

-Root : 트리에서 부모가 없는 유일한 노드, 모든 노드의 조상

-정점(Vertex or Node) : 트리에서 자료를 보관하는 곳

-패스(Link or Edge or Path) : 노드 사이의 길

-Always 정점 + 1 == 간선

-깊이(depth or Height) : 가장 높은 레벨

-Level N-1 (부모) Level N(자신) Level N+1(자식)

-노드의 차수(Degree) : 자식 수

-조상(Ancestors) : 모든 부모

-후손(descendant) : 모든 자식

-단말(Terminal) 혹은 잎(Leaf) : 자식이 없는 노드

-트리의 차수 : 모든 노드 중에서 가장 높은 차수

-서브 트리(Sub Tree) : 자식을 루트로 하는 트리, 서브 트리 개수 == 노드의 차수

-형제(Sibling): 같은 부모를 갖는 노드

binary tree(이진 트리)

\* 각 노드들이 최대 두 개의 자식 노드를 가지는 트리 자료 구조

\* 모든 노드의 차수가 2 이하로 구성하는 트리

포화 이진 트리(Full Binary Tree, 꽉 찬 이진 트리) 혹은 정 이진 트리

\* 마지막 레벨까지 모든 노드가 있는 트리

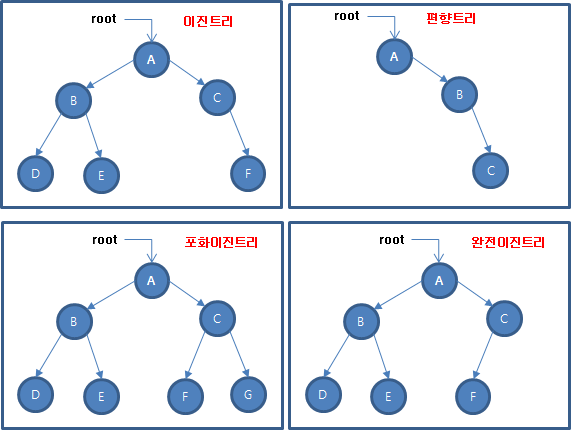
\* 꽉 찬 이진 트리에서 Level N에는 2의 N-1 승의 노드가 존재

완전 이진 트리(Complete Binary Tree) 혹은 전 이진 트리

\* 노드를 삽입할 때 왼쪽부터 차례대로 추가하는 이진 트리

사향 트리 혹은 편향 트리

\* 한 쪽으로 기울어진 트리



Heap(힙)

\* 자료 집합에서 최대값 또는 최소값을 구할 때 빠른 연산을 위해 고안 된   
완전이진트리를 기본으로 한 자료구조

\* 노드 A가 노드 B의 부모 노드이면, A와 B 사이에는 대소 관계가 성립한다.

\* 두가지 종류가 있으며,   
부모 노드의 값이 자식 노드의 값보다 항상 크면 '최대 힙', 항상 작으면 '최소 힙' 이라고 부른다.

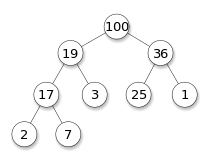
\* 대소 관계는 부모/자식 간에만 성립하며, 형제 사이에는 대소 관계가 정해지지 않는다.

\* 각 노드의 자식 노드의 최대 개수는 힙의 종류에 따라 다르지만,   
대부분의 경우는 자식 노드의 개수가 최대 2개인 이진 힙(binary heap)을 사용한다.

\* 가장 높은(혹은 가장 낮은) 우선순위를 가지는 노드가 항상 뿌리 노드에 오게 되는 특징이 있으며, 이를 응용하면 우선순위 큐와 같은 추상적 자료형을 구현할 수 있다.

\* 이진 힙 : 이진 트리를 구성하는 노드에 key와 element를 저장한 것.  
특징 2가지 :   
1. 최대 힙 or 최소 힙,   
2. 마지막 왼쪽 결합 노드들의 레벨을 제외한 다른 모든 레벨들은 완전 이진트리를 형성

\* 힙 리스트(heap list)로 표현할 때 i번째 노드의 왼쪽 자식 노드의 위치는 2i가 되며, i번째 노드의 오른쪽 자식 노드의 위치는 2i+1이고, 또한 i번째 노드의 부모 노드의 위치는 i/2가 된다.



공간 복잡도 : 알고리즘에서 사용되는 메모리 공간의 총량

시간 복잡도 : 알고리즘에서 사용되는 연산 횟수의 총량

힙의 시간 복잡도

공식을 사용한 방법  
 

원리를 이용한 방법

리프 레벨(깊이) == 시간 복잡도   
여기서 리프 레벨이 시간 복잡도가 되는 이유는 힙 트리 특성을 생각해 보면 된다.  
힙 트리는 최대 힙 or 최소 힙 2가지 형태가 존재한다.  
여기서 최대 힙 이라고 가정을 하고 하나의 노드를 추가 한다고 생각해보자.   
추가 되는 노드는 항상 마지막 leaf 레벨의 끝에 위치가 된다. 여기서 추가된 노드가 자기 자리를 최대 힙 조건에 맞게 찾으려면 부모 노드를 확인해야 하고 최악의 경우 깊이(h) 만큼 수행하게 된다. 따라서 시간 복잡도(특정 일의 반복 횟수)를 h번 만큼 수행해야 하는 것이다.  
그렇기 때문에 h = 시간복잡도가 되는 것이다.

우선순위 큐 (compactHeap.py 참고)

\* 데이터들이 우선순위를 갖는다.

\* 입력순서에는 상관없이 우선순위가 높은 데이터가 가장 먼저 처리된다.

\* 우선순위 큐를 힙을 응용하여 구현하는 형태로 사용할 수 있다. 소스 참고

완전탐색, 브루트포스(Brute-force)

\* 가능한 모든 경우를 탐색하는 방법

\* 시간 복잡도가 최대로 들어간다.

\* 탐색 방법 종류

Brute Force : for문과 if문을 이용하여 처음부터 끝까지 탐색하는 방법

**비트마스크**  
-더 빠른 수행 시간  
-더 간결한 코드  
-더 작은 메모리 사용량  
-연관 배열을 배열로 대체 : boolean으로 체크하는 배열을 비트마스크를 써서 int[]로 나타낼 수 있다. 큰 시간과 메모리 차이를 불러온다.

비트마스크 생성

bitmask = []

for n in range(0,8,1):  
bitmask.append(1<<n)



**순열** : 순열의 시간 복잡도는 O(N!)

\* 서로 다른 n개의 대상에서 r개를 뽑아 일렬로 배열한 것을 말하고 그 경우의 수는 nPr로 표현하는 것이다. 알고리즘 자체를 구현하기 보다는 itertools.permutation 모듈을 써서 언제든지 유동적으로 사용할 수 있는 힘을 기르는 것이 맞다고 판단 된다. 시간이 많이 나면 직접 구현정도는 한번 해보자.

**조합**

\* n개의 대상에서 r개를 뽑기. 중복 제외(포함)(문제에 따라서) 순서를 고려하지 않는다.((a,b),(b,a)) 이것 또한 itertools의 모듈을 사용하자.

**백트래킹(Backtracking)**

\* 가능한 모든 방법을 탐색하는 방법

\* DFS가 있다. DFS는 현재 지점에서 방문할 곳이 있으면 재귀 호출을 이용해서 계속 이동한다.   
장점은 무한히 깊은 곳을 찾아야할 때 효과적이다. 그리고 모든 곳을 방문하기에 비효율적이다.

\* 이와 같은 DFS의 비효율적인 경로를 차단하고 목표지점에 갈수 있는 가능성이 있는 루트를 검사하는 방법이 백트래킹 알고리즘이다.

\* 백트래킹은 DFS에 가지치기 (Pruning) 를 통해 가도 되지 않는 루트는 고려하지 않고 탐색하는 완전탐색 기법이다.

\* 배제, 풀이 시간 단축

\* 대표 nqueen 문제

**BFS, DFS**

\* 주요 자료형 set() - 순서가 없고 중복을 허용하지 않는다는 특징을 가진다.

\*set1 = set([1,2,3])   
set2 = set("Conor Mcgregor")

set([1, 2, 3])  
set([' ', 'C', 'e', 'g', 'M', 'c', 'o', 'n', 'r'])

\*주요 개념 {}, [], () 차이  
{} : Dictionary -> hashTable  
[] : array -> 배열  
() : Tuple -> 딕과 차이는 (1,2,3,4), (“a”,”b”,”c”)와 같이 같은 타입의 원소를 저장할 때 쓰인다.

정렬(코드 참고)

| **Name** | **Best** | **Worst** | **Stable** | **Memory** |
| --- | --- | --- | --- | --- |
| 버블정렬 | n | n^2 | True | 1 |
| 선택정렬 | n^2 | n^2 | False | 1 |
| 삽입정렬 | n | n^2 | True | 1 |
| 셸정렬 | nlogn | (best) nlog^2n | False | 1 |
| 병합정렬 | nlogn | nlogn | True | nn |
| 퀵정렬 | nlogn | nlogn ~ n^2 | False | logn ~ n |

**버블정렬**

\* 이웃한 두 값을 비교하여 정렬한다.

\* 큰 값이 오론쪽으로 이동하는 과정이 반복되면서 비교했던 모든 값들의 최댓값이  
 맨 오른쪽으로 옮겨지게 된다.

최악의 경우 (n−1)+(n−2)+⋯+1번 비교가 이루어지므로 O(n2)이다. 그러나, 데이터가 잘 졍렬돼있다면 O(n)이므로 데이터의 정렬 여부를 파악하기 위한 알고리즘으로 사용될 수 있다. Short Bubble sort는 데이터 정렬이 완료되면 early stopping한다.

**선택정렬**

\* 주어진 배열에서 최댓값(최솟값)을 찾아 맨 오른쪽(왼쪽)값과 교체한다.

\* 최대값을 맨 오른쪽으로 보내는 점은 버블정렬과 비슷하지만 이웃한 두 값을 정렬하는 과정이 없기 때문에 대체로 버블정렬 보다 빠르다.

\* 최댓값을 찾아야 하므로 정렬 상태에 관계없이 언제나 O(n^2)이다.

**삽입정렬**

\* 아직 정렬되지 않은 값을 이미 정렬된 배열 사이에 끼워 넣는 과정을 반복한다.

\* 여전히 O(n2)이지만 평균적으로 삽입정렬이 선택정렬과 버블정렬에 비해 빠르다.

\* 버블정렬과 마찬가지로 데이터가 이미 정렬되어 있다면 O(n)이다. 그러나, 데이터가 역순으로 정렬된 상태라면 삽입을 위해 값을 하나씩 뒤로 밀어내는 과정을 아주 많이 반복해야 하므로 느리다.

**셸 정렬**

\* 삽입 정렬이 거의 정렬된 배열에서 최적의 성능을 냄과 동시에 값 하나씩 위치를 결정하여 비효율적이라는 점에서 착안되었다.

\* 셸 정렬은 주어진 간격만큼 듬성듬성 떨어진 서브배열을 만들어 삽입정렬을 수행한다.

\* 간격이 3이라면 3개의 서브배열이 만들어진다. 모든 서브배열에 대해 삽입정렬을 마쳤다면, 간격을 (보통 절반으로) 줄여 반복한다.

\* 간격이 1이 되면 거의 정렬이 된 상태이므로 빠르게 정렬할 수 있다.

\* 간격 정의에 따라 성능이 제각각이라 시간복잡도 분석이 쉽지 않다. 배열이 이미 정렬되어 있다면 O(nlogn)이고 최악의 경우 아래 구현처럼 간격을 절반씩 줄인다면 O(n^2)이다. 다른 간격 정의를 사용한다 하더라도 현재까지 알려진 바로는 O(nlog^2n)이 최선이다.

**병합 정렬**

\* 폰 노이만이 개발했으며, 두 부분으로 쪼개는 작업을 재귀적으로 반복한 뒤, 쪼갠 순서의 반대로 작은 값부터 병합해나가는 분할 정복 알고리즘의 일종이다.

\* 두 부분으로 쪼개는 데 O(logn) (이진탐색 참고)이고, 데이터 병합이 O(n)이므로, 정렬 상태와 무관하게 언제나 O(nlogn)이다. 데이터 크기만한 메모리가 더 필요한 게 단점이다.