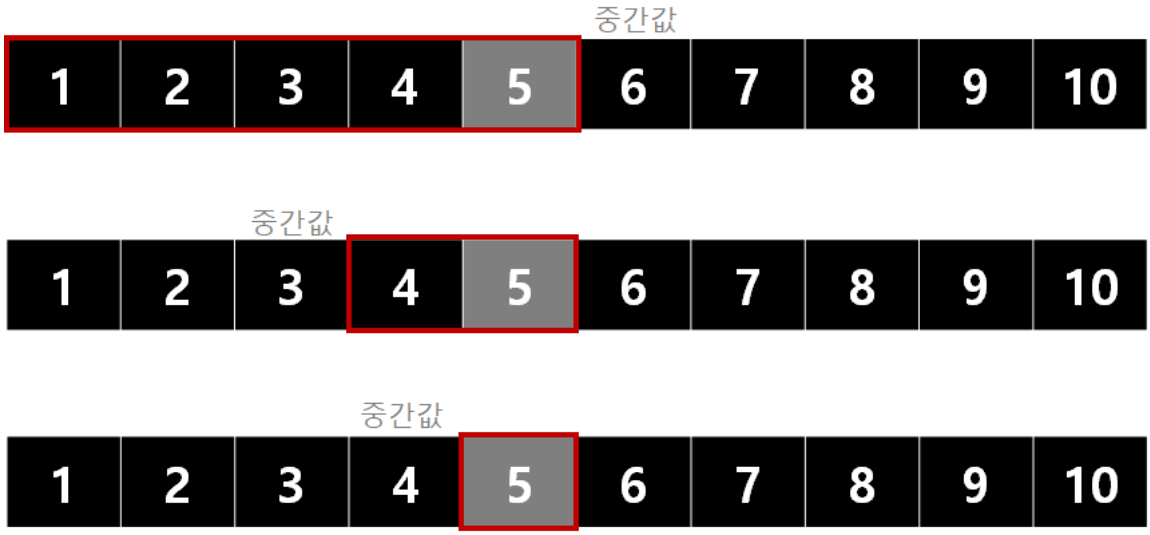


사진 출처 : https://bba-dda.tistory.com/21

- 탐색 범위를 두 부분 리스트로 나눠 절반씩 좁혀가며 원하는 값을 찾는 알고리즘
- 처음 중간의 값을 임의의 값으로 선택하여, 그 값과 찾고자 하는 값의 크고 작음을 비교하는 절차
- 처음 선택한 중앙값이 만약 찾는 값보다 크면 그 값은 새로운 최댓값이 되며, 작으면 그 값은 새로운 최솟값이 됨

과정

1. 중간지점을 선택한 뒤, 중간지점을 기준으로 왼쪽 혹은 오른쪽 부분만 남긴다.
2. 남긴 부분 중에서 다시 중간지점을 선택한 뒤, 왼쪽 혹은 오른쪽만 남긴다.
3. 위 과정을 원하는 값을 찾을 때 까지 반복한다

장점

- 평균적으로 선형 탐색보다 빠름->O(N) (단, 첫번째 요소가 탐색대상이면 선형 탐색이 더 빠름)

단점

- 정렬된 리스트에만 사용할 수 있음

시간 복잡도

길이 n짜리의 리스트를 탐색할 때,

최선의 경우

- 리스트의 중간부분이 정답일 때 : 1번

최악의 경우

- 리스트에 정답이 없는 경우 : log2(n) 번

→ **O(logn)**의 시간복잡도

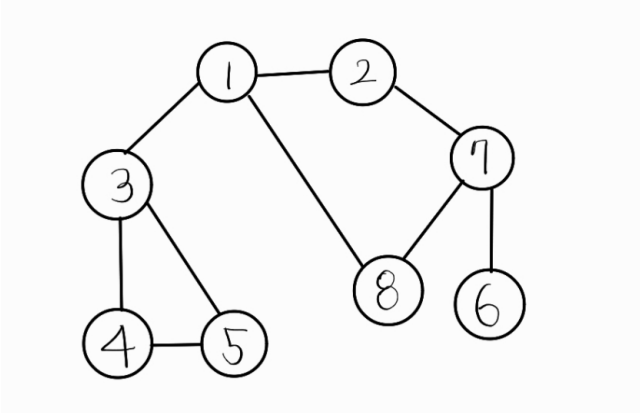
> 💡 보통 정렬된 리스트에는 **이진탐색**을 사용하고, 정렬되지 않은 리스트에는 **선형탐색**을 사용

DFS(Depth First Search, 깊이 우선 탐색)

> 한 경로로 최대한 깊숙하게 들어가서 탐색한 후 다시 돌아가 다른 경로로 탐색하는 방식

특징

- 가장 깊숙이 위치하는 노드에 닿을 때까지 확인(탐색)
- 모든 노드를 방문하고자 하는 경우 이 방법을 선택함
- 재귀함수, Stack을 이용해 구현
- BFS보다 조금 더 간단함
- 검색 속도 자체는 BFS에 비해서 느림



			6		8		
		7	7	7	7	7	
	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1

결과적으로 노드의 탐색 순서(스택에 들어간 순서는)

→ 1 2 7 6 8 3 4 5

과정

1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 한다.
2. 스택의 최상단 노드에 방문하지 않은 인접 노드가 있으면 그 인접 노드를 스택에 넣고 방문 처리를 한다.
   방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼낸다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지 반복한다.

유의할 점

- Stack Overflow (기저조건 잘 설정)

활용

- 백트래킹, 단절선/단절점 찾기, 위상정렬, 사이클 찾기 등

Code

```
// python -> js
const dfs = (graph, v, visited) => {
  // 현재 노드를 방문 처리
  visited[v] = true;
  console.log(v);
```

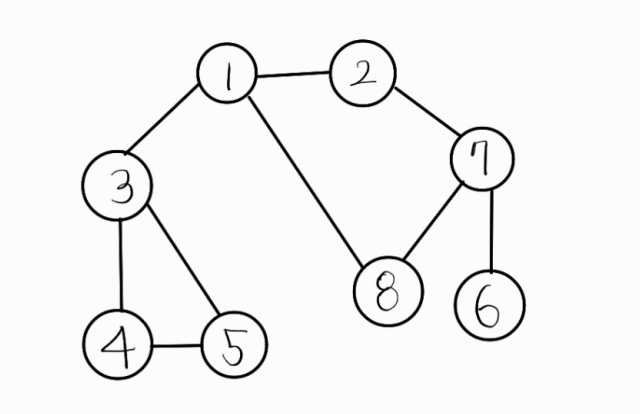
```
  // 현재 노드와 연결된 다른 노드를 재귀적으로 방문
  for (const i of graph[v]) {
    if (!visited[i]) {
      dfs(graph, i, visited);
    }
  }
};

// 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
const graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7],
];

// 각 노드가 방문된 저보를 리스트 자료형으로 표현(1차원 리스트)
let visited = Array(9).fill(false);

// 정의된 DFS 함수 호출
dfs(graph, 1, visited);
```

BFS(Breadth First Search, 너비 우선 탐색)

> 시작 노드에서 시작하여 인접한 노드를 먼저 탐색하는 방식

특징

- 주로 두 노드 사이의 최단 경로를 찾고 싶을 때 사용
- Queue를 이용해 구현



					6		
			5	5	5		
			4	4	4		
		7	7	7			
	8	8	8				
	3	3					
1	2						

결과적으로 노드의 탐색 순서(큐에 들어간 순서는)

→ 1 2 3 8 7 4 5 6

과정

1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다.
2. 큐에서 노드를 꺼내 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리를 한다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지 반복한다.

유의할 점

- 유의할 점 : 메모리 초과 (방문 체크 꼭 해줘야 함)

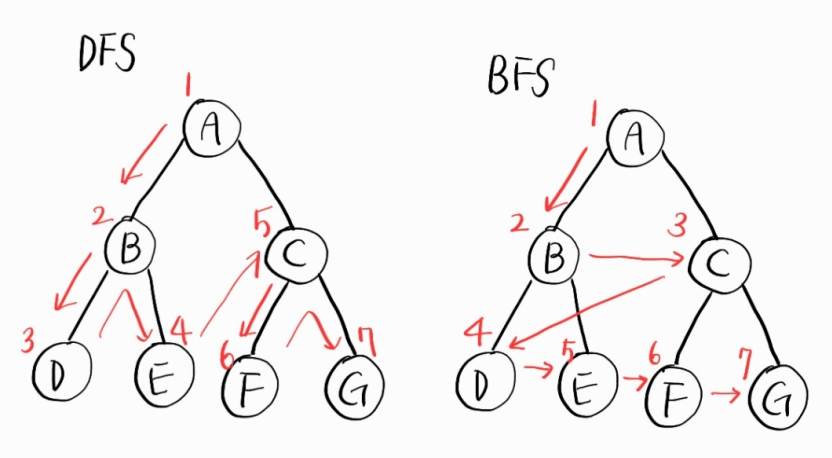
활용

- 최단경로 찾기, 위상정렬 등

Code

```
// python -> js
const bfs = (graph, start, visited) => {
  let queue = [];
  // 현재 노드를 방문 처리
  visited[start] = true;
  queue.push(start);

  // 큐가 빌 때까지 반복함
  while (queue.length !== 0) {
    // 큐에서 원소를 하나 뽑아 출력
    v = queue.shift();
    console.log(v);

    // 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
    for (const i of graph[v]) {
      if (!visited[i]) {
        queue.push(i);
        visited[i] = true;
      }
    }
  }
};

// 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
const graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7],
];

// 각 노드가 방문된 저보를 리스트 자료형으로 표현(1차원 리스트)
let visited = Array(9).fill(false);

// 정의된 BFS 함수 호출
bfs(graph, 1, visited);
```

DFS와 BFS 비교



DFS(깊이우선탐색)	BFS(너비우선탐색)
현재 정점에서 갈 수 있는 점들까지 들어가면서 탐색	현재 정점에 연결된 가까운 점들부터 탐색
스택 또는 재귀함수로 구현	큐를 이용해서 구현

DFS와 BFS의 시간 복잡도

> 두 방식 모두 조건 내의 모든 노드를 검색한다는 점에서 시간 복잡도는 동일함

N은 노드, E는 간선일 때

- 인접 리스트 : O(N+E)
- 인접 행렬 : $O(N^2)$
  -> 일반적으로 E(간선)의 크기가 N^2에 비해 상대적으로 적기 때문에 인접 리스트 방식이 효율적임