알고리즘및실습

제2장 전수조사

1. 전수조사 방법

문제가 주어졌을 때 모든 경우를 다 조사하면 항상 정확한 답을 찾을 수 있다. 하지만 조사해야 하는 전체 경우의 수가 제한적일 경우에만 사용 가능한 방법이다. 예를 들어 10명의 학생을 줄을 세우는 총 경우의 수는 $10! = 3,628,800$ 이다. 이 정도는 컴퓨터에게는 별로 큰 수는 아니다. 물론 경우의 수가 $n!$일 때, n이 조금만 더 커지면 컴퓨터를 이용하여도 빠른 시간 내에 계산할 수 없다. 즉, 컴퓨터 프로그래밍을 통해 조사하는 것이기 때문에 직관적으로 생각한 것보다는 범위가 클 수 있다.

보통 프로그래밍 경시대회의 경우 제출한 소스가 적절한 시간 복잡도를 갖추어졌는지 확인하기 위해 제출된 소스의 실행 시간을 제한 시간과 비교한다. 이때 보통 1초를 제한 시간으로 사용한다. 예를 들어 문제에 주어진 최대 테스트케이스 수가 T이고, 입력의 최대 크기가 n이며, 제출한 소스의 시간 복잡도가 $O(n^2)$이면 $T \times n^2$이 10^8보다 작으면 주어진 1초 제한 시간을 통과할 수 있다. 예를 들어 최대 테스트케이스 수가 100이고, 최대 입력 크기가 1,000이면 $T \times n^2 = 10^8$이므로 통과할 수도 있고, 통과 못할 수도 있다. 통과를 못하는 경우는 빅O 계산에서 생략된 상수 계수와 낮은 차수 항 때문일 수 있다.

전수조사 알고리즘은 보통 재귀 알고리즘으로 구현한다. 이와 같이 구현된 재귀 알고리즘은 10장에서 살펴보는 동적 프로그래밍(dynamic programming) 중 메모이제이션 기법을 적용하여 성능을 기계적으로 개선할 수 있으며, 12장, 13장에서 살펴보는 되추적(backtracking)과 분기한정법(branch-and-bound) 기법을 통해 개선할 수도 있다. 되추적과 분기한정법은 모든 경우 중 조사하지 않아도 되는 경우를 빠르게 배제할 수 있도록 해주는 기법이다.

전수조사 방법으로 알고리즘을 설계할 때 주의할 점은 조사하는 과정을 잘못 작성하면 동일한 경우를 여러 번 검사할 수 있다. 이 경우 조사를 완료한 후에 중복된 답을 제거할 수 있지만 더 효과적인 방법은 동일한 경우를 중복 검사하는 경우가 없는 조사 방법을 사용하는 것이다.

2. 조합 문제

문제 중 특정 조건을 만족하는 조합을 찾아야 하는 경우가 종종 있다. 효과적이지 않을 수 있지만 가능한 모든 조합을 찾아 이 문제를 해결할 수도 있다. 예를 들어 크기가 n인 영소문자로만 구성된 문자열이 주어졌을 때, 이 문자열에서 3개의 문자를 골라 모든 문자가 서로 다른 크기가 3인 문자열의 수를 찾는 문제를 생각하여 보자. 이 문제는 모든 조합을 찾아 해당 조합의 문자가 모두 다르면 이 문자로 만들 수 있는 6개 문자열을 집합 자료구조에 삽입하여 해결할 수 있다. 당연히 이 방법은 효과적인 방법은 아니다.

하지만 종종 n개의 원소 중 $m(< n)$개를 고르는 모든 조합을 찾아야 할 경우도 있다. 이 문제를 조합 문제라 한다.

조합 문제

- 입력. 정수 n과 정수 m, $1 \leq m \leq n$
- 출력. 1부터 n까지 수 중 m개를 고르는 모든 조합

수학 시간에 학습하였지만 n개 중 m개를 고르는 총 조합의 수는 다음과 같다.

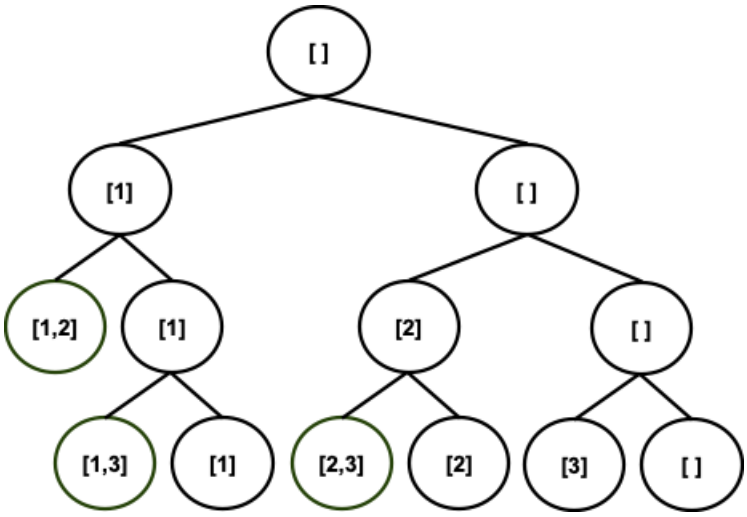
$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

따라서 n과 m에 따라 조합의 수가 너무 많아 컴퓨터를 이용하여도 모든 조합을 검사하여 문제를 해결하는 것은 가능하지 않을 수 있다. 하지만 작은 n과 m에 대해 모든 조합을 찾아주는 알고리즘을 구현해 보면 재귀 알고리즘을 구현하는 방법 나아가 전수조사 알고리즘을 구현하는 방법을 학습하는데 많은 도움이 될 수 있다.

먼저 조합 문제 자체를 생각하여 보자. 선택 가능한 각 수는 찾은 조합에 포함될 수 있고, 포함되지 않을 수 있다. 따라서 각 요소를 포함하는 경우와 포함하지 않는 경우 두 갈래로 조사하여 전체 조합을 찾을 수 있다. 보통 프로그래밍에서는 두 개의 재귀 호출을 통해 이 갈래를 만들 수 있다. 재귀 호출이 이루어지면 다음 수에 대해 고려하면 되며, 재귀 호출의 종료 조건은 찾은 조합의 크기가 m이 되는 경우이다. 이를 알고리즘을 표현하면 다음과 같다.

알고리즘 2.1 조합 전주조사 알고리즘

```
1: function COMBINATION(n, m)
2:     ret := []
3:     C := [1] × n
4:     COMBINATION(ret, n, m, 1, 1, C)
5:     return ret
1: procedure COMBINATION(ret, n, m, size, i, C)
2:     if size = m + 1 then
3:         ret.PUSHBACK(C.CLONE())          ▷ 복제의 필요성은 구현 언어의 특성에 의해 결정됨
4:     if i ≥ n then return
5:     C[size] := i
6:     COMBINATION(ret, n, m, size + 1, i + 1, C)      ▷ i 포함
7:     COMBINATION(ret, n, m, size, i + 1, C)          ▷ i 미포함
```

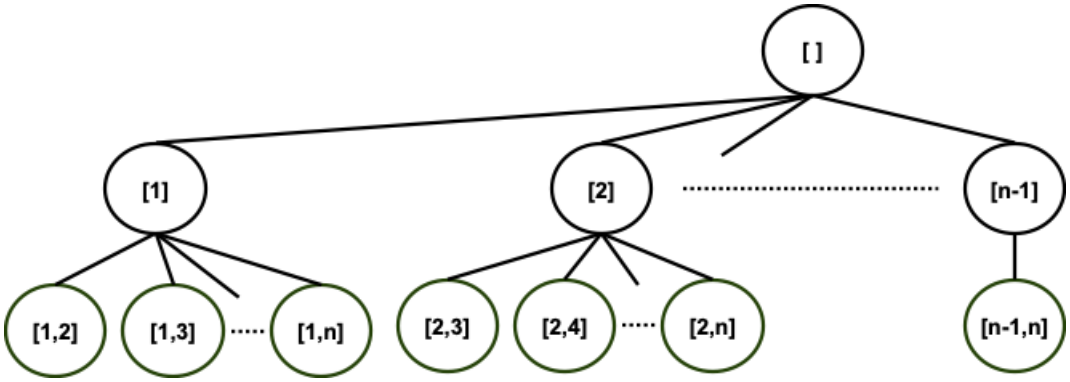


<그림 2.1> 알고리즘 2.1의 재귀 호출 트리($m = 2$)의 일부분

제시된 알고리즘의 재귀 호출 트리의 일부분은 그림 2.1과 같다. 각 레벨에서 특정 숫자의 포함 여부를 고려하기 때문에 이 트리의 높이는 n이며, 이진 트리이다. 트리의 최대 높이는 m과 무관하지만 m에 따라 각 갈래의 중단

위치가 정해진다. n과 m이 주어졌을 때 전체 노드 수를 정확하게 파악하는 것은 간단하지 않다. 하지만 트리 높이가 n이고 완벽 이진 트리이면 전체 노드 수는 2^n이므로 트리의 노드 수는 2^n보다는 적을 것이고, 비재귀 과정에 일어나는 일은 n과 무관하게 항상 일정하므로 이 알고리즘의 빅O는 대략적으로 $O(2^n)$이다. 즉, 지수 시간이 요구되는 알고리즘이다.

재귀 알고리즘의 경우 여러 방법을 통해 성능을 향상할 수 있지만 알고리즘 2.1은 필요한 모든 경우를 중복 없이 조사하고 시간 복잡도 측면에서는 개선할 여지가 별로 없다. 하지만 알고리즘 2.1에 제시된 방법이 조합을 구하는 유일한 전수 조사 방법은 아니다. 우리는 재귀 알고리즘을 만든 후 만든 재귀 알고리즘의 재귀 호출 트리를 그렸다. 이것을 거꾸로 할 수 있다. 재귀 알고리즘을 만들어야 할 때 재귀 호출 트리를 그리고 이 트리를 바탕으로 재귀 알고리즘을 구현할 수 있다.



<그림 2.2> 조합을 구하는 재귀 호출 트리($m = 2$)의 일부분

조합을 구하는 재귀 호출 트리를 그림 2.2와 같이 구성할 수 있다. 이 트리와 이전 트리를 비교하면 트리 높이가 n에 좌우하지 않고 m에 좌우한다. m은 n보다 상대적으로 작기 때문에 트리 높이가 짧아진다. 또 전체 노드의 수는 n^m보다는 적다. 트리의 높이가 짧으면 필요한 공간이 줄어든다. 이 트리를 구현한 알고리즘은 다음과 같다.

알고리즘 2.2 조합 전주조사 알고리즘

```
1: function COMBINATION(n, m)
2:     ret := []
3:     C := []
4:     COMBINATION(ret, n, m, C)
5:     return ret
1: procedure COMBINATION(ret, n, m, C)
2:     if m = 0 then
3:         ret.PUSHBACK(C.CLONE())
4:     next := 0 if C.ISEMPTY() else C.BACK() +1
5:     for i := next to n − 1 do
6:         C.PUSHBACK(i)
7:         COMBINATION(ret, n, m − 1, C)
8:         C.POPBACK()
```

재귀 호출 트리가 주어졌을 때 실제 호출되는 순서는 재귀 호출에 대한 깊이 우선 탐색을 하는 순서와 같다. 따라서 밑으로 계속 내려가다가 더 이상 내려갈 수 없으면 다시 위로 되돌아 와야 한다. 하지만 이때 내려갔을 때 한 일을 되돌려야 한다. 알고리즘 2.2에 제시된 두 번째 함수 9번째 줄이 이 역할을 하는 것이며, 이 부분은 전수조사에서 흔하게 나타나는 패턴이다.

3. 합계 문제

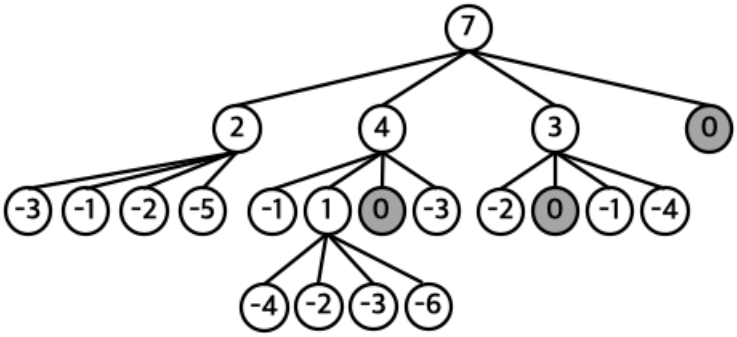
컴퓨터공학의 유명한 문제 중 하나가 부분집합 합계(subset sum) 문제이다. 이 문제를 변형한 몇 가지 문제를 살펴보고자 한다. 이 절에서는 이들을 전수조사 방법으로 해결한다. 10장에서는 이들을 전수조사 대신에 동적 프로그래밍을 이용하여 해결한다.

첫 번째 살펴볼 문제는 다음의 정의된 cansum 문제이다.

cansum 문제

- 입력. 목표 정수 $m(\geq 0)$과 n개의 서로 다른 양의 정수 x_i
- 출력. 각 x_i를 원하는 만큼 합하여 목표 정수 m을 만들 수 있으면 **`true`** 없으면 **`false`**

예를 들어 목표 정수가 7이고, **`[5, 3, 4, 7]`** 4개의 정수가 주어지면 $(3,4)$, $(4,3)$, 7 등 여러 가지 방법으로 목표 정수를 만들 수 있기 때문에 이 문제의 답은 **`true`**이다.



<그림 2.3> $m = 7$, 주어진 정수 목록이 **`[5, 3, 4, 7]`**일 때 cansum을 구하는 재귀 호출 트리

이 문제는 목표 수에서 각 정수를 반복적으로 빼서 0을 얻으면 **`true`**를 반환하고, 모든 경우에 0을 얻지 못하면 **`false`**를 반환하면 된다. 이를 구현하는 재귀 호출 트리를 그리면 그림 2.3과 같다. 이 트리의 최대 높이는 주어진 정수에 1이 있는 경우이며, 이때 트리의 높이는 m이다. 또 각 노드에서 n개 재귀 호출이 이루어진다. 따라서 전체 노드의 수는 m^n보다 적다. 따라서 이 재귀 호출 트리를 다음과 같이 그대로 구현하면 이 알고리즘의 빅O는 $O(m^n)$이다. 또 공간 복잡도는 트리 높이에 의한 스택 공간과 입력 배열을 고려하면 빅O는 $O(m+n)$이다. 이 문제에서 목표 정수가 0이면 답은 항상 **`true`**이다.

알고리즘 2.3 cansum 알고리즘

```
1: function CANSUM(m, A[])
2:     if m < 0 then return false
3:     else if m = 0 then return true
4:     for each x ∈ A do
5:         if CANSUM(m − x, A) then return true
6:     return false
```

두 번째 문제는 다음의 정의된 countsum 문제이다.

countsum 문제

- 입력. 목표 정수 $m(\geq 0)$과 n개의 서로 다른 양의 정수 x_i
- 출력. 각 x_i를 원하는 만큼 합하여 목표 정수 m을 만들 수 있는 조합의 수

이 문제는 cansum과 달리 중간에 중단할 수 없고, 모든 경우를 무조건 다 해보아야 한다. cansum과 비교하여 중요하게 달라지는 부분은 반환값과 내부 반복문 과정에서 중단되는 경우 없이 모든 경우를 진행한다는 점이다.

알고리즘 2.4 countsum 알고리즘

```
1: function COUNTSUM(m, A[])
2:     if m < 0 then return 0
3:     else if m = 0 then return 1
4:     count := 0
5:     for each x ∈ A do
6:         count+= COUNTSUM(m − x, A)
7:     return count
```

성능은 최악의 경우에 대한 분석이므로 성능은 cansum과 동일하다. 시간 복잡도의 빅O는 $O(m^n)$이고, 공간 복잡도의 빅O는 $O(m+n)$이다.

세 번째 문제는 다음의 정의된 howsum이다.

howsum 문제

- 입력. 목표 정수 $m(\geq 0)$과 n개의 서로 다른 양의 정수 x_i
- 출력. 각 x_i를 원하는 만큼 합하여 목표 정수 m을 만들 수 있는 임의의 조합

이 문제는 countsum처럼 중간에 중단할 수 있다. 하지만 조합 자체를 만들어야 하므로 리스트를 반환해야 한다.

알고리즘 2.5 howsum 알고리즘

```
1: function HOWSUM(m, A[])
2:     if m < 0 then return null
3:     if m = 0 then return []
4:     for each x ∈ A do
5:         ret := HOWSUM(m − x, A)
6:         if list ≠ null then
7:             ret.PUSHBACK(x)
8:             return ret
9:     return null
```

이 알고리즘의 시간 복잡도는 여전히 기존 cansum, countsum과 마찬가지로 $O(n^m)$이다. 하지만 공간 복잡도는 `ret` 때문에 기존 알고리즘과 같지 않다. `ret` 리스트의 공간 복잡도는 사용하는 자료구조, 생성하는 방법 등에 따라 조금 차이가 있다. 보통은 동적 배열 기반 리스트를 많이 사용할 것이며, howsum의 반환 타입을 고려하면 이 자료구조 자체도 동적 생성할 것이다. 자바 언어에서 **`ArrayList`**를 사용한다고 생각하면 이해하기 쉬울 것이다. 이 경우 중간에 확장하는 비용을 없애기 위해 생성할 때 용량이 m이 되도록 공간을 확보하여 사용하는 것이 시간 복잡도 측면에서는 효과적이다. 하지만 용량이 m이 되도록 확보하면 많은 경우 공간이 낭비될 수 있어, $m/2$ 등 초기 용량으로 m보다 적은 공간을 확보하여 사용할 수도 있다.

공간 복잡도를 간편하게 분석하기 위해 `ret`은 생성할 때 용량이 m이 되도록 확보한다고 가정하자. 그러면 입력 배열 A, `ret`, 함수 스택에 의해 howsum 알고리즘의 공간 복잡도는 $O(2m+n)$이 된다. 일반적으로 계수는 제거하기 때문에 $O(m+n)$으로 표현하는 것이 올바른 표현일 수 있지만 다른 알고리즘과 비교를 위해 howsum 알고리즘의 공간 복잡도는 계수를 제거하지 않고 $O(2m+n)$으로 표현한다.

네 번째 문제는 다음의 정의된 bestsum이다.

bestsum 문제

- 입력. 목표 정수 $m(\geq 0)$과 n개의 서로 다른 양의 정수 x
- 출력. 각 x를 원하는 만큼 합하여 목표 정수 m을 만들 수 있는 가장 짧은 조합 (같은 길이의 조합이 있으면 그들 중 임의로 출력)

이 문제는 howsum처럼 중간에 중단할 수 없지만 반환은 howsum처럼 조합을 반환해야 한다.

알고리즘 2.6 bestsum 알고리즘

```
1:  function BESTSUM(m, A[])
2:      if m < 0 then return null
3:      if m = 0 then return []
4:      best := null
5:      for each x ∈ A do
6:          list := BESTSUM(m − x, A)
7:          if list ≠ null then
8:              if best != null or LEN(best) > LEN(list) + 1 then
9:                  list.PUSHBACK(x)
10:                 best := list
11:     return best
```

이 알고리즘의 시간 복잡도는 기존 알고리즘들과 마찬가지로 $O(n^m)$이다. 공간 복잡도는 howsum에 비해 더 많이 사용한다. 알고리즘이 진행되는 동안 계속 유지하는 `best` 리스트와 현재 만들고 있는 `list` 두 개를 사용하고 있다. 둘 다 생성할 때부터 용량이 m이 되도록 생성한다고 가정하고 분석하면 이 알고리즘의 공간 복잡도는 $O(3m + n)$이 된다.

실제 bestsum의 경우 답에 해당하는 조합마다 용량이 m인 리스트를 생성한다. 자바 언어로 구현한다고 가정하고 생각해 보면 특정 순간에 2개의 리스트만 존재하고, 나머지는 쓰레기가 되었거나 아직 생성 전 상태가 된다. 이것까지 분석하는 것은 너무 복잡하다. 따라서 10번째 줄 전에 `best`가 가리키던 리스트가 사용 중인 공간은 반납한다고 가정하고 분석할 필요가 있다.

이 절에서 제시한 모든 알고리즘은 지수 시간 알고리즘이므로 n과 m에 따라 답을 구하기 위해 너무 많은 시간이 소요될 수 있다. 이때 성능을 조금 개선하기 위해 내림차순이나 오름차순으로 정렬하는 것을 생각할 수 있다. 또 정렬한 후에 1 또는 2가 있는지 보고 이를 이용하여 cansum, howsum은 예외적으로 빠르게 답변을 할 수 있다.

4. 게임판 덮기

$H \times W$ 크기의 게임판이 있고, 게임판은 검정 칸과 흰 칸으로 구성된 격자 모양을 하고 있다. 이 중 모든 흰 칸을 세 칸짜리 L자 모양의 블록으로 덮고 싶다. 블록은 자유롭게 회전 가능하지만 겹치지 않아야 하며, 검은 칸이나 게임판 밖으로 나갈 수 없다. 개임판에 있는 흰 칸의 수는 50을 넘지 않는다.

게임판 덮기 문제

- 입력. 정수 H와 W, '#'와 '.'으로 구성된 H줄
- 출력. 덮을 수 있는 경우의 수

게임판을 논리형 2차원 배열로 표현할 수 있다. 검정 칸은 **`false`**, 흰 칸은 **`true`**로 설정한 후에 **`true`**인 칸을 모두 **`false`**로 바꿀 수 있으면 덮을 수 있는 경우가 된다. 이 문제는 모든 경우를 해보면 된다. 흰 칸의 수가 50을

넘지 않으므로 전수조사를 하더라도 충분히 빠른 시간 내에 조사를 할 수 있다. L자 모양의 블록을 회전하면 총 4가지 모양이 나온다. 따라서 한 위치에서 L자 모양을 사용할 수 있는 경우의 수는 4가지이다. 또 한번 L자 모양의 블록을 덮으면 흰 칸이 3개 줄어든다. 따라서 흰 칸의 수가 3의 배수가 아니면 덮을 수 없다. 최대 48개의 흰 칸이 있을 때 가장 많은 L자 모양의 블록을 사용하게 되며, 총 조사해야 하는 경우의 수는 16^4이다.

앞서 언급한 바와 같이 잘못 전수조사를 하면 똑 같은 경우를 중복 조사할 수 있다. 따라서 중복 조사하는 경우가 없도록 조사하는 방법을 찾아야 한다. 이 문제의 경우에는 맨 윗 줄부터 오른쪽으로 검사하면서 흰 칸을 만났을 때 4가지 가능한 L모양을 덮어나가면 중복 조사하는 경우가 없이 모든 경우를 조사할 수 있다. 이 문제도 알고리즘 2.2처럼 재귀 호출하고 되돌아 오면 덮은 부분을 다시 `true`로 바꾸어 다음 조합을 검사할 수 있도록 해야 한다. 대략적 알고리즘은 다음과 같다.

알고리즘 2.7 boardCover 알고리즘

```
1:  function coverCount(H, W, board[][])
2:      if countWhiteCell(H, W, board)%3 != 0 then return 0
3:      return coverCount(H, W, board, 1, 1)
4:  function coverCount(H, W, board[][], startR, startC)
5:      nextR, nextC := findNextWhiteCell(H, W, board, startR, startC)
6:      if nextR = −1 then return 1                    ▷ 흰 칸이 없으면 덮는데 성공한 경우임
7:      count := 0
8:      for i := 1 to 4 do                             ▷ 4가지 L 모양을 해당 위치에서 시도함
9:          if putCover(H, W, board, nextR, nextC, L[i]) then
10:             count +=coverCount(H, W, board, nextR, nextC + 1)
11:             removeCover(H, W, board, nextR, nextC, L[i])
12:     return count
```

5. 3자리 짝수 찾기

3자리 짝수 찾기 문제

- 입력. n개의 정수 d_I, $3 \leq n \leq 100$, $0 \leq d_i \leq 9$
- 출력. 주어진 정수로 만들 수 있는 독특한 3자리 짝수를 오름차순으로 출력

이 문제를 이전 cansum 문제처럼 입력으로 받은 `n`개의 숫자로 만들 수 있는 모든 3자리 숫자를 만든 후에 이 중의 짝수만 리스트에 저장한 다음 이것을 정렬하여 문제를 해결할 수 있다. 이와 같은 방식으로 구현하더라도 모든 3자리 숫자를 만드는 것이 아니라 짝수만 만드는 것이 효과적이며, 만든 후 정렬하기 보다는 작은 수부터 만들어 전수조사 후에 다시 정렬할 필요가 없도록 하는 것이 효과적이다. 실제 주어진 n개 숫자를 모두 사용할 필요가 없다. 특정 수는 최대 3개만 사용할 수 있으므로 3개 이상 등장하는 수의 경우에는 3개만 남기고 나머지는 모두 제거하면 전수조사해야 하는 경우를 대폭 줄일 수 있다. 이렇게 전처리할 경우에는 빈도수 배열을 사용하는 것이 효과적이다.

하지만 이와 같은 문제의 경우 답의 범위를 생각해 볼 필요가 있다. 만들어야 하는 것은 3자리 짝수이다. 따라서 가능한 수는 100에서 998이며, 총 450개 밖에 없다. 따라서 거꾸로 100부터 2씩 증가하면서 해당 수를 주어진 정수로 만들 수 있는지 여부만 조사해 보면 더 쉽게 해결할 수 있다.

퀴즈

1. 전수조사 방법과 관련된 다음 설명 중 틀린 것은?

   ① 조사해야 하는 경우의 수가 적을 경우에만 사용할 수 있는 방법이다.

② 정확한 답을 주지 않을 수 있다.

③ 컴퓨터를 이용하여 해결하는 것이기 때문에 범위가 생각보다 크다.

④ 대충 가능한 여부를 검토할 때 알고리즘의 빅O에 입력 크기를 이용하여 계산한 결과가 10^8 이하이면 1초 이내에 수행이 가능하다.

2. n개의 도시가 있고, 모든 도시에서 다른 모든 도시로 항상 이동이 가능하다. 외판원 문제는 한 도시에서 출발하여 모든 도시를 방문하여 출발도시로 돌아오는 가장 짧은 경로를 찾는 것이다. 이 문제를 전수조사로 찾고자 한다. 가능한 경로의 수는 다음 중 어떤 것인가?

① n^2

② $n!$

③ 2^n

④ $2n$

3. 피보나치 수는 다음 식을 이용하여 계산한다.

$$f(n) = f(n-1) + f(n-2), f(1) = f(2) = 1$$

피보나치 수를 계산하기 위해 다음과 같은 함수를 정의하였다.

```
long fib(long n){
    if(n <= 2) return 1;
    else return fib(n - 1) + fib(n - 2);
} // n>=1
```

이 알고리즘의 재귀 호출 트리를 그리면 이 트리(루트에서 가장 먼 후손 노드까지의 경로 길이)의 높이는 어떻게 되는가? 또 재귀 호출을 제외하고 이 알고리즘은 덧셈 하나, 비교 연산 하나가 필요하다. 그러면 이 알고리즘의 빅O는 어떻게 되는가?

① $n, O(2n)$

② $n, O(n^2)$

③ $n-2, O(2^n)$

④ $n-2, O(n^2)$

연습문제

1. 알고리즘 2.4에 배열 $[2, 5, 2, 1, 2]$와 목표 정수 5가 입력으로 주어지면 답으로 $[2, 2, 1]$, $[2, 1, 2]$, $[2, 1, 2[$, [5] 4가지 조합을 찾아준다. 이때 같은 수로 구성된 조합은 중복으로 판단하면 답은 4가 아니라 2가 되어야 한다. 알고리즘 2.4를 수정하여 중복된 조합을 제거하는 알고리즘을 제시하라.

2. 알고리즘 2.6은 해 중에 가장 짧은 해를 찾아준다. 가장 짧은 해 대신에 가장 긴 해를 찾는 알고리즘을 제시하라.

3. 두 개의 영소문자로만 구성된 문자열이 주어진다. 첫 번째 문자열의 임의 순열이 두 번째 문자열의 부분 문자열이면 **true**, 아니면 **false**를 반환하는 알고리즘을 제시하라. 예를 들어 “okare”, “koreatech”이 주어지면 “okare”의 순열 중 하나가 “korea”이므로 **true**를 반환해 주어야 한다.