Implementation Method	Time Complexity
Adjacency matrix, linear search	$\Theta(\|V\|^2)$
Adjacency list, binary heap	$\Theta(\|E\|\log(\|V\|))$

Prim's Algorithm

1) dense graph [E = V^2] 2) sparse graph // V^2logV^2 is slower so not always faster with heap.
** For dense G, nested-loop Prim & For sparse G, Kruskal

time complexity of Kruskal's algorithm expressed as $\Theta(|E|\log(|V|))$ instead of $\Theta(|E|\log(|E|))$.

Collision Resolution Techniques
1) **Separate Chaining** (Linked List)
2) **Linear, Quadratic Probing**
3) Double Hashing

Operation	Average-Case Complexity
Insert Key	$\Theta(1)$ if duplicate keys are allowed, $\Theta(N/M)$ if duplicates not allowed
Search for Key	$\Theta(N/M)$
Erase Key	$\Theta(N/M)$

- Separate chaining
  - Best, worst, average
  - O(1), O(n), O(n/m) ≈ O(1)
- Linear probing
  - Best, worst, average
  - O(1), O(n)
  - Average has complicated formula in slides, not needed for exam
- Search complexity is the same as insert

Load Factor (α) = N / M = average # of items in each list (separate chaining)
// N : number of keys , M : size of table

Number of Probes required to search for existing element in hash table
: $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
" unsuccessfully search for non-existing / insert
: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Depth : root is 1, Height : leaf is 1
Binary Search Tree : left side smaller, right side bigger

Operation	Complexity
Insert Key (Best Case)	$\Theta(1)$
Insert Key (Worst Case)	$\Theta(n)$
Remove Key (Worst Case)	$\Theta(n)$
Find Parent	$\Theta(1)$
Find Child	$\Theta(1)$
Space Required (Best Case)	$\Theta(n)$
Space Required (Worst Case)	$\Theta(2^n)$

Array based

Operation	Complexity
Insert Key (Best Case)	$\Theta(1)$
Insert Key (Worst Case)	$\Theta(n)$
Remove Key (Worst Case)	$\Theta(n)$
Find Parent	$\Theta(n)$
Find Child	$\Theta(1)$
Space Required (Best Case)	$\Theta(n)$
Space Required (Worst Case)	$\Theta(n)$

Pointer based

1) Preorder Traversal

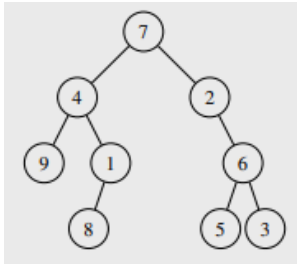
```
void preorder(Node* p) {
  if (!p) return;
  process(p->val);
  preorder(p->left);
  preorder(p->right);
} // preorder()
```

2) Inorder Traversal

```
1  void inorder(Node* p) {
2    if (!p) return;
3    inorder(p->left);
4    process(p->val);
5    inorder(p->right);
6  } // inorder()
```

3) Postorder Traversal

```
1  void postorder(Node* p)
2    if (!p) return;
3    postorder(p->left);
4    postorder(p->right);
5    process(p->val);
6  } // postorder()
```



1) 7, 4, 9, 1, 8, 2, 6, 5, 3 (first one = root)
2) 9, 4, 8, 1, 7, 2, 5, 6, 3 (looks like tree with this order)
3) 9, 8, 1, 4, 5, 3, 6, 2, 7 (last one = root)

Binary Search Tree Delete

1) Inorder predecessor : Replacing with right most node in the left subtree
2) Inorder successor : Leftmost node in the right subtree // If tree is balanced -> worst O(logn) = height

Summary of Binary Search Tree Time Complexities

Operation	Best-Case Time	Average-Case Time	Worst-Case Time
Finding a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Inserting a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Deleting a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$

What is the worst-case complexity of a search operation on this tree in terms of h and/or n?
The worst-case complexity of search, insert and remove is always the height of the tree, regardless of whether it is balanced

Graph Representation	Adjacency List	Adjacency Matrix
BFS Time Complexity	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$

Graph Representation	Adjacency List	Adjacency Matrix
DFS Time Complexity	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$

AVL Tree

: balance factor range [-1, 1] = Height of left – Height of right ,Zigzag -> Rotate twice

Summary of AVL Tree Time Complexities

Operation	Best-Case Time	Average-Case Time	Worst-Case Time
Finding a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Inserting a value	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Deleting a value	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

of a sorting algorithm implemented using an AVL tree?
$\Theta(n \log n)$ in all cases

```
int number_of_tilings(int n) {
  vector<int> memo(n+1);
  memo[0] = 1
  memo[1] = 1
  for(int i=2; i<=n; i++)
    memo[i]= memo[i-1]
    + 2 * memo[i-2];
  return memo[i];
}
```

Recurrence :
Take all sol. till i-1 and one 2x1 tile
Take all sol. till i-2 and one 2x2 tile or two 1x2 tile
f(i) = f(i-1) + 2*f(i-2)

7. BST Construction Zone Ⓐ Ⓑ Ⓒ Ⓓ Ⓔ

Which of the following statements about constructing a binary search tree is **FALSE**?

A) Given a sorted range of n elements, the optimal algorithm can construct a binary search tree in $\Theta(n)$ time

B) Inserting n elements in sorted order would require $\Theta(2^n)$ space if the binary search tree is implemented using a vector

C) Given a sequence of n elements, randomizing the insertion order of these elements will guarantee that the maximum depth of the resulting BST is strictly less than n

D) If n elements are inserted in sorted order into a pointer-based binary search tree, then the total time complexity of deleting all of the elements in the order of insertion is $\Theta(n)$

E) Given a sequence of n distinct elements, it is possible for multiple insertion orders of these elements to produce the same binary search tree

Greedy Algorithms and Divide and Conquer

1) **Brute Force** : every case

2) **Greedy** : sequence of locally optimal choices (Can be proved by induction)
Activity Selection, Breakpoint selection,

3) **Backtracking** : Similar to BF but stops when constraints does not satisfy. (Promising)
Combinations, NQueens, Sudoku

Backtracking / Branch&Bound

- Two related classes of algorithms
  - The code looks similar
- They solve **different types of problems.**
- ***Backtracking*** *is used to answer this question:*
  - **"What choices should I make to satisfy some constraints?"**
- ***Branch+Bound*** *is used to answer this question:*
  - **"What choices should I make to obtain the *best* solution?"**

4) **Branch and Bound** : Similar to Backtracking but for optimization and minimization problem. While backtracking is depth first search, it is breath first search.
1. Start with an infinity bound
2. First complete solution – cost as an upper bound
3. Measure each partial solution and calculate a lower bound estimate
4. If lower bound exceed the current upper bound, prune
5. Lower cost -> new upper bound
6. When search is done, the current **UPPER BOUND** will be a minimal solution

In summary, to obtain a lower bound for the TSP problem, you can sum of the following four weights:

1. The cost of the current partial solution.
2. The cost of the MST connecting the remaining points not in the partial solution.
3. The cost of connecting one end of the partial solution with its closest point in the MST.
4. The cost of connecting the other end of the partial solution with its closest point in the MST.

This will always produce an optimistic estimate on the remaining cost, so it can be used as a valid lower bound. Notice that the estimate will likely not be a valid solution, since an MST is not guaranteed to produce a Hamiltonian cycle on the remaining points! However, this is okay, since this estimate is only used to determine whether a partial solution *could* lead to an optimal solution, and not what the actual optimal solution is. This method of estimating a lower bound is useful because it not only finds a lower bound close to the ideal, but it also can be done rather efficiently: the cost of finding an MST is significantly cheaper than exploring all remaining permutations of a partial solution!

TSP problem
Lower bound :
current partial + MST + cost of connecting MST & partial

Upper bound :
first complete solution

TSP problem,

Dynamic Programming

Fibonacci time complexity -> Memo size

Protip

Follow these steps for an easy DP

1. What is the recursive solution to this problem?
2. How many possible inputs are there? (Size of memo)
3. Where can I use the saved information?

Knapsack Problem

0-1 Knapsack problem: (Dynamic, B&B, Greedy)
The brute force solution has a lower space complexity than the top-down dynamic programming (Quadratic space) while linear
The best complexity of greedy is O(MN) where M is the capacity of the knapsack, N is the # of available items

- Brute Force
  - n*2^n, optimal
- Greedy
  - O(n), not optimal
- Dynamic Programming
  - O(nC), optimal
  - Knap(int n, int Capacity) = max(knap(n-1, Capacity), knap(n-1, Capacity-cost[n]) + value[n])
  - Requires integer weights, why?
- Backtracking
  - Constraint, doesn't work that well
- Branch and Bound
  - Initial estimate of best solution is lower bound on knapsack value
  - Need to estimate upper bound on remaining value of partial solution

Fractional Knapsack:
Greedy O(nlogn)

Basic Hashing FI[1]=0, FI[2]=1, ...

```
int max_repeated_distance(const vector<int> &vec) {
    unordered_map <int, int> firstIndices;
    int result = 0;
    for(int i=0; i<vec.size(); i++){
        if(firstIndices.find(vec[i]) == firstIndices.end()){
            firstIndices[vec[i]] = i;
        }
        else {
            result = max(result, i - firstIndices[vec[i]]);
        }
    }
    return result;
}
```

```
Node * trim_BST(Node *root, int min_val, int max_val) {
if(!root){                    min val 보다 작으면 BST에서 제거,
    return root;
}
root->left = trim_BST(root->left, min_val, max_val);
root->right = trim_BST(root->right, min_val, max_val);
if( root->val < min_val){
    Node * right_child =root->right;
    delete root;
    return right_child;
}
if(root->val > max_val){
    Node * left_child= root->left;
    delete root;
    return left_child;
}
return root;
```

```
vector<int> right_side_view(Node *root) {
    vector<int> rhs_view;
    traverse(root, rhs_view, 0);
    return rhs_view;
}

void traverse(Node *root, vector<int> & rhs_view, int level)
    if(!root) return;
    if(rhs_view.size()== level)
        rhs_view.push_back(root->val);
    traverse(root->right, rhs_view, level+1);
    traverse(root->left, rhs_view, level+1);
```

```
int count_possible_moves(int num_tiles, int num_moves, int start_pos) {
```

s=1 6tiles 3move

num moves \ # of tiles	0	1	2	3	4	5	6	7
0	0	1	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0
2	0	2	2	1	0	0	0	0
3	0	4	5	3	1	0	0	0

: # of ways to get tile# from s.p. 1

$\therefore f(nm, nt) = f(nm-1, nt-1) + f(nm-1, nt) + f(nm-1, nt+1)$

```
memo[m][t] = memo[m-1][t-1] + memo[m-1][t]
           + memo[m-1][t+1]

vector<vector<int>> memo =
vector<vector<int>>(num_moves+1, vector<int>(num_tiles+2, 0));
memo[0][start_pos] = 1;
for(int m=1; m<=num_moves; m++)
    for(int n=1; n<=num_tiles; n++)
        memo[m][n] =
return memo[num_moves][start_pos];
```

{1,2,5,6} {2,3,4,5}

```
int knapsack(int C, const std::vector<int> value, const std::vector<int> weight);
```

Capacity = 8, n = 4 (# of items)

v	w	i\W	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	(5)	5	6	7	7
6	5	4	0	0							

Values

$K[i,w] = \max\{K[i-1,w], K[i-1, w-w[i]] + v[i]\}$

$K[3,4] = \max\{K[2,4], K[2,4-4] + v[3]\} = \max\{2, 0+5\}$

주사위 합 = target N주사위개수 M면수 X타겟

```
int find_ways(int M, int N, int X) {
    vector<vector<int>> memo (N+1, vector<int> (X+1));
    for(int j=1; j<= M && j <= X ; j++) {
        memo[1][j] = 1;
    }
    for (int i=2; i<=N; i++) {
      for(int j = 1; j <= X; j++){
        for(int k=1; k<=M && k<j; k++){
          memo[i][j] += memo[i-1][j-k];
        }
      }
    }
    return memo[N][X];
}
```

```
bool can_jump_out(vector<int> &nums) {   끝으로 점프 가능?  [2,3,1,1,4]
    if(!nums.empty() && !nums[0]) {
        return nums.size() == 1;
    }
    if( nums.size() > 1) {
        vector<int> dp (nums.size(), nums[0]);
        for(int i=1; i<nums.size()-1; i++) {
            dp[i] = max(nums[i], dp[i-1]-1);
            if(!dp[i]){
                return false;
            }
        }
    }
    return true;
}
```

```
int knapsack(int C, const vector<int>& value, const vector<int>& weight) {
    vector<vector<int>> K(value.size() + 1);

    for (size_t i = 0; i <= value.size(); i++)
    {
        for (int w = 0; w <= C; w++)
        {
            if (i == 0 || w == 0)
                K[i].push_back(0);
            else if (weight[i - 1] <= w)
                K[i].push_back(max(value[i - 1] + K[i - 1][w - weight[i - 1]], K[i - 1][w]));
            else
                K[i].push_back(K[i - 1][w]);
        }
    }

    return K[value.size()][C];
}
```