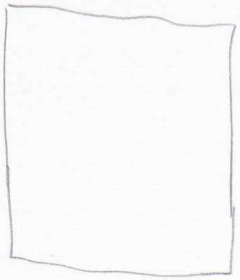
5.2.

8x8



(1) (1,1) o → (x,1) → (x,x), (x,1), (1,x) 배열

(2) (2,1) x, (2,2) x, (2,3) o → (x, x+1) 배열

(3) (3,1) x, (3,2) x, ... (3,5) o

(4) (4,2) o

(5) (5,4) o

(6) (6,6)

(7) (7,8)

(8) (6,7)

(9) (6,8)

(10) (7,6)

(11) (5,6)

(12) (6,4)

(13) (7,7)

(14) (4,7)

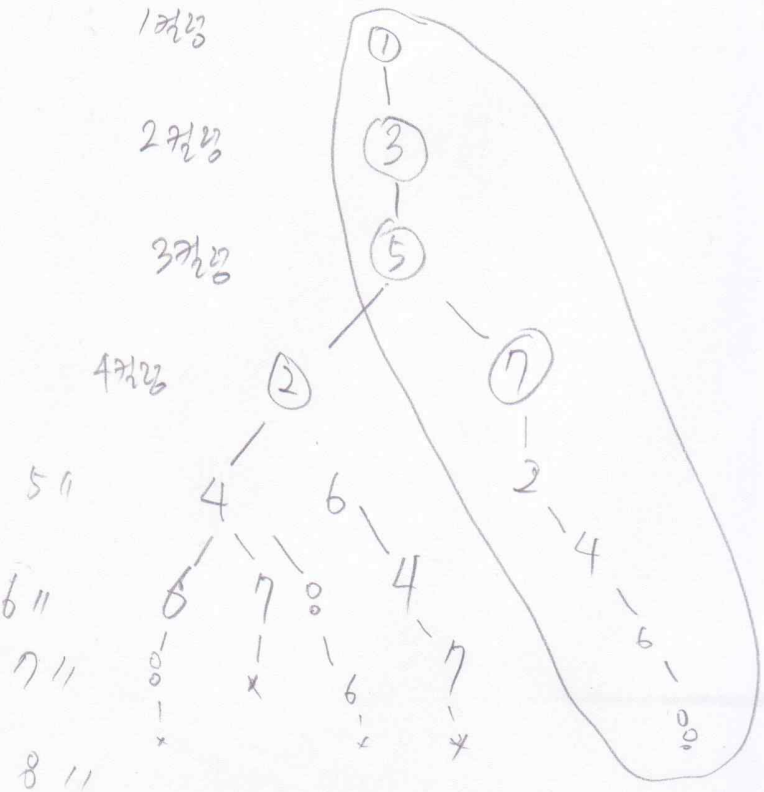
(15) (5,2)

(16) (6,4)

(17) (7,6)

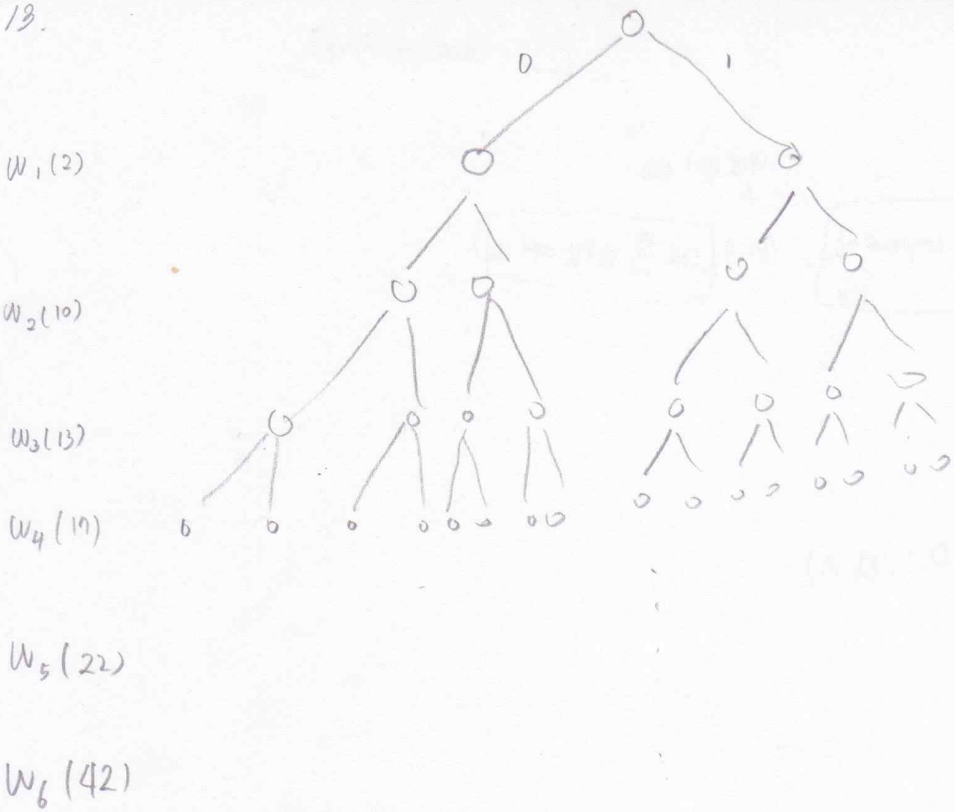
(18) (8,8)

방문한 코만 표시.



0: 선택 (왼쪽) 1: 안택 (오른쪽)

19.



결과: tuple(0,0,1,1,1,0) - [13,17,22]

tuple(0,1,0,0,0,1) - [10,42]

18. 알고리즘은 이용하여 풀이. (Python 3.) 실행한 화면을 대체.

```
# 간선에 대한 연결 확인
W = [ [0,1,0,1,0,0]  ,
      [1,0,1,0,1,0]  ,
      [0,1,0,0,0,1]  ,
      [1,0,0,0,1,0]  ,
      [0,1,0,1,0,1]  ,
      [0,0,1,0,1,0] ]
# 컬러 개수
color = ['Red' , 'White', 'Green']
# 색칠한 Vertex 배열
vcolor = [0 for _ in range(len(W))]
count = 1
def coloring(i) :
        global count
        if (promising(i)) :
                # 0은 이미 색칠해서 들어오므로 -1과 비교
                if (i == len(W)-1) :
                        print(count, end = " ")
                        count += 1
                        for n in vcolor :
                                print(n, end= " ")
                        print()
                else :
                        for j in range(0,len(color)) :
                                vcolor[i+1] = color[j]
                                coloring(i+1)

# 연결된 Vertex와 같은 색이 존재하는지 체크
# 같은 색이 아니라면 True를 리턴 같은 색이 존재한다면 False를 리턴
def promising(i) :
        j = 0
        switching = True
        while (j<i and switching) :
                if (W[i][j] == 1 and vcolor[i] == vcolor[j]) :
                        switching = False
                j += 1
        return switching
# 3가지 색에 대한 결과값
for j in range(0,len(color)) :
        vcolor[0] = color[j]
        coloring(0)
```

22. $\frac{P_i}{W_i}$ 순으로 정렬되어 있다.

= recursion function

backtracking - recursive의 변형.

① max (20 + [W=7에 대한 knapsack 값] ← 최대, W=9 [다음 $\frac{P_i}{W_i}$ 값을 뺀 값]).

아무것도 넣지 않음.

결과값 : tuple (1, 0, 1, 0, 0).

알고리즘 : Python 3.

```
def knapSack(W, wt, val, n):
    # 모든 List를 방문했을 때 or 무게가 다 찼을 때 0을 return
    if n < 0 or W == 0 :
        return 0
    # 물건의 무게가 가방에 남은 무게보다 클 경우
    # 다른 물건으로 테스트. (이 부분이 교재에선 Promising)
    if (wt[n] > W):
        return knapSack(W, wt, val, n-1)

    # 물건의 무게가 가방에 남은 무게보다 작을경우
    # 지금 물건의 가치 + 지금 물건을 넣은 후의 값 과
    # 지금 물건을 넣지 않고, 다른 물건을 넣었을 때의 값 중 큰 값을 리턴
    else:

        return max(val[n] + knapSack(W-wt[n], wt, val, n-1),
                   knapSack(W, wt, val, n-1))

val = [20, 30, 35, 12 ,3]
wt = [2, 5, 7, 3, 1]
W = 9
n = len(val)
print (knapSack(W, wt, val, n-1) )
```

43. Algorithm with python 을 이용한 구현 설명은 주석으로 대체

```
# 간선에 대한 Weight
W = [ [0,1,0,2,0,0] ,
      [3,0,4,0,5,0] ,
      [0,6,0,0,0,7] ,
      [8,0,0,0,9,0] ,
      [0,10,0,11,0,12] ,
      [0,0,13,0,14,0] ]

# 연결된 Vertex 배열
vindex = [0 for _ in range(len(W))]
# 최대값을 저장할 글로벌 변수
maximum = 0

def hamiltonian(i) :
        global maximum
        if (promising(i)) :
                if (i == len(W)-1) :
                        visitmax = 0
                        # 간선의 총합을 구함
                        for i in range(1, len(W)) :
                                start = vindex.index(i-1)
                                end = vindex.index(i)
                                visitmax += W[start][end]
                        print(vindex)
                        # max값을 maximum에 대입
                        maximum = max(maximum, visitmax)
                else :
                        # 모든 변수 Visit
                        for j in range(0,len(W)) :
                                vindex[i+1] = j
                                hamiltonian(i+1)

# Visit 했던 Vertex가 있는지 체크
# 간선이 연결되어있는지 체크
def promising(i) :
        j = 0
        switching = True
        if i != 0 and W[vindex[i-1]][vindex[i]] == 0 :
                switching = False
        if vindex[i] in vindex[:i] :
                switching = False
        return switching

result = list()
# 시작점을 다르게하여 출발
for j in range(len(W)) :
        vindex = [0 for _ in range(len(W))]
        vindex[0] = j
        hamiltonian(0)

print(maximum)
```