**백트래킹(Back Tracking)**

재귀: “자신을 정의할 때 자기 자신을 재창조하는 방법”

Base case: 재귀 호출을 끝내기 위한 경계 조건에 도달하는 경우

Recursive case: base case가 아닌, 추가적으로 재귀 호출을 부르는 경우

**<재귀문 vs 반복문>**

|  |  |
| --- | --- |
| 1. 동일한 함수 내에서 함수를 호출  2. 반복문보다 높은 시간 복잡도  3. 재귀 실행은 일반적으로 조금 느림  4. 종료조건이 없을 시 무한 재귀 가능  5. 재귀에서 스택은 함수가 호출 될 때 로컬 변수를 저장하는데 사용 | 1. 주어진 조건이 참이 될 때까지 반복되는 명령 블록  2. 재귀보다 낮은 시간 복잡도  3. 반복이 일반적으로 조금 더 빠름  4. 조건이 거짓이 되지 않으면 무한 반복  5. 스택이 이용되지 않으며 가독성 낮음 |

그래프 알고리즘에서 배우는 bfs, dfs와 동적 계획법의 top-down, bottom-up 모두 문제에 따라 백트래킹의 재귀냐 반복문이냐의 문제이다.

**6603번 – 로또**

|  |  |  |
| --- | --- | --- |
| **import** itertools **while True**:  file = list(map(int, input().split()))  **if** file == [0]:  **break  else**:  k,num = file[0], file[1:]  num.sort()  ans = []  **for** i **in** itertools.combinations(num, 6):  print(**' '**.join(map(str, i)))  print()   |  | | --- | | #python의 itertools  #itertools.combinations(list, n)  중복 없는 조합을 list에서 n개 선택  #itertools.permutations(list, n)  순열을 list에서 n개 선택 | | **def** search(now, v):  **if** now == 6:  **for** i **in** range(6):  print(ans[i], end = **' '**)  print()  **return  for** i **in** range(v, len(num)):  **if** check[i] == 0:  ans.append(num[i])  check[i] = 1  search(now+1, i)  ans.pop()  check[i] = 0 **while True**:  file = list(map(int, input().split()))  **if** file == [0]:  **break  else**:  k,num = file[0], file[1:]  num.sort()  ans, check = [], [0]\*len(num)  search(0,0)  print()  #재귀로 해결 |

1. 배열의 인덱스를 우선적으로 판단하는 경우 발생하는 오류:

이미 6개를 뽑음과 동시에 배열의 인덱스를 넘어가게 되는 조합의 경우를 무시

2. 6개를 뽑았는지를 우선적으로 판단하는 경우 발생하는 오류:

배열의 인덱스를 넘어가게 되지 않아도 6개를 뽑으면 조합의 경우를 고려

배열의 인덱스를 넘어가게 되도 6개를 뽑았다면 조합의 경우를 고려

즉, 여기서 주의해야 하는 것은 base case가 여러 개일 수 있으며, base case사이의 관계 또한 고려해 주어야 한다는 것이다.

#1 base case: 6개를 뽑았다면 종료

#2 base case: 현재 인덱스가 배열의 범위를 벗어난다면 종료

#1 recursive case: 현재 idx != n 이고 개수가 6개가 안되었을 때 현재 idx를 뽑는 경우

#2 recursive case: 현재 idx != n 이고 개수가 6개가 안되었을 때 현재 idx를 뽑지 않는 경우

**2529번 – 부등호**

1. 부등호 관계를 만족시키지 못하는 순간 다음에 어떤 수가 오더라도 절대 정답이 될 수 없다.

2. 부등호 관계를 만족시키면 다음에 어떤 수가 올 수 있고 정답이 될 수도 있고 아닐 수도 있다.

필요한 정보: 1. 현재 인덱스 2. 현재까지의 순서

Base Case: 현재 인덱스가 k+1보다 클 수 없음

Recursive Case: 중복이면 뽑지 않고 중복x+부등호 만족이면 뽑고 중복x + 부등호 만족이 아니면 안 뽑음

만들려는 함수들

1. check(x,y,op):

현재 부등호가 op일 때, 두 정수 x, y가 부등호를 만족하는지 아닌지 리턴하는 함수

2. go(idx, seq):

현재 인덱스가 idx일 때 현재 순서가 seq

|  |
| --- |
| **import** sys n = int(input()) file = list(map(str, sys.stdin.readline().split())) **def** check(x,y,op):  **if** op == **'>'**:  **if** x <= y:  **return False  else**:  **if** x >= y:  **return False  return True  def** go(idx, seq):  **global** min, max  **if** idx == n+1:  **if** int(seq) < int(min):  min = seq  **if** int(seq) > int(max):  max = seq  **return  for** i **in** range(10):  **if** visit[i] == 0:  **if** idx == 0 **or** check(int(seq[idx-1]), i, file[idx-1]) == **True**:  visit[i] = 1  go(idx+1, seq+str(i))  visit[i] = 0   visit = [0]\*10 min, max = **'1000000000000'**, **'0'** go(0, **''**) print(max) print(min) |

vist 리스트에서 재귀 호출을 한 이후에 다시 그 방문 배열을 초기화해 주는 것, 그리고 재귀 호출을 부를 때 조건에 맞는 숫자를 재귀로 불러오는 것이 제일 중요한 두개의 포인트이다.

그리고 min, max는 극단적으로 int로 바꿀 때 각각 크고 작은 값으로 초기 값을 설정한 것이다.

**1248번 – 맞춰봐**

|  |  |  |  |
| --- | --- | --- | --- |
| - | + | 0 | + |
|  | + | + | + |
|  |  | - | - |
|  |  |  | + |

위의 표와 같은 상황으로 문자가 주어졌을 때 세로 방향으로, 즉 행을 기준으로 살펴보면 너무 구현이 복잡해 질 수 있어서 이번에는 열을 기준으로 살펴보자. 즉, 가로 방향으로 이동하면서 순서대로 가능한 숫자를 찾아 나선다.

처음이 아니라 조건을 만족하지 않는 순간으로 다시 돌아가는 back tracking을 실행한다.

필요한 정보: 1. 현재 인덱스 2. 현재 인덱스가 i번째일 떄, board[k][i]의 부호 조건을 만족하는지

Base Case: 현재 인덱스가 n이상인 경우

Recursive Case: 현재 인덱스가 n이 아니거나 부호 조건을 만족 할 때 재귀 호출을 한다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **import** sys n = int(input()) board = [[[]**for** \_ **in** range(n)]**for** \_ **in** range(n)] file = list(map(str, input().strip())) k = 0 ans = [0]\*n **for** i **in** range(n):  **for** j **in** range(i, n):  **if** file[k] == **'-'**:  board[i][j].append(-1)  **elif** file[k] == **'+'**:  board[i][j].append(1)  **else**:  board[i][j].append(0)  k += 1 **def** check(idx):  s = 0  **for** i **in** range(idx, -1,-1):  s += ans[i]  **if** board[i][idx][0] == 0:  **if** s != 0:  **return False  elif** board[i][idx][0] > 0:  **if** s <= 0:  **return False  elif** board[i][idx][0] < 0:  **if** s >= 0:  **return False  return True  def** go(idx):  **if** idx == n:  **return True  if** board[idx][idx][0] == 0:  ans[idx] = 0  **return** check(idx) **and** go(idx+1)  **for** i **in** range(1, 11):  ans[idx] = int(i\*board[idx][idx][0])  **if** check(idx) **and** go(idx+1):  **return True  return False** go(0) **for** i **in** ans:  print(i, end= **' '**) | |  |  |  |  | | --- | --- | --- | --- | | - | + | 0 | + | |  | + | + | + | |  |  | - | - | |  |  |  | + |   위의 화살표와 같은 방향으로 확인해 나가는데, ans라는 리스트에 각 index에 해당하는 값을 넣는다. 이후 check라는 함수를 통해 각 열의 파란색 화살표 방향으로 조건에 맞는지 확인해 준다. 만약에 True가 나온다면 계속 이어가는데 무조건 ans[idx]는 board[idx][idx][0]의 조건에 맞춘다. 즉, board[idx][idx][0]이 0이면 ans[idx]도 0, 양수면 양수, 음수면 음수인 것이다.  이후 재귀적으로 go 함수를 실행시켜 나가다가 모든 조건에 맞는 순자 조합이 있으면 하나의 답만 알면 된다고 했으니 그 것을 출력 시켜주면 된다. |
| go(idx): 현재 idx번째 수를 채우는 함수  Base Case: check(idx)를 만족해야 go(idx+1)의 재귀 호출이 가능하므로 idx = n이면 board의 모든 부호 조건을 만족했으므로 답을 찾았다는 의미로 True 리턴  Recursive Case 1: 현재 idx번쨰 수가 n미만이고 0이면 바로 조건 검사 후 재귀 호출  Recursive Case 2: 현재 idx번째 수가 n미만이고 0이 아니면 idx번째 부호를 붙인 후 조건 검사 후 재귀 호출 | | |

**<결론>**

1. 모든 경우가 n! 이거나 일 때 완전 탐색을 의심해 보자(순열, 조합, 백트래킹 등)

2. n이 너무 커서 모든 경우를 탐색 할 수 없는데 가지치기 하다 보면 경우의 수가 충분히 완전 탐색을 할 수 있을 정도로 줄어드는 경우

3. 뽑는다, 뽑지 않는다/속한다, 속하지 않는다/간다, 가지 않는다 의 느낌이 드는 문제의 경우, 1번과 2번을 만족한다면 백트래킹을 의심해 보자.

4. 중복을 허용하지 않는 백트래킹의 경우 재귀 호출이 끝나면 반드시 방문 여부를 초기화 시키자.

5. base case간의 우선 순위도 고려하자.