

Stack 2

Stack2 - 1. 계산기

문자열 수식의 일반적인 계산 방법

Step 1 중위 표기법의 수식을 후위 표기법으로 변경

- 스택 이용
 - 중위 표기법(infix notation): 연산자를 피연산자의 가운데 표기하는 방법
 - ex) $A+B$
 - 일반적으로 사용하는 수식 표기법

Step 2 후위 표기법의 수식을 스택을 이용하여 계산

- σ 후위 표기법(postfix notation): 연산자를 피연산자의 뒤에 표기하는 방법
ex) $A + B \rightarrow AB +$
(중위) (후위)

중위 표기식을 후위 표기식으로 변환하는 방법 1

- 1 $((A * B) - (C / D))$: 우선순위에 따라 연산마다 괄호로 다 묶기
 - 2 $((AB)^*(CD)/) -$: 연산자를 괄호 뒤로 다 빼기
 - 3 $A B * C D / -$: 괄호 다 없애기

$$(6 + 5 * (2 - 8) / 2)$$

1. $(6 + ((5 * (2 - 8)) / 2))$
 2. $(6((5(28) -) * 2)) +$
 3. $6\ 5\ 2\ 8\ -\ *\ 2\ /\ +$

🔍 중위 표기식을 후위 표기식으로 변환하는 방법 2 - 스택 이용

1 입력 받은 중위표기식에서 토큰을 읽음

2 토큰이 피연산자이면 토큰을 출력

3 토큰이 연산자(괄호포함)일 경우

- ▶ 우선순위가 높으면 → 스택에 push
- ▶ 우선순위가 안 높으면 → 연산자의 우선순위가 토큰의 우선순위보다 작을 때까지 스택에서 pop한 후 토큰의 연산자를 push
- ▶ 만약 top에 연산자가 없으면 → push

4 토큰이 오른쪽 괄호 ')' 일 경우

- ▶ 스택 top에 왼쪽 괄호 '('가 올 때까지 스택에 pop 연산을 수행
- ▶ pop한 연산자를 출력
- ▶ 왼쪽 괄호를 만나면 pop만 하고 출력하지는 않음

5 중위표기식에 더 읽을 것이 없다면 종지, 더 읽을 것이 있다면 1부터 반복

6 스택에 남아 있는 연산자를 모두 pop하여 출력

- ▶ 스택 밖의 왼쪽 괄호는 우선 순위가 가장 높으며,
스택 안의 왼쪽 괄호는 우선 순위가 가장 낮음

< 영산자 우선순위 >

토큰	ISP	ICP
)	-	-
*, /	2	2
+, -	1	1
(0	3

icp(in-coming priority)

isp(in-stack priority)

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

중위표기법

(6 + 5 * (2 - 8) / 2)

변환

후위표기법

6 5 2 8 - * 2 / +

1. 연산자는 스택을 거쳐 출력으로!
2. 우선순위에 따라

↑ 높
↓ 낮

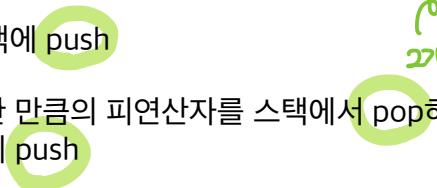


피연산자는 바로 출력으로 이동



★ 연산자가 거쳐갈 stack과 출력을 담을 공간이 필요함 ★

🔍 후위 표기법의 수식을 스택을 이용하여 계산

- 1 피연산자를 만나면 스택에 push
- 2 연산자를 만나면 필요한 만큼의 피연산자를 스택에서 pop하여 연산하고, 연산결과를 다시 스택에 push

- 3 수식이 끝나면, 마지막으로 스택을 pop하여 출력

먼저 pop한 값이
뒤의 값이 된다!

2개!

- , / 연산시주의

! 후위 표기식을 계산 시, 피연산자를 스택에 쌓아 계산 !

- 문자열로 된 수식을 계산 시 :: 내장함수 `eval()`

>> 올바른 수식이 아닌 경우, SyntaxError 예외가 발생

Stack2 - 2. 백트래킹

: 해를 찾는 도중에 막히면, (해가 아니면) 되돌아가서 다시 해를 찾아가는 기법

최적화 (Optimization) 문제

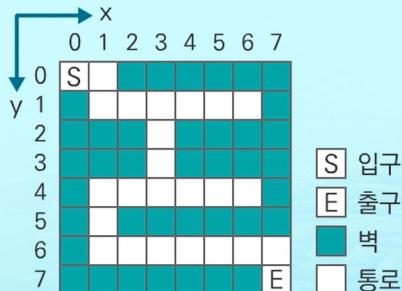
결정 (Decision) 문제

문제의 조건을 만족하는 해가 존재하는지
여부를 'yes' 또는 'no'로 답하는 문제
→ ex) 미로찾기, n-Queen, Map coloring,
부분집합의 합 (Subset Sum) 문제 등

백트래킹 :: 미로찾기

1 입구와 출구가 주어진 미로에서 입구부터 출구까지의 경로를 찾는 문제

2 이동할 수 있는 방향은 4방향으로 제한

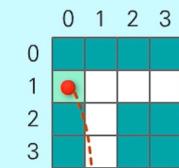


```
mazeArray = [  
    [0, 0, 1, 1, 1, 1, 1, 1],  
    [1, 0, 0, 0, 0, 0, 0, 1],  
    [1, 1, 1, 0, 1, 1, 1, 1],  
    [1, 1, 1, 0, 1, 1, 1, 1],  
    [1, 0, 0, 0, 0, 0, 0, 1],  
    [1, 0, 1, 1, 1, 1, 1, 1],  
    [1, 0, 0, 0, 0, 0, 0, 0],  
    [1, 1, 1, 1, 1, 1, 1, 0]  
]
```

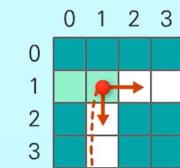
2차 리스트에 표현된 미로



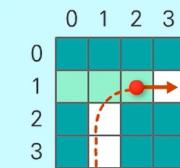
예 백트래킹 기법 활용 - 미로 찾기 알고리즘 1



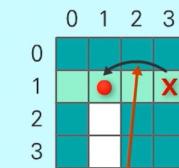
push(1, 0) 오른쪽



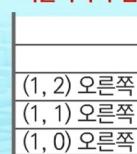
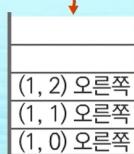
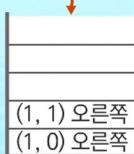
push(1, 1) 오른쪽



push(1, 2) 오른쪽



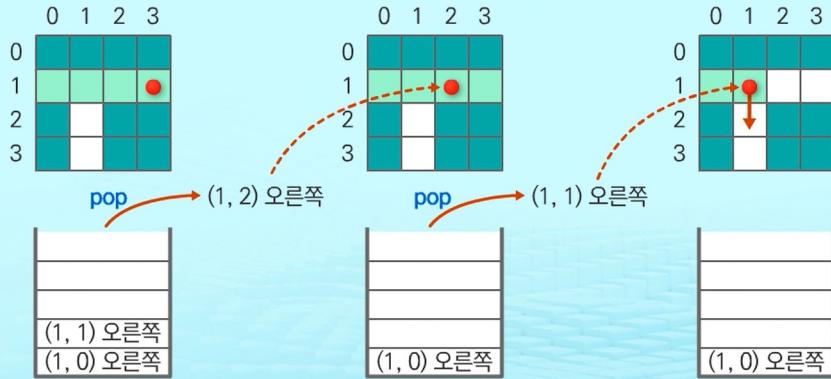
더 이상 진행할 수 없으면
진행할 있는 상태로
되돌아가야 함





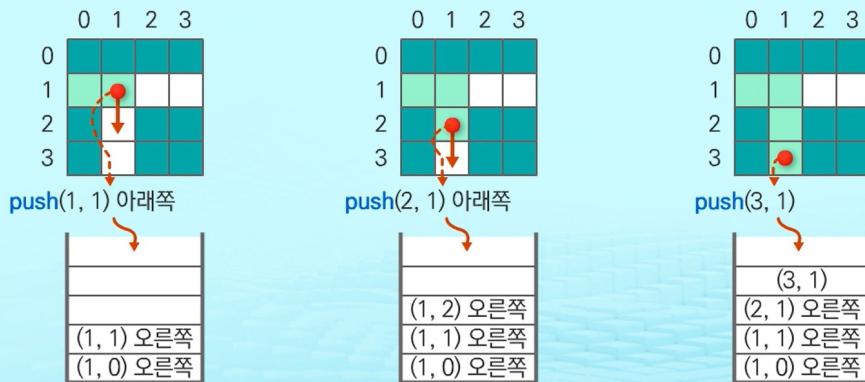
백트래킹 기법 활용 - 미로 찾기 알고리즘 2

스택을 이용하여 지나온 경로를 역으로 되돌아감



백트래킹 기법 활용 - 미로 찾기 알고리즘 3

스택을 이용하여 다시 경로를 찾기



백트래킹과 깊이 우선 탐색의 차이

백트래킹

- ▶ 어떤 노드에서 출발하는 경로가 해결책으로 이어질 것 같지 않으면 더 이상 그 경로를 따라가지 않음으로써 **시도의 횟수를 줄임**
- ▶ 가지치기(Pruning)
- ▶ 불필요한 경로의 조기 차단
- ▶ N! 가지의 경우의 수를 가진 문제에 대해 백트래킹에 가하면 일반적으로 **경우의 수가 줄어들지만** 이 역시 최악의 경우에는 여전히 지수함수 시간(Exponential Time)을 요하므로 처리 불가능
- ▶ 모든 후보를 검사하지 않음

깊이 우선 탐색

- ▶ 모든 경로를 추적
- ▶ N! 가지의 경우의 수를 가진 문제에 대해 깊이 우선 탐색을 하면 **처리 불가능한 문제**
- ▶ 모든 후보를 검사

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 함

반대로 **해답의 가능성성이 있으면 유망하다고 함**

가지치기(Pruning): 유망하지 않은 노드가 포함되는 경로는 더 이상 고려하지 않음

🔍 백트래킹을 이용한 알고리즘의 절차

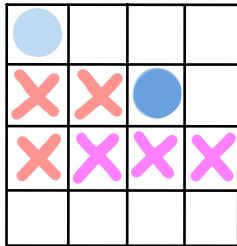
- 1 상태 공간 Tree의 깊이 우선 검색을 실시
- 2 각 노드가 유망한지 점검
- 3 그 노드가 유망하지 않으면, 그 노드의 부모 노드로 돌아가서 검색을 계속

백트래킹 :: n-Queen

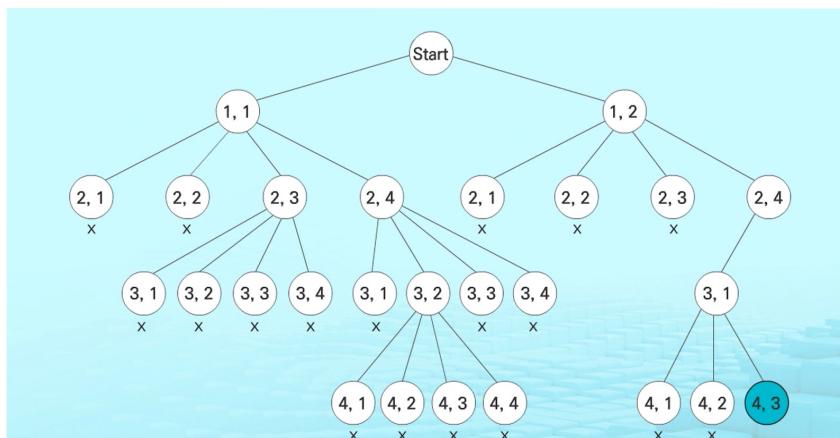
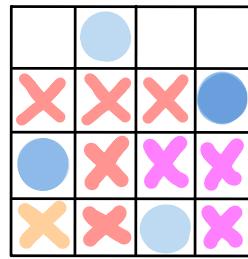
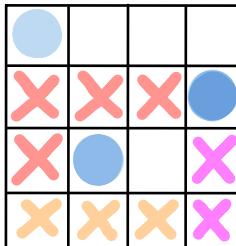
- + $n \times n$ 의 정사각형 안에 n 개의 queen을 배치하는 문제
- + 모든 queen은 자신의 일직선상 및 대각선상에 아무것도 놓이지 않아야 함

```
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)
```

① 1, 1에 있을 때 \Rightarrow 실패



② 1, 2에 있을 때 \Rightarrow 성공!



백트래킹 :: Power Set

- + 어떤 집합의 공집합과 자기자신을 포함한 모든 부분집합
- + 구하고자 하는 어떤 집합의 원소 개수가 n일 때, 부분집합의 개수는 2^n 개

- ✓ 일반적인 백트래킹 접근방법 이용
- ✓ n개의 원소가 들어있는 집합의 2^n 개의 부분집합을 만들 때,
True 또는 False값을 가지는 항목들로 구성된 n개의 리스트를 만드는 방법
- ✓ 리스트의 i 번째 항목은 i 번째 원소가 부분집합의 값인지 아닌지를 나타내는 값

```
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)
```

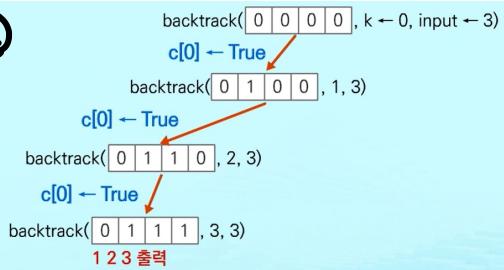
백트래킹 구문

```
def construct_candidates(a, k, input, c) :  
    c[0] = True  
    c[1] = False  
    return 2
```

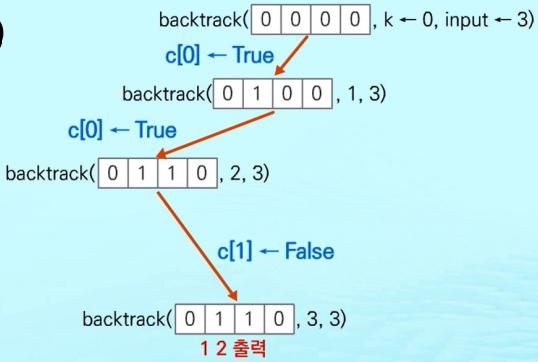
MAXCANDIDATES = 100
NMAX = 100
a = [0] * NMAX
backtrack(a, 0, 3) → 3개의 원소로 이루어진
부분집합 구해줘!

```
def process_solution(a, k) : # 1, 2, 3 출력  
    print(" ", end="")  
    for i in range(k+1) :  
        if a[i] :  
            print(i, end=" ")  
    print("")
```

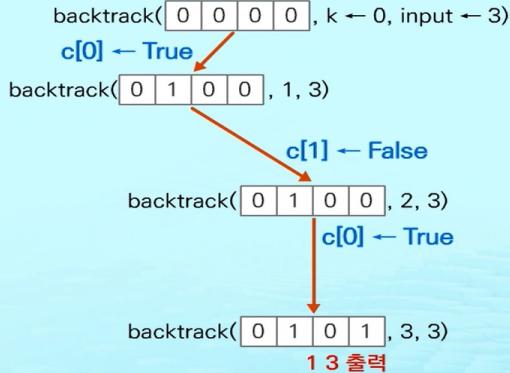
①



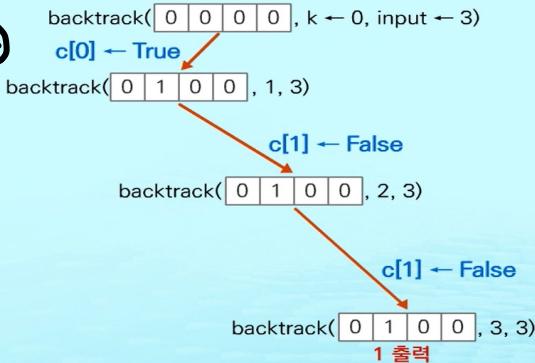
②



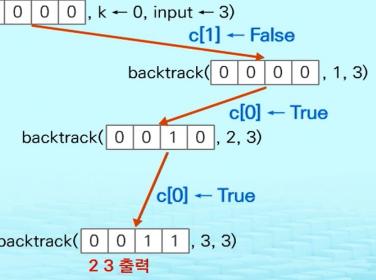
③



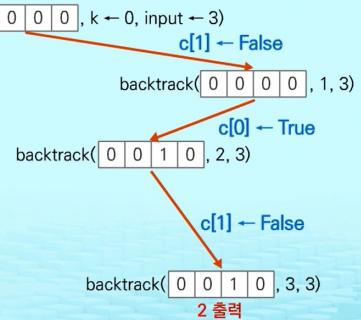
④



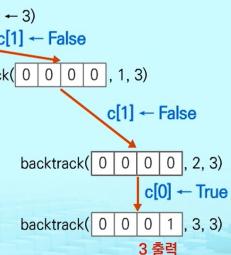
⑤



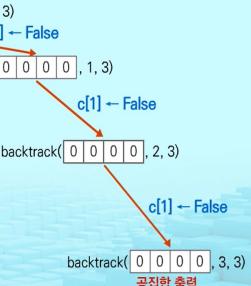
⑥

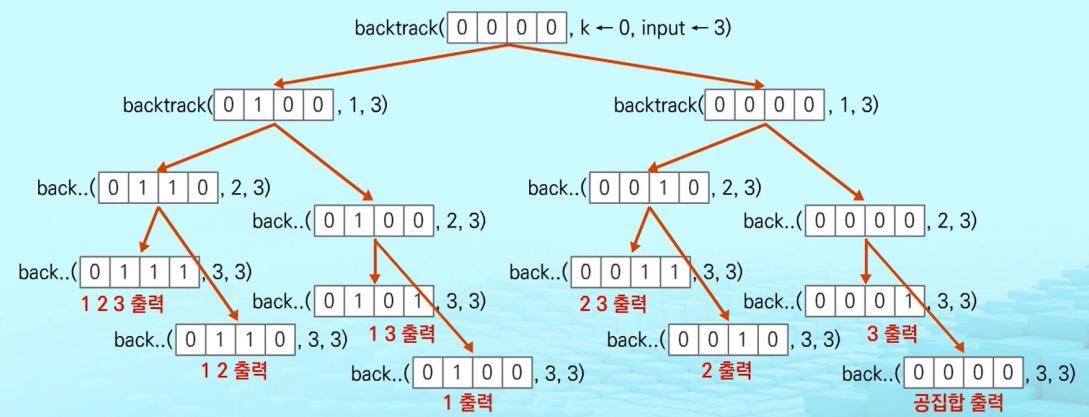


⑦

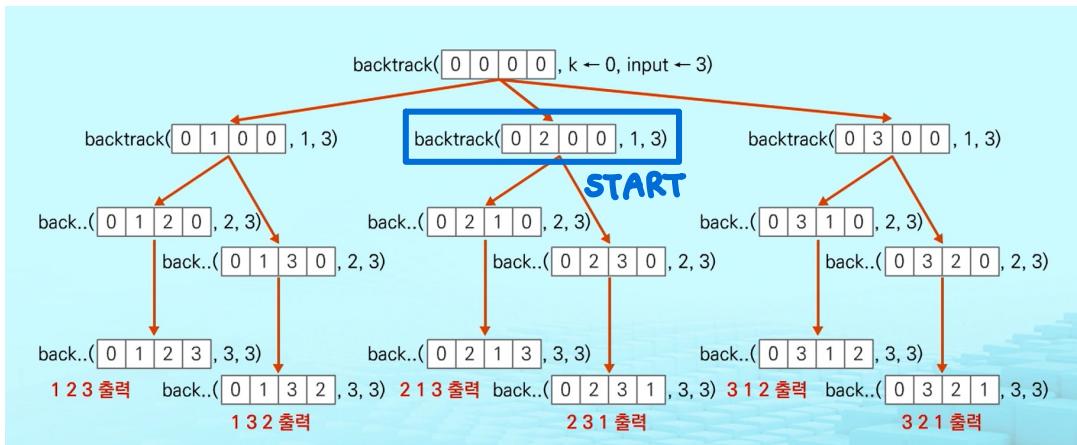


⑧





[부분집합]



[순열]



순열을 구하는 백트래킹 알고리즘

```
def backtrack(a, k, input) :  
    global MAXCANDIDATES  
    c = [0] * MAXCANDIDATES  
  
    if k == input :  
        for i in range(1, k+1) :  
            print(a[i], end=" ")  
        print()  
    else :  
        k+=1  
        ncandidates = construct_candidates(a, k, input, c)  
        for i in range(ncandidates) :  
            a[k] = c[i]  
            backtrack(a, k, input)
```



부분집합과
차이

```
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
```

a[] 

k = 1

input = 3

c[] 

ncandidates = ?

3 a[]  in_perm[] 

k = 1

i = 1

input = 3

c[] 

ncandidates = 1

1 a[]  in_perm[] 

k = 1

i = 2

input = 3

c[] 

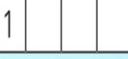
ncandidates = ?

4 a[]  in_perm[] 

k = 1

i = 2

input = 3

c[] 

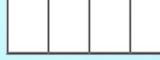
ncandidates = 1

2 a[]  in_perm[] 

k = 1

i = 2

input = 3

c[] 

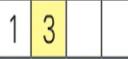
ncandidates = 0

5 a[]  in_perm[] 

k = 1

i = 3

input = 3

c[] 

ncandidates = 2

Stack2 - 3. 분할정복

Quick 정렬의 평균시간복잡도 : $O(n \log n)$
 * 최악은 $O(n^2)$

분할

정복

통합

해결할 문제를 여러개의
작은 부분으로 나눔 !

나눈 작은 문제를 각각 해결

(선택)
해결된 해답을 모음

❖ 거듭 제곱(Exponentiation) 알고리즘: $O(n)$

```
def Power(Base, Exponent):
    if Base == 0: return 1
    result = 1 # Base^0은 1이므로
    for i in range(Exponent):
        result *= Base
    return result
```

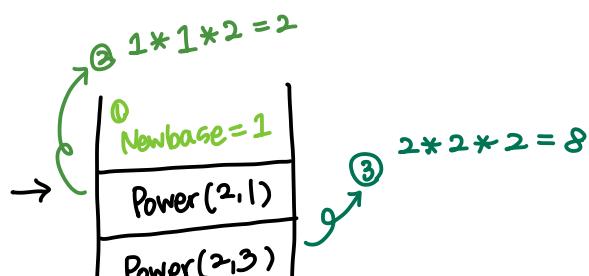
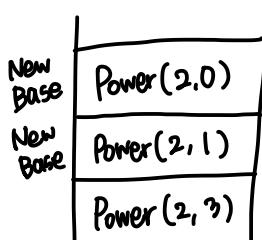
$$\begin{aligned} C^2 &= C \times C \\ C^3 &= C \times C \times C \\ &\dots \\ C^n &= C \times C \times \dots \times C \end{aligned}$$

❖ 분할 정복 기반의 알고리즘 : $O(\log 2n)$

```
def Power(Base, Exponent):
    if Exponent == 0 or Base == 0:
        return 1
    if Exponent % 2 == 0:
        NewBase = Power(Base, Exponent/2)
        return NewBase * NewBase
    else:
        NewBase = Power(Base, (Exponent-1)/2)
        return (NewBase * NewBase) * Base
```

$$\begin{aligned} C^8 &= C \times C \\ C^8 &= C^4 \times C^4 = (C^4)^2 = ((C^2)^2)^2 \\ C^n &= C^{\frac{n-1}{2}} \times C^{\frac{n-1}{2}} \times C = (C^{\frac{n-1}{2}})^2 \times C \\ C^n &= \begin{cases} C^{\frac{n-1}{2}} \cdot C^{\frac{n-1}{2}} & n \text{은 짝수} \\ C^{\frac{(n-1)}{2}} \cdot C^{\frac{(n-1)}{2}} \cdot C & n \text{은 홀수} \end{cases} \end{aligned}$$

ex) Power (2, 3)





퀵 정렬과 합병 정렬의 비교

	합병 정렬	퀵 정렬
공통점	주어진 리스트를 두 개로 분할하고, 각각을 정렬	
차이점	분할할 때, 단순하게 두 부분으로 나눔 각 부분 정렬이 끝난 후, ‘합병’이란 후처리 작업이 필요함	분할할 때, 기준 아이템(Pivot Item)을 중심으로, 이보다 작은 것은 왼편, 큰 것은 오른편 에 위치시킴 각 부분 정렬이 끝난 후, 후처리 작업이 필요로 하지 않음

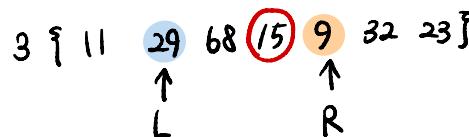
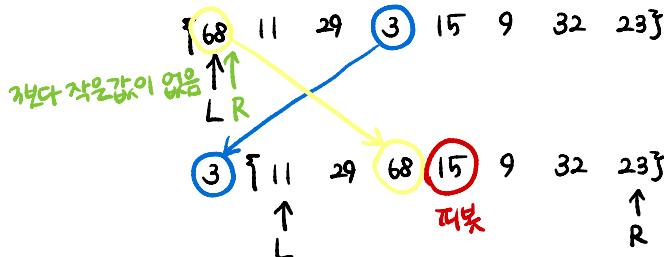


퀵 정렬 알고리즘

```
def quickSort(a, begin, end) :
    if begin < end :
        p = partition(a, begin, end)
        quickSort(a, begin, p-1)
        quickSort(a, p+1, end)
```

```
def partition (a, begin, end) :
    pivot = (begin + end) // 2
    L = begin
    R = end
    while L < R :
        while(a[L]< a[pivot] and L<R) : L += 1
        while(a[R]>=a[pivot] and L<R) : R -= 1
        if L < R :
            if L==pivot : pivot = R
            a[L], a[R] = a[R], a[L]
    a[pivot], a[R] = a[R], a[pivot]
    return R
```

L → **R ←**



↑
 L R
 피봇보다 작은 원소가 = 피봇의 오른쪽에는
 피봇의 왼쪽에 없다 = 피봇보다 큰 수뿐이다!

