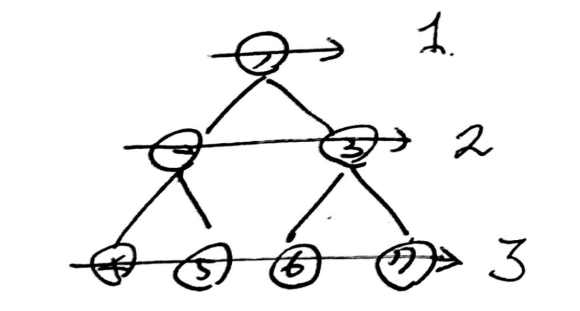
[[알고리즘] 너비 우선 탐색(BFS : Breadth First Search) 알고리즘](https://dheldh77.tistory.com/entry/%EB%84%88%EB%B9%84-%EC%9A%B0%EC%84%A0-%ED%83%90%EC%83%89BFS-Breadth-First-Search-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)

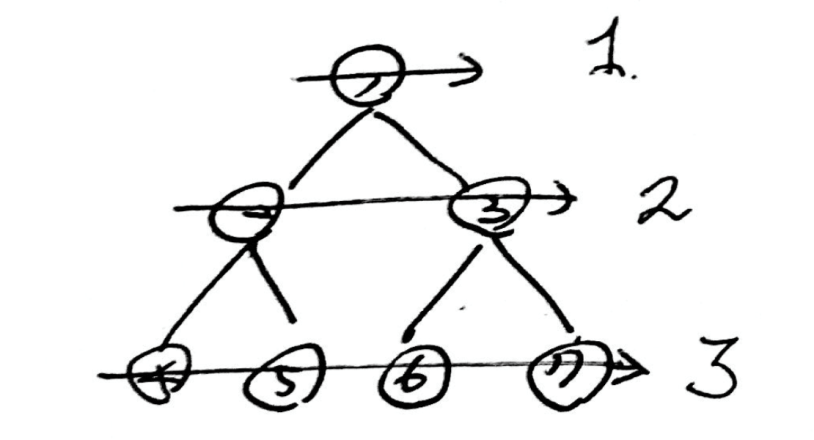
**너비 우선 탐색 알고리즘이란?**

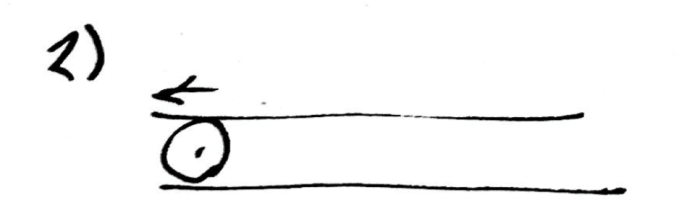


한 노드에서 인접한 노드을 우선으로 탐색하는 방법으로 그림과 같은 그래프가 주어졌을 때, 넓게 탐색한다.

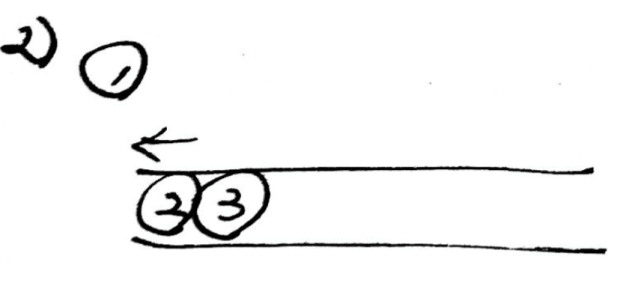
BFS는 큐로 구현할 수 있고 주로 두 노드 간의 최단 경로를 찾을 때 사용할 수 있다.

**너비 우선 탐색 과정**

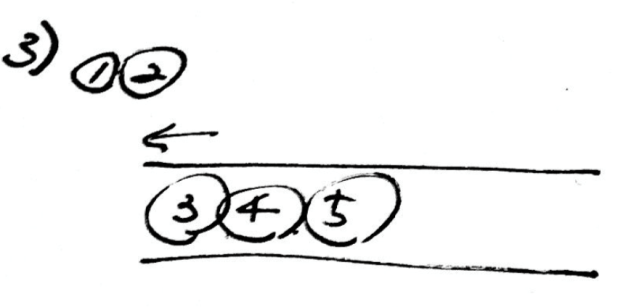




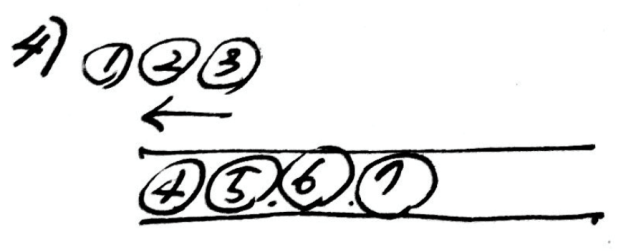
1) 시작 노드를 큐에 넣는다.



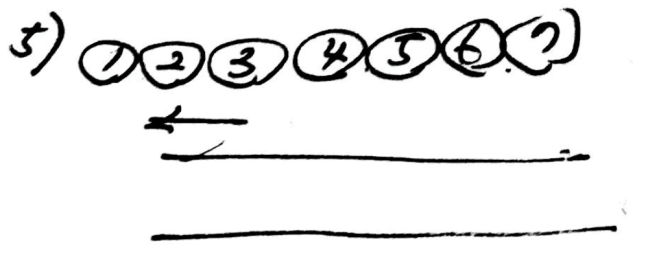
 2) 큐에서 노드 하나를 꺼내고 꺼낸 노드와 인접한 노드를 모두 큐에 넣는다



 3) 다시 큐에서 노드 하나를 꺼내 인접한 노드를 큐에 모두 넣는다.



4) 큐에 더 이상 노드가 없을 때까지, 즉, 인접한 노드가 존재하지 않을 때까지 반복한다.



**너비 우선 탐색 구현**

너비 우선 탐색을 소스코드로 구현하기 위해서는 먼저 그래프를 인접행렬로 만들어주어야한다.

**인접행렬?**

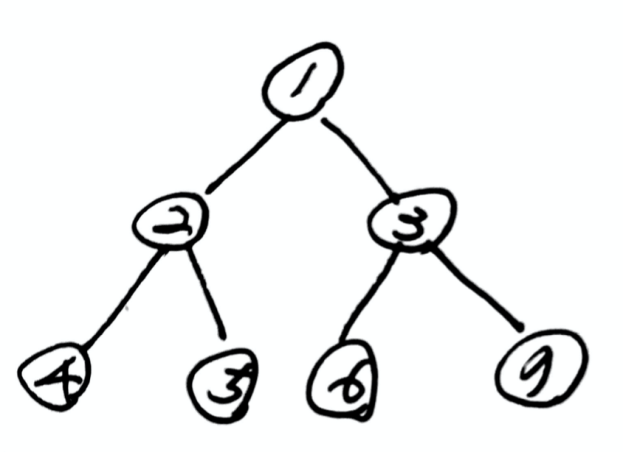
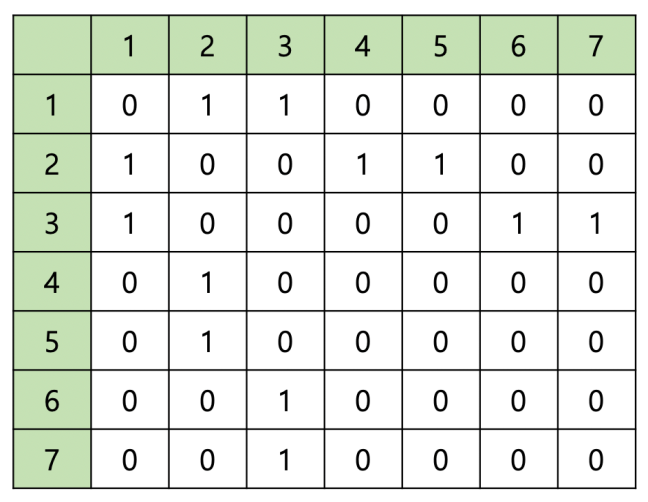
인접행렬이란 그래프가 주어졌을 때, 각 노드와의 연결관계를 이차원 행렬로 나타낸 것이다.

인접행렬을 a[n][m]이라고 할 때,

a[i][j]는 노드 i에서 노드 j로가는 간선을 의미하며 간선이 있을 경우 1, 없을 경우 0으로 나타낸다.

만약, 그래프가 가중치 그래프라면 1대신 가중치 w를 넣어준다.

즉, 두 노드 사이에 간선이 존재하면 1, 존재하지 않으면 0으로 이루어진 2차원 배열을 만든다.

그래프인접행렬

**JAVA**

package algorithm;

import java.util.\*;

public class bfs {

static int n;

static int[][] map;

static boolean[] visit;

public static void bfs(int x) {

Queue <Integer> q = new LinkedList<>();

q.add(x);

visit[x] = true;

while(!q.isEmpty()) {

x = q.poll();

System.out.print((x + 1) + " ");

for(int i = 0; i < n; i++) {

if(map[x][i] == 1 && visit[i] == false) {

q.add(i);

visit[i] = true;

}

}

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

n = sc.nextInt();

map = new int[n][n];

visit = new boolean[n];

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

map[i][j] = sc.nextInt();

}

}

bfs(0);

}

}

[[알고리즘] 되추적(Backtracking) 알고리즘](https://dheldh77.tistory.com/entry/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EB%90%98%EC%B6%94%EC%A0%81Backtracking-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)

**되추적(Backtracking) 알고리즘**

 되추적 알고리즘은 핵심은 '배제'와 '풀이 시간 단축'이다. 조건을 만족하는 모든 경우의 한에서 모든 조합을 탐색한다.

되추적 알고리즘은 적용할 수 있는 대표적인 완전 탐색 알고리즘은 깊이 우선탐색(DFS)이다.

 깊이 우선 탐색(DFS)는 현재 노드에서 방문해야 하는 곳까지 재귀나 스택을 이용해 단말 노드에 도착할 때 까지 깊이 곳을 탐색하는 방법이다. 하지만 DFS의 경우 모든 노드를 방문하기 때문에 굳이 방문해도 되지 않을 노드까지 방문하게 되고 이로인해 비효율을 초래한다.

 되추적 알고리즘은 이러한 비효율을 해결한다. 노드를 방문함에서 있어서 문제 해결에 가능성이 있지 않은 노드는 배제 함으로써 비효율을 해결한다. 즉, 되추적 알고리즘은 DFS에 Pruning을 통해 가능성이 없는 노드를 배제하고 탐색하는 완전 탐색 방법이다.

되추적 알고리즘은

 1. 깊이 우선 탐색 시작

2. 각 노드가 유망한지 검사

3. 유망하다면 탐색을 계속해서 진행

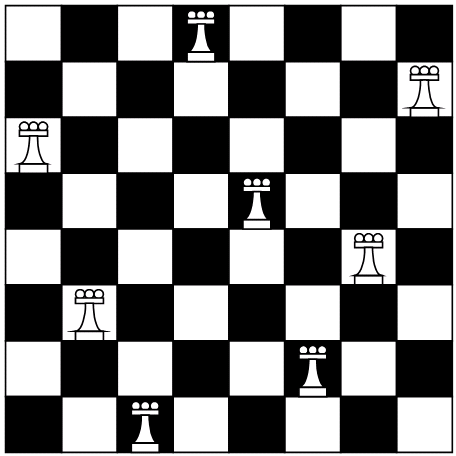
4. 유망하지 않다면 부모 노드로 돌아가서 탐색을 계속 진행(백 트레킹)

 의 순서로 진행된다.

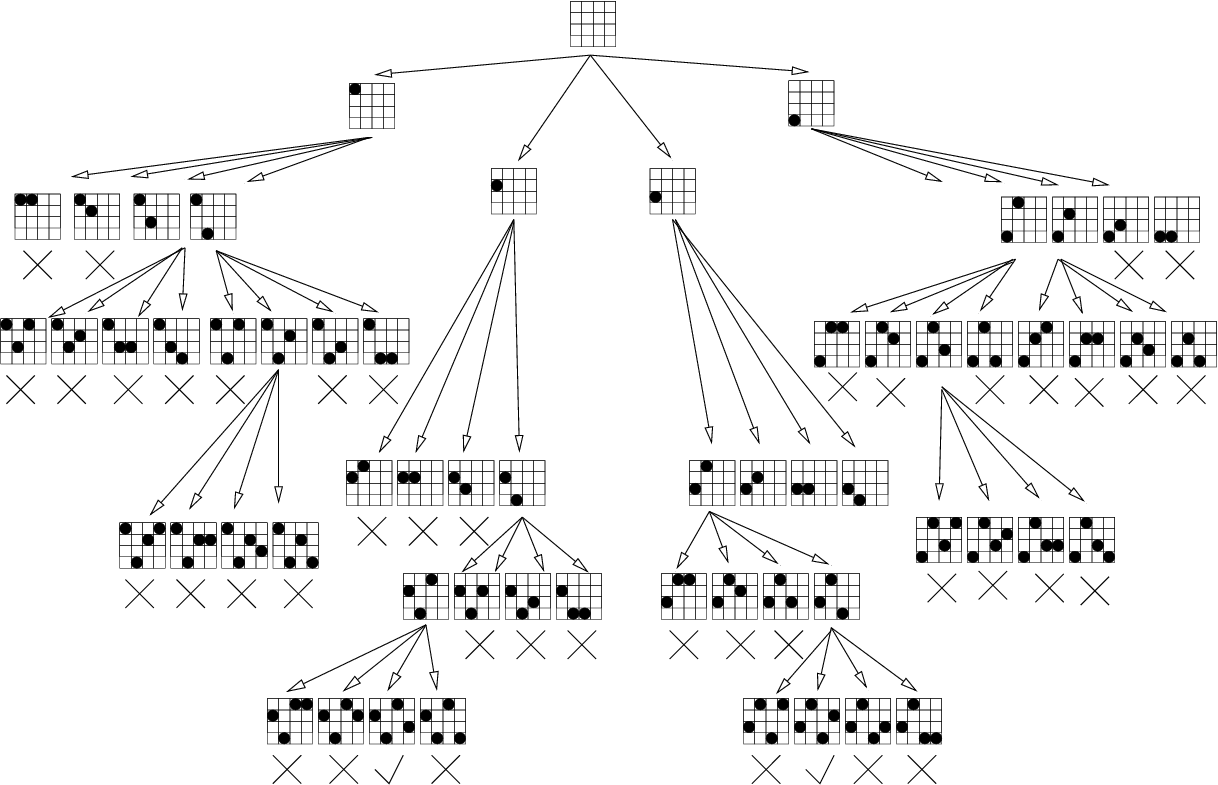
**되추적(Backtracking ) 과정 - N-Queen**

dfs와 되추적 알고리즘을 이용해 해결할 수 있는 대표적인 문제는 n-queen 문제가 있다.

n-queen 문제란 n x n 크기의 체스판에서 가로 세로 대각선으로 움직일 수 있는 n개의 queen을 체스판에 올려둘 때, n개의 queen이 서로 다른 queen을 공격할 수 없는 경우의 수를 구하는 문제이다.



위의 그림은 n = 8일 때 가능한 경우 중 하나이다.



원래 DFS에서 모든 경우를 탐색했다면 위의 그림은 다르게 나타났을 것이다. 모든 경우를 전부 탐색했을 것이고, n이 커질수록 탐색해야 하는 양은 엄청나게 증가할 것이다. Back Tracking을 이용해 유망한 Node 를 검사하고 만약 유망하지 않다면 더 이상 깊이 탐색을 진행하지 않고 부모 Node 로 Back Tracking 하게 된다.

**너비 우선 탐색 구현**

package algorithm;

import java.util.\*;

public class nQueens {

static int cnt, n;

static int[] col;

public static boolean isPossible(int c) {

// 이전까지 입력된 열 중에 같은 행이 존재하는 지 확인

for(int i = 1; i < c; i ++) {

if(col[i] == col[c])

return false;

if(Math.abs(col[i] - col[c]) == Math.abs(i - c))

return false;

}

return true;

}

public static void dfs(int row) {

if(row == n) {

cnt++;

}

else {

for(int i = 1; i <= n; i++) {

col[row + 1] = i;

if(isPossible(row + 1)) {

dfs(row + 1);

}else {

col[row + 1] = 0;

}

}

}

col[row] = 0;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

n = sc.nextInt();

for(int i = 1; i <= n; i++) {

col = new int[n + 1];

col[1] = i;

dfs(1);

}

System.out.println(cnt);

}

}

[[알고리즘] 유니온파인드(Union-Find) 알고리즘](https://dheldh77.tistory.com/entry/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%9C%A0%EB%8B%88%EC%98%A8%ED%8C%8C%EC%9D%B8%EB%93%9CUnion-Find-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)

[Union-find](https://dheldh77.tistory.com/tag/Union-find), [알고리즘](https://dheldh77.tistory.com/tag/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98), [유니온파인드](https://dheldh77.tistory.com/tag/%EC%9C%A0%EB%8B%88%EC%98%A8%ED%8C%8C%EC%9D%B8%EB%93%9C)

**Union-Find 알고리즘이란?**

Union-Find 알고리즘은 그래프 알고리즘 중 하나로

 서로소 집합, 상호배타적 집합(Disjoint-Set)알고리즘이라고도 불린다.

 여러 노드가 존재할 때, 선택한 두 노드가 서로 같은 그래프에 속하는지 판별하는 알고리즘이다.

**Union-Find 과정**

**1) Init**

for(int i = 0; i < parent.length; i++) parent[i] = i;

초기화. 최초 노드는 자기자신을 루트노드로 가진다.

**1) Find(x)**

// x가 속한 그래프의 루트 노드를 반환

static int find(int x) {

// x의 루트노드가 x일 경우. 즉, x와 연결된 노드가 없을 경우.

if(x == parent[x]) return x;

// 재귀를 이용해 부모노드 -> 부모노드 ->...->루트노드의 방식으로 루트노드를 찾아 반환한다.

return find(x);

}

x가 속한 집합(그래프)의 root 노드를 반환한다.

**2) Union(x, y)**

// x와 y가 속한 두 그래프를 하나로 합친다.

static void union(int x, int y) {

// 두 노드의 루트노드를 찾는다.

x = find(x);

y = find(y);

// 두 루트노드 중 원소 값이 더 작은 루트노드를 그래프의 루트노드로 선택한다.

if(x > y) parent[x] = y;

else parent[y] = x;

}

x가 속한 집합(그래프)와 y가 속한 집합(그래프)를 합친다.

**너비 우선 탐색 구현**

package baekjoon;

import java.util.\*;

import java.io.\*;

public class BOJ\_2606 {

static int[] parent;

public static int find(int x) {

if(x == parent[x]) return x;

int y = find(parent[x]);

parent[x] = y;

return y;

}

public static void union(int x, int y) {

x = find(x);

y = find(y);

if(x != y) parent[y] = x;

}

public static void main(String[] args) throws IOException {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

StringTokenizer tk = new StringTokenizer(br.readLine());

int v = Integer.parseInt(tk.nextToken());

tk = new StringTokenizer(br.readLine());

int e = Integer.parseInt(tk.nextToken());

int cnt = 0;

parent = new int[v + 1];

for(int i = 1; i <= v; i++) {

parent[i] = i;

}

for(int i = 0; i < e; i++) {

tk = new StringTokenizer(br.readLine());

int x = Integer.parseInt(tk.nextToken());

int y = Integer.parseInt(tk.nextToken());

union(x, y);

}

for(int i = 2; i <= v; i++) if(find(1) == find(i)) cnt++;

System.out.println(cnt);

}

}

<https://dheldh77.tistory.com/67>

[[알고리즘] 유니온파인드(Union-Find) 알고리즘](https://dheldh77.tistory.com/entry/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%9C%A0%EB%8B%88%EC%98%A8%ED%8C%8C%EC%9D%B8%EB%93%9CUnion-Find-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)

[Union-find](https://dheldh77.tistory.com/tag/Union-find), [알고리즘](https://dheldh77.tistory.com/tag/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98), [유니온파인드](https://dheldh77.tistory.com/tag/%EC%9C%A0%EB%8B%88%EC%98%A8%ED%8C%8C%EC%9D%B8%EB%93%9C)

**Union-Find 알고리즘이란?**

Union-Find 알고리즘은 그래프 알고리즘 중 하나로

 서로소 집합, 상호배타적 집합(Disjoint-Set)알고리즘이라고도 불린다.

 여러 노드가 존재할 때, 선택한 두 노드가 서로 같은 그래프에 속하는지 판별하는 알고리즘이다.

**Union-Find 과정**

**1) Init**

for(int i = 0; i < parent.length; i++) parent[i] = i;

초기화. 최초 노드는 자기자신을 루트노드로 가진다.

**1) Find(x)**

// x가 속한 그래프의 루트 노드를 반환

static int find(int x) {

// x의 루트노드가 x일 경우. 즉, x와 연결된 노드가 없을 경우.

if(x == parent[x]) return x;

// 재귀를 이용해 부모노드 -> 부모노드 ->...->루트노드의 방식으로 루트노드를 찾아 반환한다.

return find(x);

}

x가 속한 집합(그래프)의 root 노드를 반환한다.

**2) Union(x, y)**

// x와 y가 속한 두 그래프를 하나로 합친다.

static void union(int x, int y) {

// 두 노드의 루트노드를 찾는다.

x = find(x);

y = find(y);

// 두 루트노드 중 원소 값이 더 작은 루트노드를 그래프의 루트노드로 선택한다.

if(x > y) parent[x] = y;

else parent[y] = x;

}

x가 속한 집합(그래프)와 y가 속한 집합(그래프)를 합친다.

**너비 우선 탐색 구현**

package baekjoon;

import java.util.\*;

import java.io.\*;

public class BOJ\_2606 {

static int[] parent;

public static int find(int x) {

if(x == parent[x]) return x;

int y = find(parent[x]);

parent[x] = y;

return y;

}

public static void union(int x, int y) {

x = find(x);

y = find(y);

if(x != y) parent[y] = x;

}

public static void main(String[] args) throws IOException {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

StringTokenizer tk = new StringTokenizer(br.readLine());

int v = Integer.parseInt(tk.nextToken());

tk = new StringTokenizer(br.readLine());

int e = Integer.parseInt(tk.nextToken());

int cnt = 0;

parent = new int[v + 1];

for(int i = 1; i <= v; i++) {

parent[i] = i;

}

for(int i = 0; i < e; i++) {

tk = new StringTokenizer(br.readLine());

int x = Integer.parseInt(tk.nextToken());

int y = Integer.parseInt(tk.nextToken());

union(x, y);

}

for(int i = 2; i <= v; i++) if(find(1) == find(i)) cnt++;

System.out.println(cnt);

}

}

<https://dheldh77.tistory.com/67>

[[알고리즘] 에라토스테네스의 체](https://dheldh77.tistory.com/entry/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%97%90%EB%9D%BC%ED%86%A0%EC%8A%A4%ED%85%8C%EB%84%A4%EC%8A%A4%EC%9D%98-%EC%B2%B4)

2020. 2. 27. 11:10

**에라토스테네스의 체**

에라토스테네스의 체는 고대 그리스의 수학자 에라토스테네스가 소수를 찾기위해 만들어낸 방법으로,

 마치 체를 치듯이 수를 걸러내기 때문에 에라토스테네스의 체라고 부른다.

 '에라토스테네스의 체'로 소수를 구하는 방법은 2부터 시작해 하나의 수를 뽑고, 뽑은 수의 배수를 소수의 후보에서 제외시킨다.

**에라토스테네스의 체 과정**

만약, 1부터 10까지의 수 중 소수를 찾는다면

 2부터 시작해서 2의 배수를 소수의 후보에서 제외시킨다.

 2, 3, 4, 5, 6, 7, 8, 9, 10 (1은 소수가 아니므로 후보에서 제외)에서 2의 배수를 제거하면 2, 3, 5, 7, 9가 남는다.

 2, 3, 5, 7, 9 에서 3의 배수를 제거하면 2, 3, 5, 7이 남는다.

 이러한 방식으로 소수의 후보를 계속해서 제거하면, 남은 수들이 소수가 된다.

**에라토스테네스의 체 구현**

package samsung;

import java.util.\*;

public class s\_test {

public static void main(String[] args) {

int[] arr = new int[101];

for(int i = 2; i < arr.length; i++) {

int j = 2;

while(i \* j < arr.length) {

arr[i \* j] = 1;

j++;

}

}

for(int i = 2; i < 101; i++) {

if(arr[i] == 0)

System.out.print(i + " ");

}

}

}

## [[알고리즘] 배낭 문제(Knapsack Problem)](https://dheldh77.tistory.com/entry/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EB%B0%B0%EB%82%AD-%EB%AC%B8%EC%A0%9CKnapsack-Problem)

2020. 3. 2. 18:36

### 배낭 문제(Knapsack Problem)

배낭 문제란 배낭에 담을 수 있는 무게의 최대값이 정해져 있고,

일정한 가치와 무게가 정해져있는 짐들을 배낭에 닮을 때,

가치의 합이 최대가 되도록 짐을 고르는 방법을 찾는 문제이다.

배낭 문제는 크게

1) 물건을 쪼갤 수 있는 배낭문제(Fraction Knapsack Problem)와

2) 물건을 쪼갤 수 없는 배낭문제(0/1 Knapsack Problem)으로 나뉜다.

1) 물건을 쪼갤 수 있는 배낭문제의 경우는 가치가 큰 물건부터 담고, 남은 무게 만큼 물건을 쪼개는 방식으로

**그리디 알고리즘**으로 해결할 수 있다.

2) 물건을 쪼갤 수 없는 배낭문제의 경우는 **동적계획법(DP, Dynamic Programming)**을 활용해 해결할 수 있다.

### 0/1 Kanpsack Problem 과정

0/1 Knapsack Problem 문제를 해결하기 위해서 동적 계획법을 이용한다.

가방에 담을 수 있는 물건의 무게와 가치가 아래 표와 같을 때,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 물건 | 1 | 2 | 3 | 4 |
| 무게 | 6 | 4 | 3 | 5 |
| 가치 | 13 | 8 | 6 | 12 |

2차원 배열을 만들고 각 행과 열에 i번째 물건까지 넣을 수 있고, 배낭에 넣을 수 있는 최대무게가 j일 때, 최대 이윤을 저장해나간다.

이 때, 행 i 는 물건 i까지 넣을 수 있을 때를 의미하고, 열 j는 가방의 최대 무게를 의미한다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 13 |
| 2 |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |

 첫 번째 물건(무게 : 6, 가치 : 13)만 넣을 수 있을 때, 가방의 최대 무게에 따른 최대 이윤이다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 13 |
| 2 | 0 | 0 | 0 | 0 | 8 | 8 | 13 | 13 |
| 3 |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |

두 번째 물건(무게 : 4, 가치 : 8)만 넣을 수 있을 때, 가방의 최대 무게에 따른 최대 이윤이다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 13 |
| 2 | 0 | 0 | 0 | 0 | **8** | 8 | 13 | 13 |
| 3 | 0 | 0 | 0 | 6 | 8 | 8 | 13 | **14** |
| 4 |  |  |  |  |  |  |  |  |

세 번째 물건(무게 : 3, 가치 : 6)만 넣을 수 있을 때, 가방의 최대 무게에 따른 최대 이윤이다.

3행 7열을 보면 DP를 사용하는 이유를 알 수 있다. 가방의 최대 무게가 7일 때, 세번째 물건을 넣고 나서도 가방의 무게가 4가 남는다.

그렇다면 가방에는 최대무게가 4일 때와 같은 최대이윤을 더 남길 수 있다.

즉, DP[i][j]에는 v[i] + DP[i-1][j-w[i]]가 들어간다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 13 |
| 2 | 0 | 0 | 0 | 0 | 8 | 8 | 13 | 13 |
| 3 | 0 | 0 | 0 | 6 | 8 | 8 | 13 | 14 |
| 4 | 0 | 0 | 0 | 6 | 8 | 12 | 13 | 14 |

마지막 사이클까지 돌게되면 최대 이윤이 14인 것을 확인할 수 있다.

위의 과정을 점화식으로 나타내면 아래와 같이 나타낼 수 있다.

각 가방의 가치 : v[n]

각 가방의 무게 : w[n]

최대이윤 : dp[m][n]이라 할 때,

**(w[i] <= j ) dp[i][j] = max(v[i] + dp[i-1][j - w[i]], dp[i-1][j]**

**(w[i] > j) dp[i][j] = dp[i-1][j]**

### 0/1 Knapsack Problem 구현

package baekjoon;

import java.util.\*;

public class BOJ\_12865 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int k = sc.nextInt();

int[] w = new int[n];

int[] v = new int[n];

int[][] dp = new int[n+1][k+1];

int max = 0;

for(int i = 0; i < n; i++) {

w[i] = sc.nextInt();

v[i] = sc.nextInt();

}

for(int i = 1; i <= n; i++) {

for(int j = 1; j <= k; j++) {

if(w[i-1] <= j)

dp[i][j] = Math.max(v[i-1] + dp[i-1][j - w[i-1]], dp[i-1][j]);

else

dp[i][j] = dp[i-1][j];

max = Math.max(dp[i][j], max);

}

}

System.out.println(max);

}

}