stack_02

≔ 태그

계산기 1

중위 표기법과 후위 표기법

- 문자열로 된 계산식이 주어졌을 때, 스택을 사용하여 이 계산식의 값을 계산 가능
- 일반적 방법
    - 중위 표기법 (A+B)
    - 후위 표기법 (AB+)
    - 연산자를 만나면 앞선 숫자를 pop 하는 것

- 중위 표기식의 후위 표기식 변환 방법

- 수식의 각 연산자에 대해서 우선순위에 따라 괄호를 사용하여 다시 표현한다.
- 각 연산자를 그에 대응하는 오른쪽괄호의 뒤로 이동시킨다.
- 괄호를 제거한다.

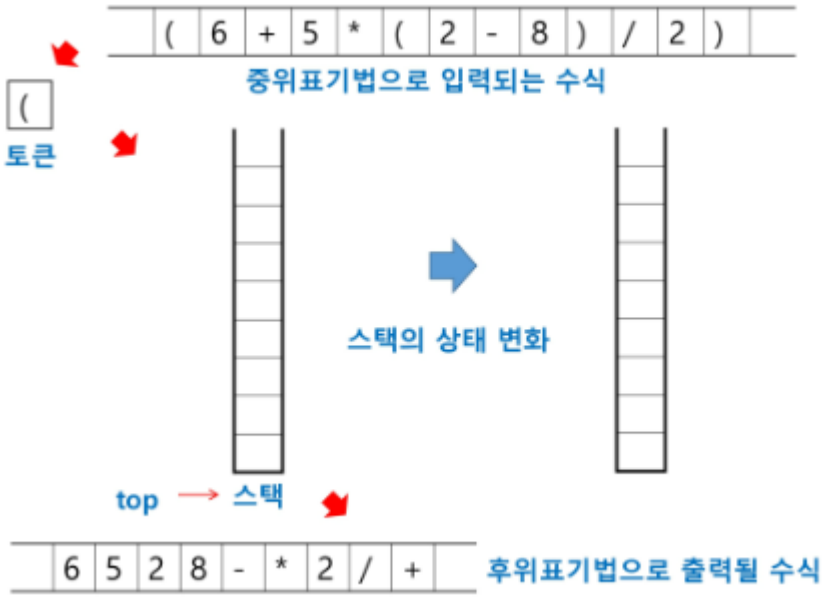
예) A*B-C/D

1단계 : ((A*B) - (C/D))

2단계 : ((A B)* (C D)/)-

3단계 : AB*CD/-

- 변환 알고리즘 이용(스택 이용)2
  1. 입력 받은 중위 표기식에서 토큰을 읽는다.
  2. 토큰이 피연산자라면 토큰을 출력한다.
  3. 스택의 top에 저장되어 있는 연산자보다, 우선순위가 높으면 push, 아니면 pop
  4.) 이 나오면 (나올때 까지 pop
  5. 더 읽을 것이 있으면 1부터 다시 반복
  6. 스택에 남아있는 연산자를 모두 pop

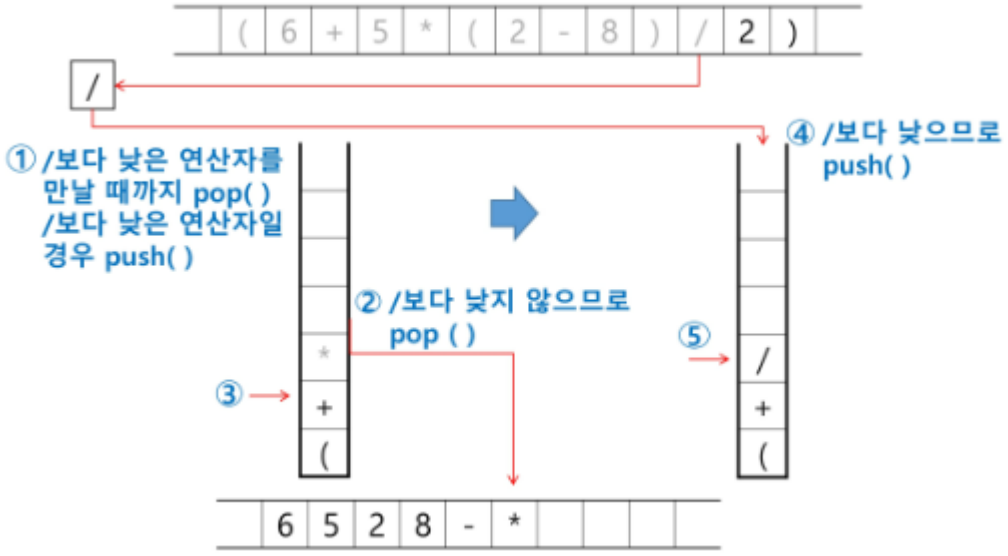


```
icp(in-coming priority)
isp(in-stack priority)

if (icp > isp)   push()
else pop()
```

토큰	isp	icp
)	-	-
*, /	2	2
+,-	1	1
(	0	3

- 우선순위(icp,isp)를 기억하는 것이 중요하다

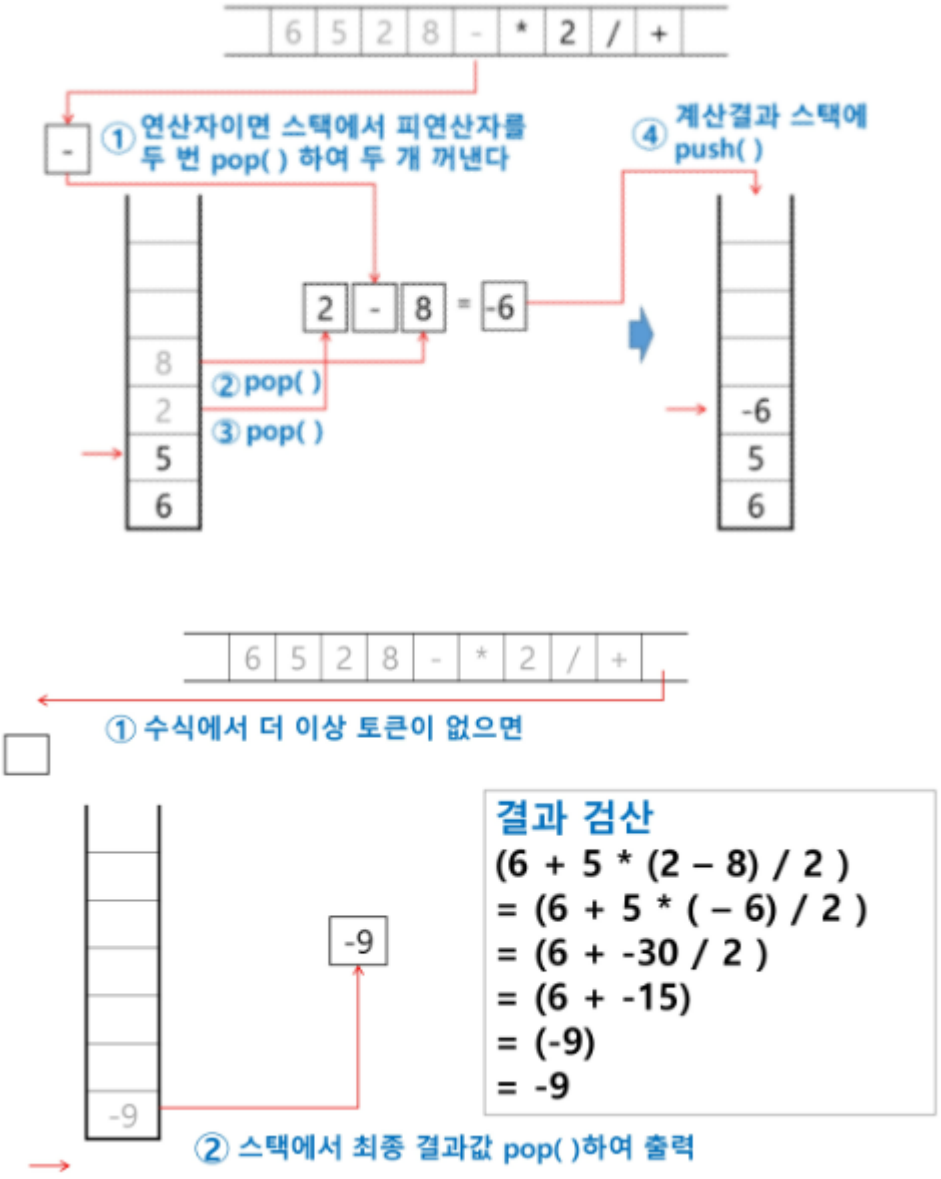
- 우선순위가 같다면



계산기 2

- 후위표기법으로 바뀐 수식을 스택을 통해 계산한다.
  1. 피연산자를 만나면 push
  2. 연산자를 만나면 필요한 만큼 피연산자를 pop

3. 수식이 끝나면, 마지막으로 스택을 pop하여 출력



- 후위표기법 코드

```
'''
fx = (6+5*(2-8)/2)
'''
```

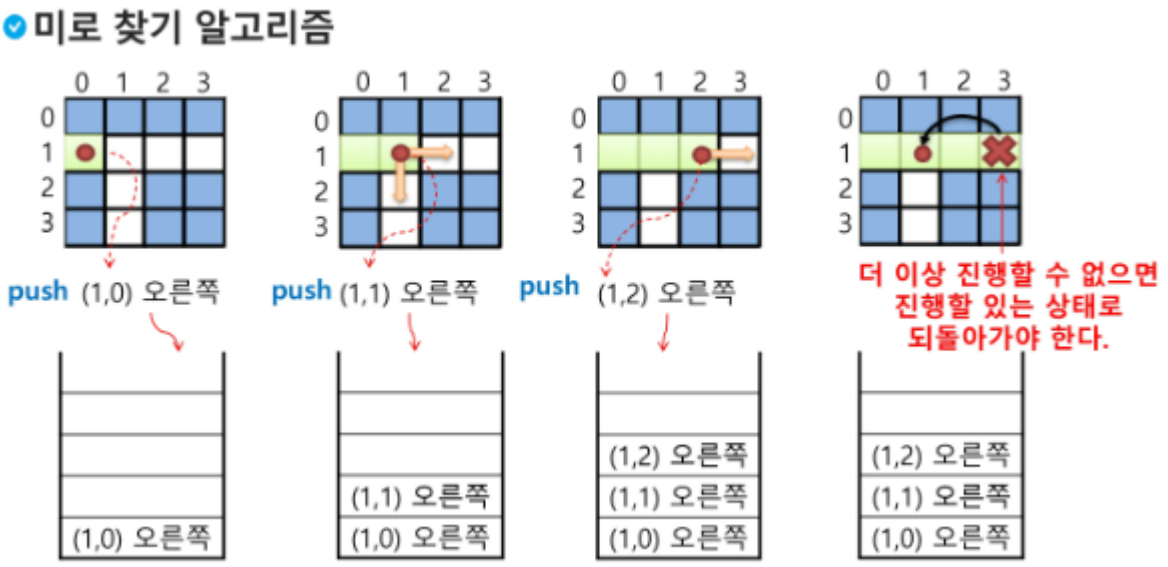
```
top = -1
stack = [0] * 100

icp = {'(' : 3, '*' : 2, '/' : 2, '+' : 1, '-' : 1} # 스택 밖에
isp = {'(' : 0, '*' : 2, '/' : 2, '+' : 1, '-' : 1} # 스택 안에

fx = '(6+5*(2-8)/2)'
postfix = ''
for tk in fx:
     # 여는 괄호 push, top 원소보다 우선순위가 높으면 push
    if tk == '(' or (tk in '*/+-' and isp[stack[top]] < icp[t
        top += 1 # push
        stack[top] = tk
    elif tk in '*/+-' and isp[stack[top]] >= icp[tk]: # 연산자
        #top원소의 우선순위가 낮을 때 까지 pop
        while isp[stack[top]] >= icp[tk]:
            top -= 1 # pop
            postfix += stack[top+1]
        top += 1
        stack[top] = tk
    elif tk == ')': # 여는 괄호를 만날 때 까지
        while stack[top] != '(':
            top -= 1
            postfix += stack[top + 1]
        top -= 1 # 여는 괄호 pop 해서 버림
    else: # 피연산자인 경우
        postfix += tk
print(postfix)
```

백트래킹

- 해를 찾는 도중에 '막히면' 되돌아가서 다시 해를 찾는 방법
- 이는 최적화 문제와 결정 문제를 해결할 수 있다.
- 결정 문제 ⇒ 문제의 조건을 만족하는 해가 존재하는지의 여부를 yes 또는 no로 답하는 문제

미로찾기

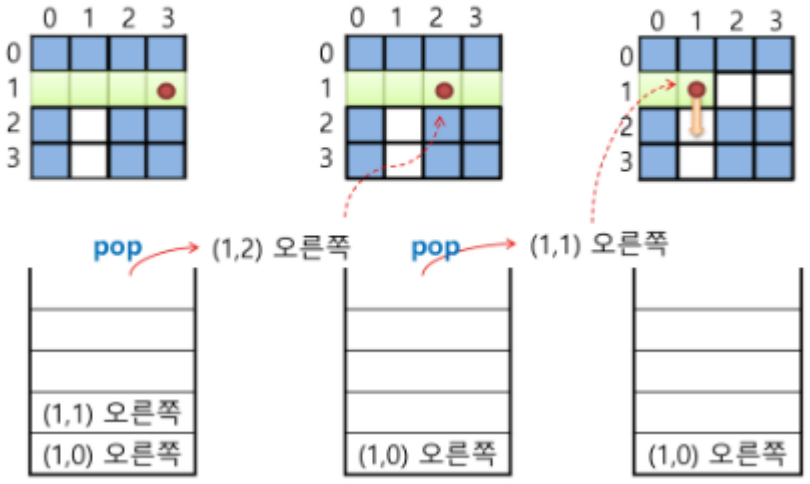
- 이동할 수 있는 4가지 방향
- 일단 가고 본다



- pop 해서 지나온 경로를 역으로 되돌아간다.

미로 찾기 알고리즘

- 스택을 이용하여 지나온 경로를 역으로 되돌아 간다.



- 백트래킹은 불필요한 경로를 조기에 차단한다.
- 깊이 우선 탐색이 너무 많은 경우의 수로 인해, 처리 불가능 할 수 있는 것을 처리 가능
- 허나 백트래킹도 무조건 처리 가능한 것은 아니다.(지수함수시간이 될 수 있어서)

모든 후보를 검사?

- No!

백트래킹 기법

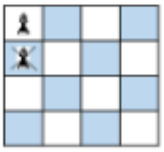
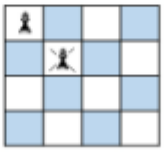
- 어떤 노드의 유망성을 점검한 후에 유망(promising)하지 않다고 결정되면 그 노드의 부모로 되돌아가(backtracking) 다음 자식 노드로 감
- 어떤 노드를 방문하였을 때 그 노드를 포함한 경로가 해답이 될 수 없으면 그 노드는 유망하지 않다고 하며, 반대로 해답의 가능성이 있으면 유망하다고 한다.
- 가지치기(pruning) : 유망하지 않는 노드가 포함되는 경로는 더 이상 고려하지 않는다.

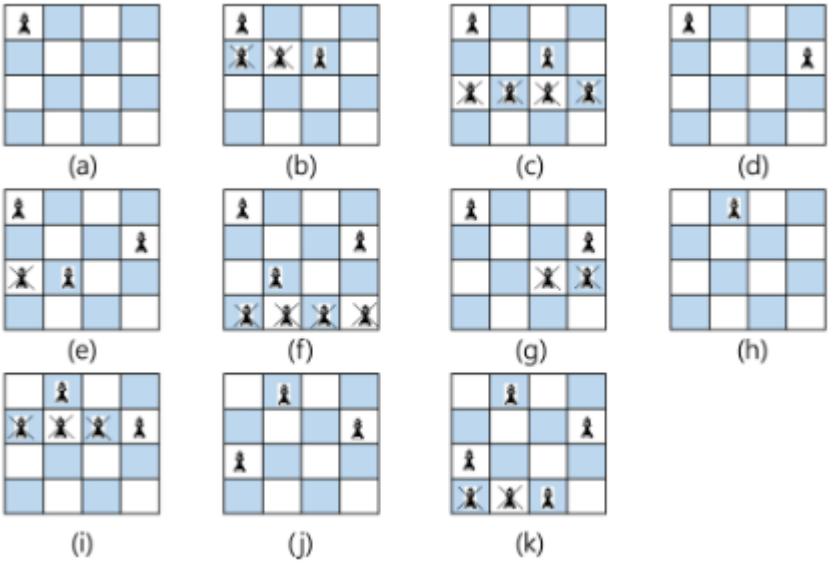
- 각 노드가 유망한지를 점검한다. → 그 노드가 유망하지 않으면, 그 노드의 부모로 돌아간다. 는 근본적인 차이가 존재한다.

- 일반 백트래킹 알고리즘

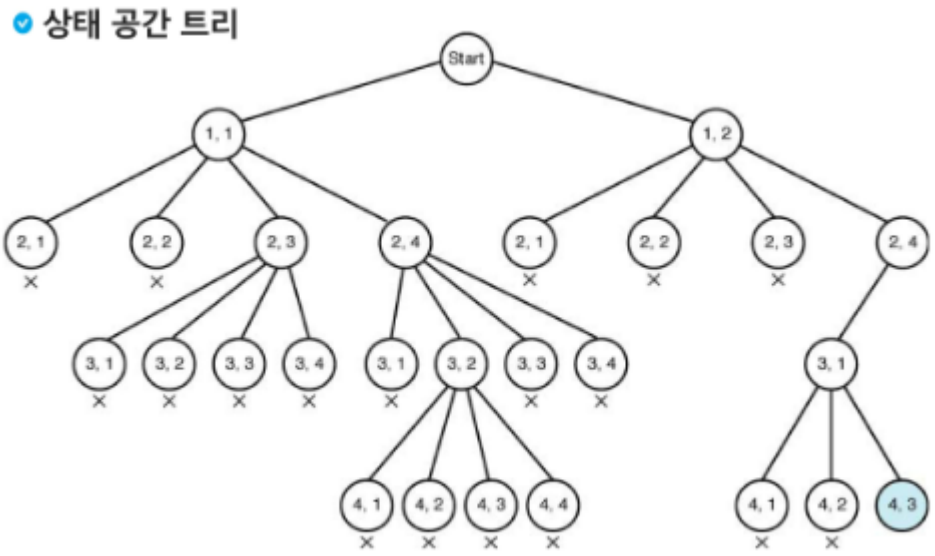
일반 백트래킹 알고리즘

```
def checknode (v) : # node
    if promising(v) :
        if there is a solution at v :
            write the solution
        else :
            for u in each child of v :
                checknode(u)
```



- 상태 공간 트리



부분집합

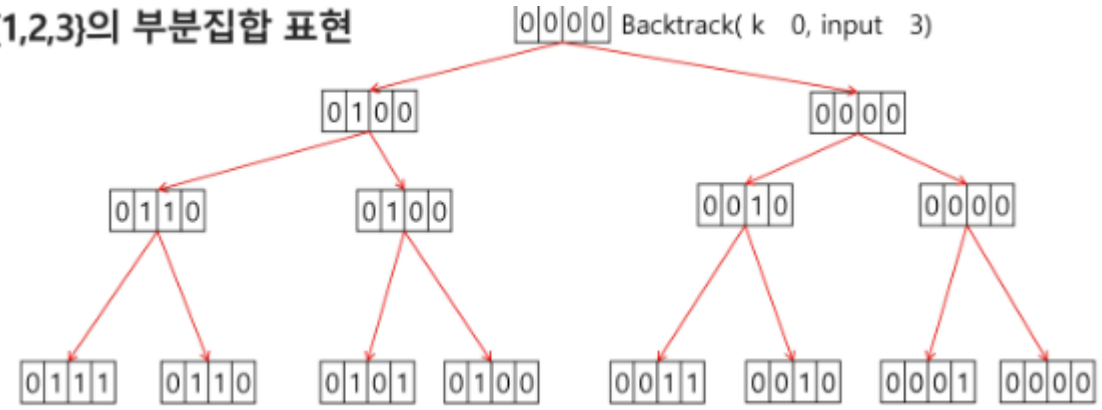
원소의 개수가 n개이면 부분집합의 개수는 2^n개 이다.

- 백트래킹 기법으로 powerset을 만들면
  1. n개의 원소가 들어있는 집합의 2^n개의 부분집합을 만들 때는 , true, false값을 가지는 항목으로 구성된 n개의 배열을 만드는 방법 이용
  2. i번째 항목은 i번째의 원소가 부분집합의 값인지 아닌지를 나타내는 값

각 원소가 부분집합에 포함되었는지를 loop 이용하여 확인하고 부분집합을 생성하는 방법

```
bit = [0, 0, 0, 0]
for i in range(2) :
    bit[0] = i                  # 0번째 원소
    for j in range(2) :
        bit[1] = j              # 1번째 원소
        for k in range(2) :
            bit[2] = k          # 2번째 원소
            for l in range(2) :
                bit[3] = l      # 3번째 원소
                print(bit)      # 생성된 부분집합 출력
```

{1,2,3}의 부분집합 표현



- powerset을 구하려면

```
def backtrack(a, k, input) :
    global MAXCANDIDATES
    c = [0] * MAXCANDIDATES

    if k == input :
        process_solution(a, k) # 답이면 원하는 작업을 한다
    else :
        k+=1
        ncandidates = construct_candidates(a, k, input, c)
        for i in range(ncandidates) :
            a[k] = c[i]
            backtrack(a, k, input)
```

- 예시 코드

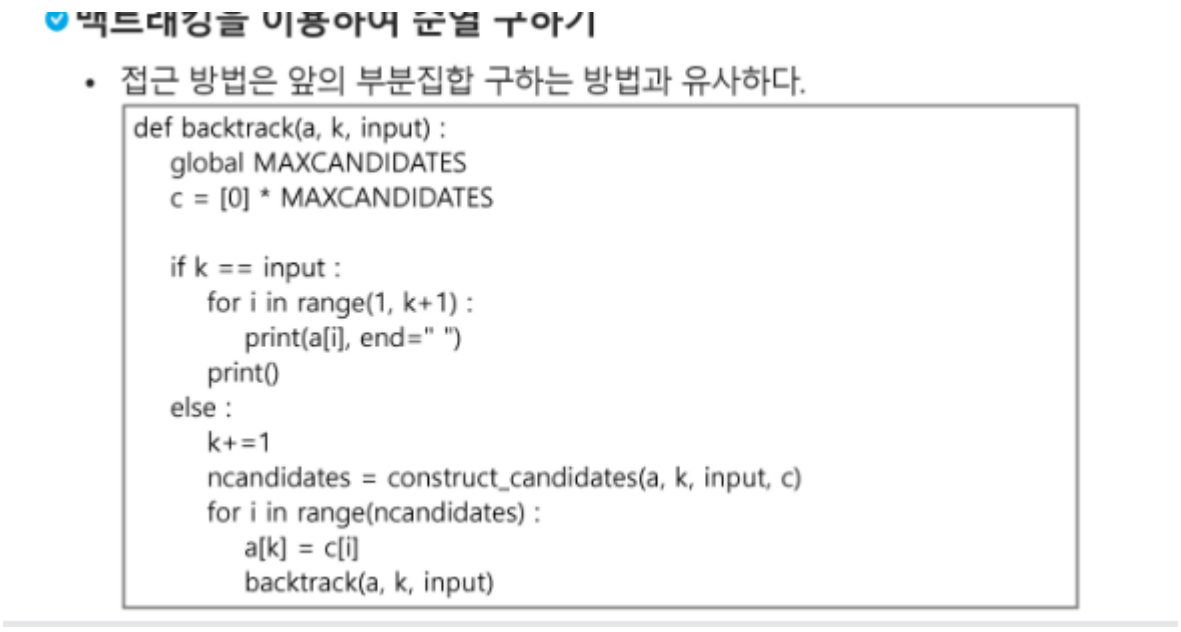
```
def f(i,k):
    if i == k:
        for j in range(k):
            if bit[j]:
                print(arr[j], end = ' ')
        print()
    else:
        for j in range(2):
            bit[i] = j
            f(i+1, k)


N = 4
arr = [1,2,3,4]
bit = [0] * N # bit[i] : arr[i] 가 부분집합에 포함되었는지를 나타내는
f(0, N)       # bit[i]에 1또는 0을 채우고, N개의 원소가 결정되면 부분
```

순열

- 그냥 순열을 생성하려면

```
for i1 in range(1,4):
    for i2 in range(1,4):
        if i2 != i1 :
            for i3 in range(1,4):
                if i3 != i1 and i3 != i2:
                    print(i1,i2,i3)
```

- 이를 백트래킹으로 구현하려면

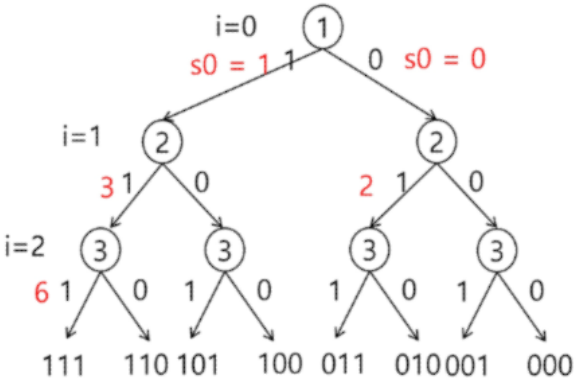


백트래킹을 이용하여 순열 구하기(계속)

```
def construct_candidates(a, k, input, c) :
    in_perm = [False] * NMAX

    for i in range(1, k) :
        in_perm[a[i]] = True

    ncandidates = 0
    for i in range(1, input+1) :
        if in_perm[i] == False :
            c[ncandidates] = i
            ncandidates += 1
    return ncandidates
```

부분집합 구현

- i원소의 포함 여부를 결정 하면 i까지의 부분 집합의 합 s_i를 결정할 수 있음
- s_{i-1}이 찾고자 하는 부분집합의 합보다 크면 남은 원소를 고려할 필요가 없음



i	0	1	2
A	1	2	3

f(i, N, s, t) # i-1원소까지의 합 s

- 코드

```
## {1,2,3,4,5,6,7,8,9,10} => 합이 10인 부분집합을 구하시오
```

```
def f(i,k,t): # k개의 원소를 가진 배열 A, 부분집합의 합이 t인 경우를 찾
    if i == k: # 모든 원소에 대해 결정하면
        ss = 0 # 부분집합 원소의 합
        for j in range(k):
            if bit[j]: # A[j]가 포함된 경우
                ss += A[j]
        if ss == t:
            for j in range(k):
                if bit[j]: # A[j]가 포함된 경우
                    print(A[j], end = ' ')
            print() # 부분집합 출력
    else:
        for j in range(1, -1, -1):
            bit[i] = j
            f(i+1, k, t)

        # bit[i] = 1
        # f(i+1, k)
        # bit[i] = 0
        # f(i+1, k)

N = 10
A = [1,2,3,4,5,6,7,8,9,10]
bit = [0] * N # bit[i] 는 A[i]가 부분집합에 포함되는지 표시
f(0,10,10)
```

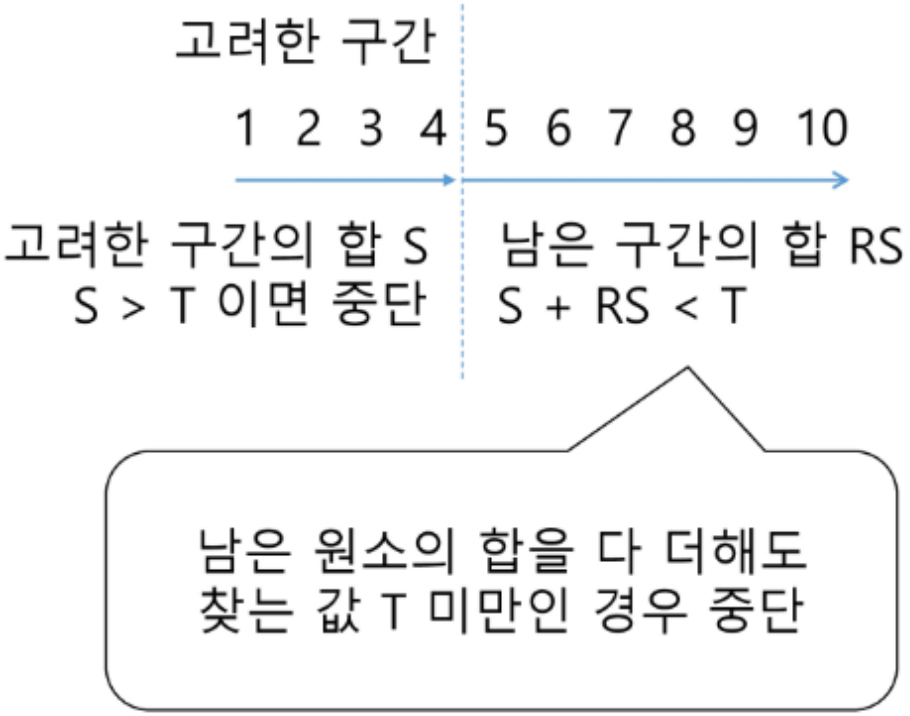
- 부분집합의 부분집합을 고려하는 방식

```
## {1,2,3,4,5,6,7,8,9,10} => 합이 10인 부분집합을 구하시오

def f(i, k, s, t): # k개의 원소를 가진 배열 A, 부분집합의 합이 t인 경
    global cnt
    cnt += 1
    if s == t: # 모든 원소에 대해 결정하면
        for j in range(k):
            if bit[j]: # A[j]가 포함된 경우
```

```
                print(A[j], end = ' ')
        print() # 부분집합 출력
    elif i == k: # 모든 원소를 고려했으나 s!=t
        return
    elif s > t: # 고려한 원소의 합이 t보다 큰 경우
        return
    else:
        bit[i] = 1
        f(i+1, k, s+A[i], t)
        bit[i] = 0
        f(i+1, k, s, t)
        # bit[i] = 1
        # f(i+1, k)
        # bit[i] = 0
        # f(i+1, k)

N = 10
A = [1,2,3,4,5,6,7,8,9,10]
bit = [0] * N # bit[i] 는 A[i]가 부분집합에 포함되는지 표시
cnt = 0
f(0, N, 0, 5)
print(cnt)
```

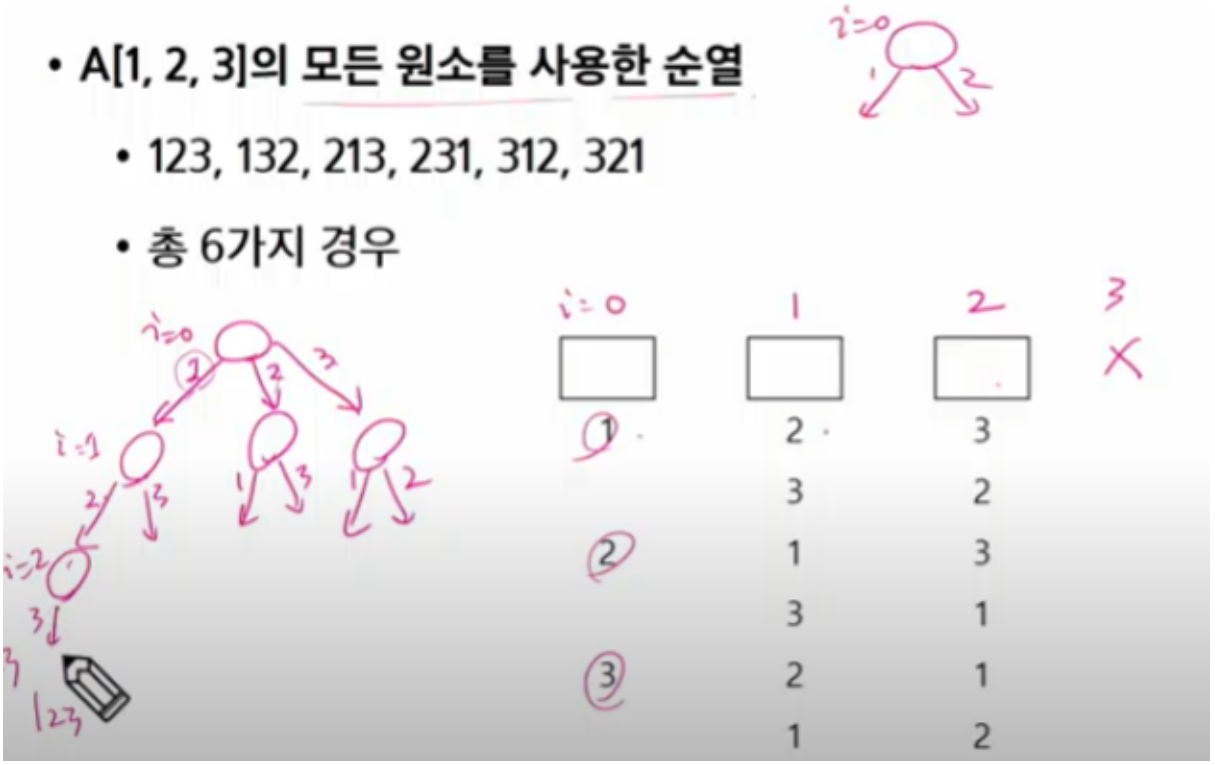
- 추가 고려 사항

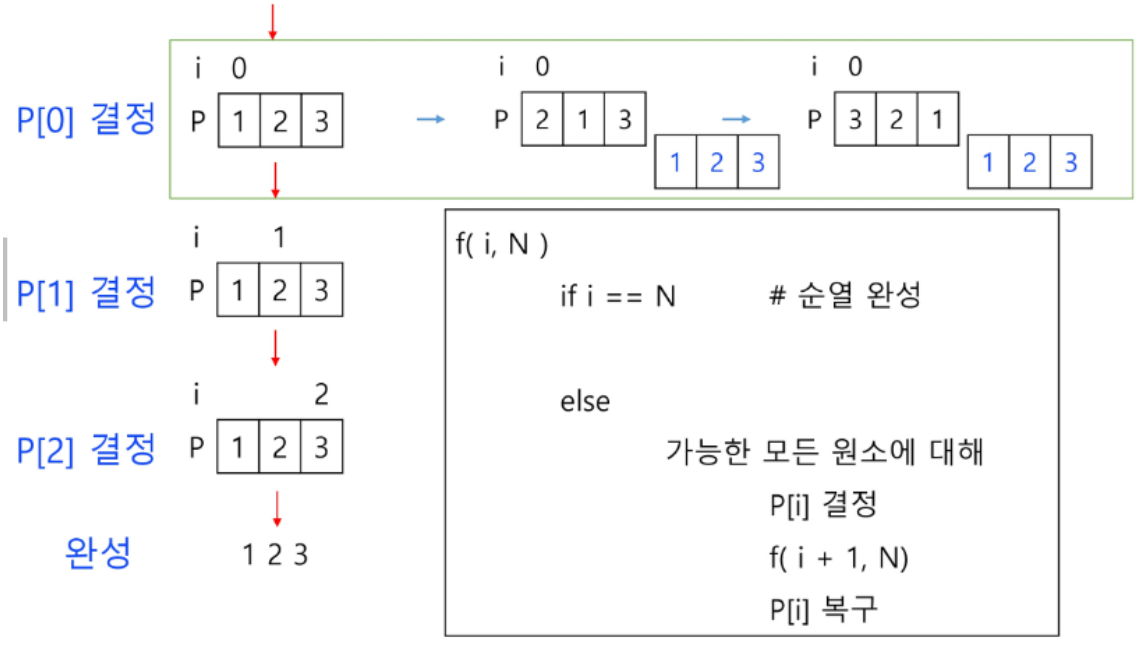
• **추가 고려 사항**


고려한 구간
1 2 3 4 5 6 7 8 9 10
고려한 구간의 합 S
S > T 이면 중단
남은 구간의 합 RS
S + RS < T
남은 원소의 합을 다 더해도
찾는 값 T 미만인 경우 중단

순열 구현

- A[1, 2, 3]의 모든 원소를 사용한 순열
  - 123, 132, 213, 231, 312, 321
  - 총 6가지 경우





- 코드

```
def f(i,k,s):
    global cnt
    global min_v
    cnt += 1
    if i == k:
        # print(*P)
        if min_v > s:
            min_v = s
    elif s >= min_v:
        return
    else:
        for j in range(i, k):  # P[i]자리에 올 원소 P[j]
            P[i], P[j] = P[j], P[i] # P[i] <-> P[j]
            f(i+1, k, s + arr[i][P[i]])
            P[i], P[j] = P[j], P[i] # 원상
            # 복구

N = int(input())
arr = [list(map(int,input().split())) for _ in range(N)]
P = [i for i in range(N)]
min_v = 100
cnt = 0
f(0, N, 0)
print(min_v, cnt)
```

분할 정복

- 분할 : 해결할 문제를 여러 개의 작은 부분으로 나눈다.
- 정복 : 나눈 작은 문제를 각각 해결한다.
- 통합 : 해결된 해답을 모은다.

- 퀵정렬

- **pivot을 잡는다**

- 코드

```
## quick sort
def quick_sort(start, end):
    # 언제까지 조사할거냐
    # stack에 값이 있는 동안
    stack = [(start, end)]
    while stack:
        start, end = stack.pop()
        if start < end: # 조사 범위가 꼬이지 않았다면
            pivot_index = partition(start,end)
            stack.append((start, pivot_index -1)) # pivot 왼쪽
            stack.append((pivot_index + 1, end))


def partition(start, end):
    pivot = arr[end]
    i = start - 1
    for j in range(start, end):
        if arr[j] <= pivot:
            i += 1      # 마지막에 pivot위치의 값이 들어가야 할 위치
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[end] = arr[end], arr[i+1]
    return i + 1


arr = [3,6,8,10,1,2,1]
N = len(arr)
quick_sort(0,N-1)
print(arr)
```

- 재귀로 푸는 방법

```
## quick sort
def quick_sort(lst): # index 조절이 아닌 list 자체를 조절한다.
    if len(lst) <= 1: # 정렬대상을 분할해 나가다가
        return lst # 더 이상 분할 할 수 없는 상태가 되면, 해당 리스트
    else:
        pivot = lst[0] # 퀵 소트의 pivot 위치는 아무 대상이어도 상관
        # pivot보다 작은 대상만 모음
        less_than_pivot = [x for x in lst[1:] if x <= pivot]
        # pivot보다 큰 대상만 모음
        greater_than_pivot = [x for x in lst[1:] if x > pivot
        return quick_sort(less_than_pivot) + [pivot] + quick_
arr = [3,6,8,10,1,2,1]
result = quick_sort(arr)
print(result)
```