# 2020년도 제15회 5번

문제 초기 코드 테스트 케이스

문자열에 등장하는 문자의 종류가 한정되어 있는 경우에는 연달아 등장하는 문자의 출현 횟수를 기록함으로써 문자열의 저장에 소요되는 기억 공간을 줄일 수 있습니다. 특히, 등장하는 문자들이 두 가지 뿐일 때에는 두 가지의 문자가 번갈아서 나타난다고 볼 수 있기 때문에 기억 공간을 더욱 줄일 수 있습니다.

"0"과 "1"로만 이루어진 비트열의 경우에는 같은 비트가 연속해서 등장하는 횟수들을 기록하여 저장 공간을 줄일 수 있습니다. 예를 들어, 첫 문자가 "0"으로 시작하는 "00011110"의 경우, "CDA"라고 표기하여 저장 공간을 줄일 수 있는데, "C"는 "0"이 3개 연속으로 등장하였다는 의미이고, "D"는 "1"이 4개, "A"는 "0"이 1개가 연속하여 등장했다는 의미입니다.

만약, 비트열이 "0"이 아니라 "1"로 시작하는 경우에는 저장 공간의 제일 앞에 "1"을 붙여서 혼돈을 방지하도록 합니다. 예를 들어, "110100"의 경우에는 "1BAAB"로 표시하면 됩니다. 주어진 문자열이 "111100100011" 일 때 출력할 문자열은 "1DBACB" 입니다. 이 문자열 압축 알고리즘을 구현하는 함수 solution() 을 완성하세요.

[제한사항]

- 동일한 문자가 1번 등장하는 경우에는 "A", 2번 등장하는 경우에는 "B", ..., 동일한 문자가 26회 연달아 등장하는 경우에는 "Z"로 대응되며, 동일한 문자가 27회 이상 연달아 등장하는 경우는 입력으로 주어지지 않습니다.
- 입력으로 주어지는 문자열 src 의 길이는 1 이상 10,000 이하이며, 이 문자열에는 "0" 과 "1" 외의 문자는 등장하지 않음을 가정합니다.
- 출력되는 문자열에는 대문자가 사용됩니다.

[입출력 예]

| src | answer |
|---|---|
| "000" | "C" |
| "11111" | "1E" |
| "00011110" | "CDA" |
| "111100100011" | "1DBACB" |

문자열 압축 문제의 목표는 입력 받은 문자열의 앞문자가 0인지 1인지 체크하고,
앞에서부터 문자가 변화할 때마다 해당 문자 개수를 알파벳으로 변환하여 answer에 추가하는 것입니다.

# 핵심 풀이

먼저 C언어의 경우, answer에 최대 (입력 받은 문자열의 길이+1)만큼의 char형 크기만큼 동적할당을 해야 합니다. 그 이유는 입력 받은 문자열의 길이가 n이고 연속된 숫자가 없을 때, answer의 문자열 길이는 011…11 혹은 111…11로, 길이가 최대 (n+1)이기 때문입니다. 예를 들어 입력값이 101로, n=3이면 answer은 1AAA로 answer의 길이는 4입니다.

다음, 입력 받은 문자열의 앞문자가 0인지 1인지 if문으로 비교하여 1이면 answer에 '1'을 저장합니다. C언어의 경우, 문자열에 접근하기 위하여 index가 필요하므로 index를 하나 증가시킵니다.

입력 받은 문자열 앞에서부터 반복문과 if문을 이용해 문자 변화를 체크하여 문자로 저장합니다.
반복문이 i=0부터 (n-1)까지일 때,입력 받은 문자열의 i번째 문자와 (i+1)번째 문자 비교하여 같으면 문자가 연속하므로 count를 하나 증가시키고, 다르면 문자가 연속하지 않고 변화하므로 answer에 'A'+count 저장합니다.

* 아스키코드를 이용해 'A'에 숫자를 더하면 'A'~'Z'까지 저장 가능합니다.(ex) 'A'+2는 'C'입니다.)

C언어의 경우, 반복문을 완료하면 answer의 마지막에 널문자를 저장하여 문자열의 끝임을 입력합니다.

# C 소스코드

```c
char* solution(const char* s) {
    int n, count = 0, index = 0; // 연속하는 문자 개수, 인덱스
    n = strlen(s); // 입력 받은 문자열의 길이

 // 최대 (문자열의 길이 + 1)만큼 동적할당합니다.
    char* answer = (char*)malloc(sizeof(char) * (n + 1));

 // 앞문자가 1이면 answer에 '1'을 저장하고 index를 증가합니다.
    if (s[0] == '1')  answer[index++] = '1';

 // 문자 변화를 체크하여 연속된 문자 개수를 answer에 저장하고, index를 증가시킵니다.
    for (int i = 0; i < n  -  1; i++) {
        if (s[i] != s[i + 1]) {
            answer[index++] = 'A' + count;
            count = 0;
        }
        else  count++; // 문자가 연속하므로 하나 증가시킵니다.
    }
    answer[index++] = 'A' + count; // 마지막까지 저장합니다.

// 널문자 저장하여 answer 입력 종료합니다.
    answer[index] = '₩0';
    return answer;    }
```

# JAVA 소스코드

```java
import java.util.Scanner;

public class Solution{
    public static String solution(String s) {
    String answer="";
    int count=0; // 연속하는 문자 개수

    // 앞문자가 1이면 answer에 '1'을 저장하고 index를 증가합니다.
    if(s.charAt(0)=='1') answer+="1";

    // 문자 변화를 체크하여 연속된 문자 개수를 answer에 저장합니다.
    for(int i=0; i<s.length()-1; i++) {
      if(s.charAt(i) != s.charAt(i+1)) {
        answer += (char)('A'+count);
        count=0;    // 연속하는 문자 개수를 초기화합니다.
      }
      else count++; // 문자가 연속하므로 하나 증가시킵니다.
    }
    answer += (char)('A'+count); // 마지막까지 저장합니다.

    return answer;
    }
}
```

# 2019년도 제13회 C 부문 7번

7. (10점) 이화는 회문(palindrome: 앞으로 읽으나 뒤로 읽으나 동일한 것)을 너무 좋아한다. $n$개의 요소를 갖는 배열 $A$도 $A[i] = A[n-1-i]$인 회문을 만들고 싶어 한다.(배열 인덱스는 0부터 시작)

즉, {1, 3, 1}, {1, 2, 2, 1}과 같은 모양의 배열이다.

이화를 위해 회문이 아닌 배열은 다음 규칙에 따라 수정해서 회문을 만들 수 있게 했다. 수정 규칙은 $n$개의 요소를 갖는 배열이 있을 때, 인접한 요소끼리 합해서 $n-1$개의 요소를 갖는 배열로 만드는 방법이다.

예를 들어 {1, 3, 5, 7}인 배열은 인접한 두 요소 1, 3을 더해 {4, 5, 7}인 배열로 수정할 수 있다.

처음에 회문이 아닌 배열이 주어졌을 때, 몇 번의 수정을 통해 회문이 만들어지는지 그 횟수를 구하는 프로그램을 작성하시오.

[입력 형식]
- 첫 번째 줄에 배열을 구성하는 요소의 개수 $n$을 입력한다. $(1 \le n \le 10)$
- 두 번째 줄에 $n$개의 숫자 $X_i$를 공백으로 구분하여 입력한다. $(1 \le X_i \le 100)$

[출력 형식]
- 회문을 만들기까지 최소한의 배열 수정 횟수를 출력한다.

[입력 예1]
```
3
1 2 3
```
[출력 예1]
```
1
```

[입력 예2]
```
5
1 2 4 6 1
```
[출력 예2]
```
1
```

[입력 예3]
```
4
1 4 3 2
```
[출력 예3]
```
2
```

※ [예1] 1 2 3 → 3 3,
[예2] 1 2 4 6 1 → 1 6 6 1
[예3] 1 4 3 2 → 5 3 2 → 5 5

회문 문제의 목표는 배열이 대칭이므로 포인터 2개를 이용하여 하나는 앞, 다른 하나는 뒤에서부터 시작해 포인터가 가리키는 배열 원소를 비교하여 회문을 수정하는 것으로, 모든 경우의 수를 확인하는 완전 탐색을 이용합니다.

# 핵심 풀이

포인터 2개를 이용하는 투포인터(Two Pointer)는 1차원 배열에서 2개의 포인터로 원하는 결과를 얻는 방법입니다. 앞은 start, 뒤는 end로 int 매개변수를 선언하여 배열 인덱스에 접근합니다.

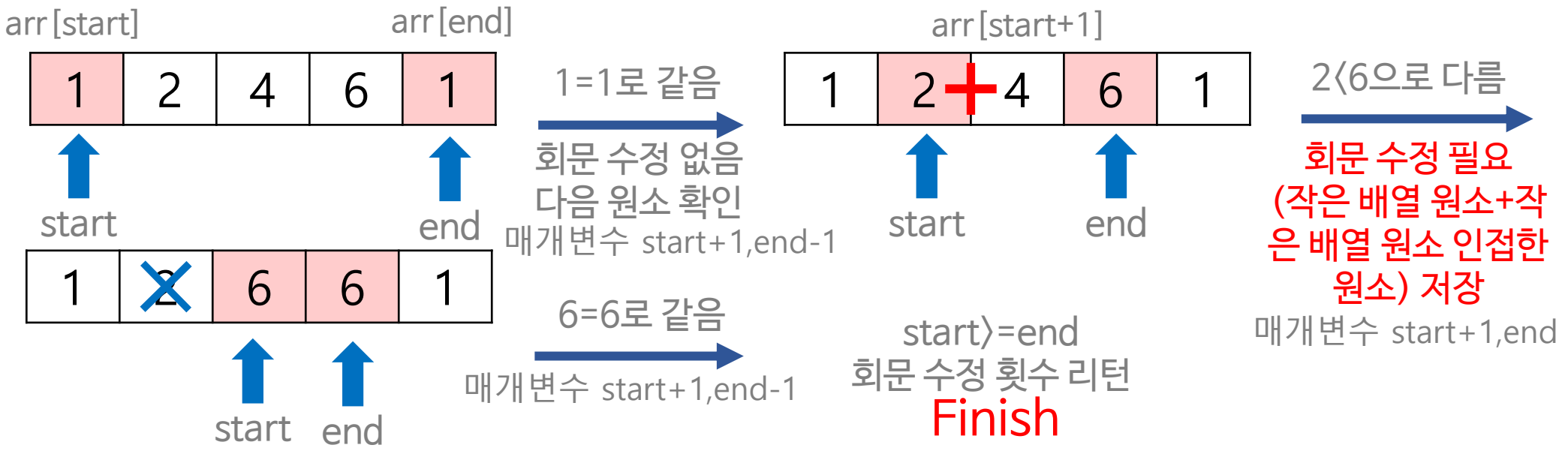
완전탐색(Brute Force)은 모든 경우의 수를 전부 탐색하는 방법으로, 모든 배열의 원소를 확인하기 위해 매개변수 start가 end보다 크거나 같기 전까지 배열 원소 비교 함수를 재귀함수로 호출합니다.

- start와 end가 가리키는 배열 원소가 같으면 대칭이므로 회문을 수정할 필요가 없어, 다음 원소를 확인하기 위해 start를 하나 증가, end를 하나 감소시켜 매개변수로 전달하고, 다음 원소 비교를 위해 다시 호출합니다.

- start와 end가 가리키는 배열 원소가 다르면 대칭이므로 회문을 수정해야 합니다. 대칭을 위해 작은 배열 원소와 작은 배열 원소의 인접한 원소를 합하여 저장하고, start나 end를 증가 혹은 감소시켜 매개변수로 전달합니다.
다음 원소 비교를 위해 다시 호출합니다.

예시2

```
5
1 2 4 6 1
```



# C 소스코드

```c
int cnt = 0;   // 회문 수정 횟수
int solution(int* arr, int start, int end) {
// 더 이상 수정할 필요 없으므로 횟수를 리턴합니다.
  if (start == end || start > end)
    return cnt;
// 대칭으로 회문 수정할 필요 없으므로 다음 원소들 확인하기 위해 함수를 다시 호출합니다.
  if (arr[start] == arr[end]) {
    return solution(arr, start + 1, end - 1);
  }
 // 대칭이 아니므로 회문 수정이 필요합니다. 작은 배열 원소와 작은 배열 원소 인접한 원소를 합하여 저장하고, 함수를 다시 호출합니다.
  else {
    if (arr[start] < arr[end]) {
      arr[start + 1] = arr[start] + arr[start + 1];
      cnt++;  // 회문 수정 횟수 증가
      return solution(arr, start + 1, end);
    }
    else {
      arr[end - 1] = arr[end] + arr[end - 1];
      cnt++;  // 회문 수정 횟수 증가
      return solution(arr, start, end - 1);
    }
  }
}
```

# JAVA 소스코드

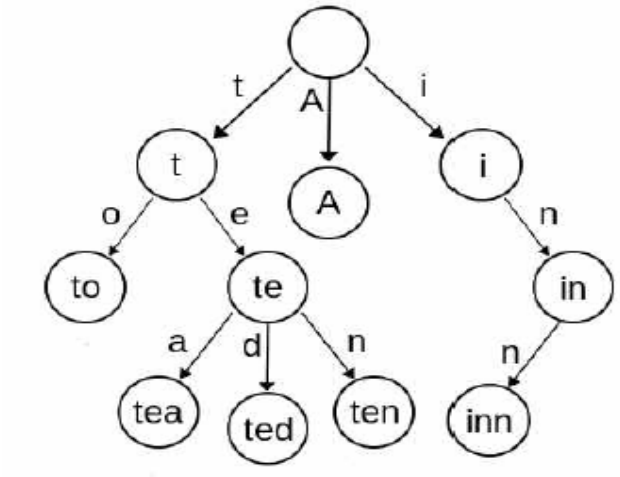
```java
import java.util.Scanner;

public class Solution {
  static int cnt;  // 회문 수정 횟수
  public static int solution(int[] arr, int start, int end) {
      if (start == end || start > end) // 회문 수정 횟수를 리턴
          return cnt;

      // 배열 원소 비교 (C 소스 코드 주석 참고)
      if (arr[start] == arr[end]) {
          return solution(arr, start + 1, end - 1);
      }
      else {
        if (arr[start] < arr[end]) {
            arr[start + 1] = arr[start] + arr[start + 1];
            cnt++;
            return solution(arr, start + 1, end);
        }
        else {
            arr[end - 1] = arr[end] + arr[end - 1];
            cnt++;
            return solution(arr, start, end - 1);
        }
      }
  }
}
```

# 2019년도 제13회 C 부문 10번

10. (10점) 트리는 노드(node)와 두 개의 노드를 연결하는 간선(edge)들로 구성된 사이클이 없는 그래프의 일종이다. 이러한 트리(tree) 구조를 활용해 접두어 트리를 구성할 수 있다. 접두어 트리는 다음과 같은 방법으로 특정 집합의 단어 접두어를 모두 나타내는 데이터 구조이다.

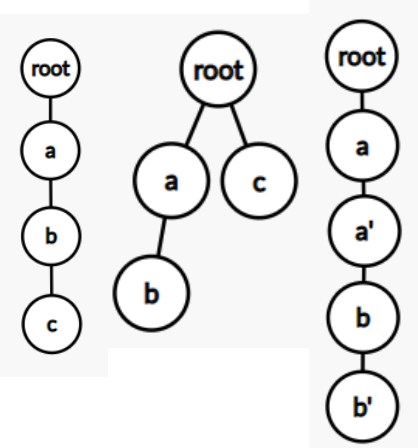
- 간선에 알파벳 문자를 표기된다.
- 트리의 루트 노드는 접두어가 비어 있다. (루트노드는 부모노드가 없는 출발점의 노드이다.)
- 루트노드를 제외한 각 노드는 접두어가 표기되는데, 트리의 루트에서 그 노드로 이어지는 간선에 표기된 문자를 순차적으로 연결해 만들어진다.
- 단일 노드에서 동일한 문자가 표기된 두 개의 간선은 존재하지 않는다.

영어 알파벳의 소문자로 구성된 $n$ 개의 단어(접두어)가 주어졌을 때, 각 단어의 글자는 임의의 방식으로 재배열할 수 있다. 즉 bac는 abc로 재배열한 후 이를 접두어로 한다. 이와 같이 재배열한 접두어들을 생성하는 노드의 개수가 최소인 접두어 트리의 노드 수를 구하는 프로그램을 작성하시오.




(위의 접두어 트리로 만들 수 있는 접두어들은 ‘t’, ‘A’, ‘i’, ‘to’, ‘te’, ‘in’, ‘tea’, ‘ted’, ‘ten’, ‘inn’은

예1 예2 예3




예3) aabb, abab, abba, bbaa, baba, baab 6가지 모두 가능합니다.

[입력 형식]
- 첫 번째 줄에 접두어의 개수 $n$을 입력한다.($1 \leq n \leq 16$)
- 다음 $n$개의 줄에 걸쳐 알파벳의 소문자로 구성된 단어를 입력한다.(각 단어의 길이는 1000보다 작다.)

[출력 형식]
- 접두어 트리가 가질 수 있는 최소 노드 개수를 출력한다.

[입력 예1]

```
3
a
ab
abc
```

[출력 예1]

```
4
```

[입력 예2]

```
3
a
ab
c
```

[출력 예2]

```
4
```

[입력 예3]

```
4
baab
abab
aabb
bbaa
```

[출력 예3]

```
5
```

접두어 트리 문제의 목표는 문자열을 재배열한 모든 경우의 접두어 트리 중 노드 개수(알파벳)의 최솟값 구하는 것입니다.

직관적으로 생각해보면 아래와 같이 문자열을 하나씩 확인하며 중복되는 알파벳을 제외한 알파벳 개수를 계속 더하고, 조합에 추가하여 조합 안의 문자열 개수가 입력한 문자열 개수와 같을 때 알파벳 개수에 root 1개 더한 값을 구한 값이 정답일 것이라고 추측됩니다.

하지만 이는 문자열 입력한 순서대로 조합을 확인하여 구한 값이 최솟값이 아닐 수 있으므로 모든 조합을 확인할 필요가 있고, 문자열을 하나씩 확인하는 것보다 2가지 조합으로 나누어 조합별로 확인하는 것이 훨씬 효율적입니다. 시간 단축을 위하여 이미 계산한 결과는 부분 문제 반복을 통해 이전 결과를 활용해야 합니다.

반례

```
3
ab
bc
cd
```

① a 1개, b 1개

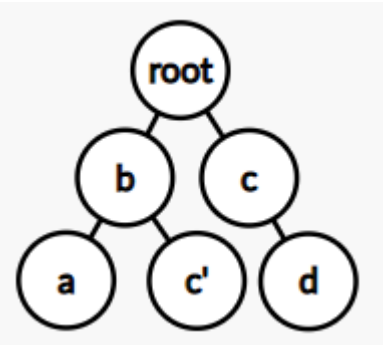
② b 1개, c 1개

③ c 1개, d 1개

①+② 문자열 조합 확인
a 1개, b 1개, c 1개 =〉 3개

①+②+③ 문자열 조합 확인
a 1개, b 1개, c 1개, d 1개 ∴ 4개
(root 포함 총 5개)

정답 中 하나




a 1개, b 1개, c 2개, d 1개
∴ 5개 =〉 root 포함 총 6개

# 핵심 풀이

입력 받은 문자열들을 2가지 조합으로 나누어 모든 조합을 확인하는 solve 함수를 작성합니다.

(나눈 2가지 조합의 알파벳 개수의 합 - 나누기 전 문자열 모두에 중복되는 알파벳)을 계산하기 위하여 문자열들에 등장한 알파벳의 최소 개수(중복하는 알파벳)를 계산하는 calc_pref 함수를 작성합니다.

한 번 계산한 조합은 다시 계산하지 않고 이전 결과를 활용하기 위하여 부분 문제로 나누는 동적 계획법을 사용합니다. 결과들을 저장하는 dp배열을 선언합니다.

나눈 문자열 조합의 문자열 개수가 1이 될 때까지 부분 문제를 반복하고, 문자열 개수가 1이면 해당 문자열의 알파벳 개수 curr를 반환합니다.

이러한 과정으로 모든 조합을 확인하면 최솟값을 선택하여 에 배열에 저장합니다.

입력 받은 문자열들을 2가지 조합으로 나누어 모든 조합을 확인하는 solve 함수와 (나눈 2가지 조합의 알파벳 개수의 합 - 나누기 전 문자열 모두에 중복되는 알파벳 pref) 을 계산하기 위하여 문자열들에 등장한 알파벳의 최소 개수를 계산하는 calc_pref 함수를 작성합니다 한 번 계산한 조합은 다시 계산하지 않고 이전 결과를 활용하기 위하여 부분 문제로 나누는 동적계획법을 사용합니다. 나눈 문자열 조합의 문자열 개수가 1이 될 때까지 부분 문제를 반복하고, 문자열 개수가 1이면 해당 문자열의 알파벳 개수 curr를 반환합니다. 모든 조합을 확인하면 curr 최솟값을 dp배열에 저장합니다. (그림 참고)

앞쪽 예시

| 문자열 3개 입력 |
|---|
| {ab, bc, cd} |

①

solve 함수 실행하여 문자열 3개를 2가지 조합으로 나눕니다.

| 문자열 3개 | |
|---|---|
| 1 | {bc, cd}, {ab} |
| 2 | {ab, cd}, {bc} |
| 3 | {cd}, {ab, bc} |
| 4 | {ab}, {bc, cd} |
| 5 | {bc}, {ab, cd} |
| 6 | {ab, bc}, {cd} |

**문자열 3개 모두에 중복되는 알파벳이 없음**

* 4, 5, 6번은 1, 2, 3번 결과와 같습니다.
* 입력 받은 문자열이 3개일 때, 만들 수 있는 문자열 조합의 개수는 2^3개로, 문자열을 모두 포함하는 경우 1개, 문자열을 모두 포함하지 않는 경우 1개를 제외하면 총 6개의 조합이 존재합니다.

②

| 문자열 2개 | |
|---|---|
| 1 | {bc, cd} |
| 2 | {ab, cd} |
| 3 | {ab, bc} |

③

solve 함수 실행하여 문자열 2개를 2가지 조합으로 나눕니다.

| 문자열 2개 | |
|---|---|
| 1 | {bc}, {cd} |
| 2 | {ab}, {cd} |
| 3 | {ab}, {bc} |

**2개 문자열에 등장한 알파벳 최소 개수는 b 0개, c 1개, d 0개로 c 1개 중복**

**b 1개 중복**

④

| 문자열 1개 | |
|---|---|
| 1 | {ab} |
| 2 | {bc} |
| 3 | {cd} |

**중복 없음**

⑤

| 문자열 1개 | |
|---|---|
| 1 | {ab} |
| 2 | {bc} |
| 3 | {cd} |

이전 결과 ④와 같습니다.

solve함수에서 해당 조합 내 가능한 문자열 조합을 모두 확인하여 조합을 합칠 때 구한 curr 중 최솟값을 dp배열에 저장합니다.

calc_pref 함수에서 ③의 중복 개수를 계산하고 문자열 조합들의 알파벳 개수 합에서 뺍니다.

| 문자열 3개 | dp |
|---|---|
| {ab, bc, cd} | 5 |

최솟값이 5이므로 dp배열 ⑨에 5 저장

| | 문자열 3개 | 개수(curr) |
|---|---|---|
| 1 | {bc, cd}, {ab} | 5(=3+2) |
| 2 | {ab, cd}, {bc} | 6(=4+2) |
| 3 | {cd}, {ab, bc} | 5(=3+2) |
| 4 | {ab}, {bc, cd} | 5 |
| 5 | {bc}, {ab, cd} | 6 |
| 6 | {ab, bc}, {cd} | 5 |

⑧

| | 문자열 2개 | dp |
|---|---|---|
| 1 | {bc, cd} | 3 |
| 2 | {ab, cd} | 4 |
| 3 | {ab, bc} | 3 |

⑦

| | 문자열 2개 | 개수 |
|---|---|---|
| 1 | {bc}, {cd} | 3(=2+2-1) |
| 2 | {ab}, {cd} | 4(=2+2) |
| 3 | {ab}, {bc} | 3(=2+2-1) |

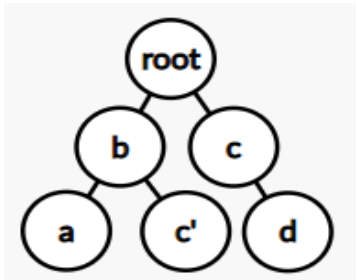
dp배열에 curr 그대로 저장

⑥

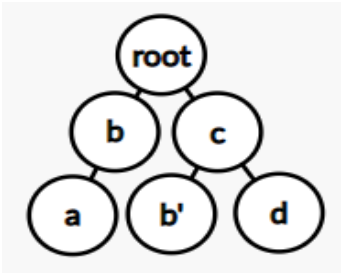
| | 문자열 1개 | 개수 |
|---|---|---|
| 1 | {ab} | 2 |
| 2 | {bc} | 2 |
| 3 | {cd} | 2 |

| | 문자열 1개 | 개수 |
|---|---|---|
| 1 | {ab} | 2 |
| 2 | {bc} | 2 |
| 3 | {cd} | 2 |

정답




or




**∴ 최소 5개 ⟩ root 포함 총 6개**

solve 함수에서 문자열 조합 안의 문자열이 1개이므로 알파벳 개수를 반환합니다.(calc_pref 함수를 이용하면 문자열들에 나타난 알파벳의 최소 개수를 계산하므로 문자열 1개의 알파벳 개수 계산도 가능합니다.)

## * 동적계획법(Dynamic Programming)

표의 회색 부분은 이미 계산한 결과로 다시 계산하지 않도록 동적계획법을 이용해 dp배열에 결과값을 저장하면 효율적입니다.
동적계획법이란, 주어진 문제를 여러 개의 부분 문제로 나눈 후(부분 문제 반복) 각 부분 문제의 결과를 이용하여 원래의 문제를 해결하는 방법(최적 부분 구조)입니다.
동적계획법은 부분 문제를 해결하기 위한 dp배열을 정의하여야 합니다. dp[x]를 x조합일 때 중복 알파벳을 제외한 알파벳 개수의 최솟값으로 정의합니다. 이때, 문자열 포함 여부를 x에 저장하기 위하여 비트마스크를 이용하면 수월하게 저장할 수 있습니다.

## * 비트마스크(BitMask)

비트마스크란 정수를 이진수 표현으로 저장하는 기법입니다. 정수를 이용해 이진수 자릿수가 1이면 조합 안의 문자열 포함, 0이면 미포함을 나타냅니다. 예를 들어 dp[x]에 x가 6일 때, 6을 이진수로 나타내면 110이므로 세번째, 두번째 문자열은 조합에 포함되지만, 첫번째 문자열은 조합에 포함되지 않습니다.

ex) 입력한 문자열이 ab, bc, cd일 경우, dp[7]의 7을 이진수로 표현하면 111이므로, 조합 안에 3번째+2번째+1번째 문자열을 모두 포함한다는 것을 의미합니다. 111 조합을 2가지 조합으로 나누려면 {110과 001}, {101과 010}, … {001과 110}으로 나눌 수 있고, 중복된 문자는 없습니다. 문자열 1개를 포함한 조합인 100, 010, 001은 해당 문자열의 알파벳 개수를 dp에 저장하므로 dp[4]=2, dp[2]=2, dp[1]=2입니다. 문자열 2개 이상 포함한 조합인 dp[6], dp[5], dp[3]은 다시 2가지 조합으로 나누어 중복 알파벳 개수를 제외하는 과정을 반복합니다.
(그림 참고)

| 문자열 3개 입력 |
| --- |
| {ab, bc, cd} |

조합이 문자열 3개를 모두 포함하는 경우 최솟값 이 5 였 으 므 로 dp[7]=5

| x(이진수) | 문자열 3개 |
| --- | --- |
| 111 | {ab, bc, cd} |

X가 7일 때 이진수로는 111로, 모든 문자열 3개를 포함한 조합을 의미합니다.

| x | 문자열 3개 |
| --- | --- |
| 110, 001 | {bc, cd}, {ab} |
| 101, 010 | {ab, cd}, {bc} |
| 100, 011 | {cd}, {ab, bc} |
| 011,100 | {ab}, {bc, cd} |
| 010, 101 | {bc}, {ab, cd} |
| 001, 110 | {ab, bc}, {cd} |

| x | 문자열 2개 |
| --- | --- |
| 100, 010 | {bc}, {cd} |
| 001, 100 | {ab}, {cd} |
| 001, 010 | {ab}, {bc} |

| x | 문자열 1개 |
| --- | --- |
| 001 | {ab} |
| 010 | {bc} |
| 100 | {cd} |

## Before) 해당 문제에서 필요한 비트 연산

1. **1 << i** : 왼쪽으로 i만큼 1비트 이동합니다. (i번째 자릿수가 1인 이진수로, 정수로는 2의 i승을 의미합니다.)
2. **x&(1 << i)** : x 이진수의 i번째 자릿수가 1인지 확인합니다. (여기서는 x라는 조합에 i번째 문자열 포함 여부를 확인합니다.)
3. **x&-x** : x 이진수에서 가장 오른쪽에 있는 1이 몇 번째 자릿수인지 찾습니다.
   (x&-x와 x가 같으면 x는 1비트가 1개 존재하므로 문자열이 1개만 존재하는 조합입니다)
4. **for(i = (x-1)&x; i > 0; i = (i-1)&x** : 부분 집합 순회(모든 문자열 조합 순회)
   (ex) x=111이면 i=110, 101, 100, 011, 010 ,001 순회)
5. **x^i** : XOR연산자, x 이진수와 i를 비트끼리 비교하여 같으면 0, 다르면 1비트입니다.
   (x=110이면 x^i=001로 111을 2가지 조합으로 나눌 수 있습니다.

# C 소스코드

memset함수를 이용하여 dp배열을 -1로 초기화합니다. 아직 dp배열 계산을 하지 않았음을 의미합니다.

모든 문자열의 알파벳 개수를 인덱스에 맞추어 이중배열 cnt에 미리 저장합니다. 나중에 중복 알파벳을 비교할 때 사용합니다.

입력 받은 문자열들을 2가지 조합으로 나누는 solve 함수의 리턴값에 root노드 1개 더한 값이 최종 결과값입니다. solve 함수 실행을 위해 입력받은 문자열의 개수 n과 입력 받은 문자열들을 모두 포함하는 경우(1<<(n-1)를 정수로 전달합니다.

ex) 입력 받은 문자열 개수가 3이면 1<<3 -1 = 2^3-1 = 7이므로 x=7, 즉 문자열 3개가 모두 조합에 포함될 때 solve함수에 7, 이진수로는 111을 전달합니다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXN 16          // 최대 문자열 개수
#define MAXL 1001        // 최대 문자열 길이
#define INF 999999999 // INFINITY로 어떤 수와 비교해도 가장 큰 수

int cnt[MAXN][26];       // 문자열의 a~z 26개의 알파벳 개수
int dp[1 << MAXN];       // 최대 문자열 조합 경우의 수 2^16만큼 필요

int solution(int n, char str[][1001]) // n 문자열 개수
{
   // 아직 dp배열 계산하기 전이므로 dp배열 -1로 초기화합니다.
    memset(dp, -1, sizeof(dp));

    // i번째 문자열의 알파벳 개수를 인덱스에 맞추어 모두 저장합니다.
   for (int i = 0; i < n; i++) {
      for (int j = 0; str[i][j]; j++) {
         cnt[i][str[i][j] - 'a']++;
      }
   }
   return solve(n, (1 << n) - 1) + 1;   } // solve(2^n-1) 실행 후 +1
```

매개변수 x 조합을 2가지 조합으로 나누는 solve 함수 작성합니다.

한 번 계산한 dp는 다시 계산하지 않도록 -1 값이 dp[x]에 저장되어 있지 않으면 이미 계산한 결과로 그대로 리턴합니다.

calc_pref 함수를 실행하여 문자열 조합 x의 문자열 모두에 중복되는 알파벳 개수를 계산합니다.

ex) x=111(이진수)이면, 1번째, 2번째, 3번째 문자열 모두에 포함되는 알파벳 개수 연산

비트 연산을 이용하여 조합 x에 포함된 문자열 개수가 1개이면 알파벳 개수를 리턴합니다.

ex) x=100(이진수)이면, 3번째 문자열만 포함
=〉 3번째 문자열의 문자 개수 반환

```
// 문자열 2가지 조합으로 나눕니다.
int solve(int n, int x) {
    // 한 번 계산한 dp는 다시 계산하지 않고 이전 계산 결과 활용
    if (dp[x] != -1) return dp[x];

    // 문자열 조합 x에서 모두 중복되는 알파벳 개수를 계산합니다.
    int pref = calc_pref(n, x);

    // x가 2의 제곱수인지 확인
    // =〉 문자열 1개만 포함한 조합인 경우,
    //    해당 문자열의 알파벳 개수 리턴
    if ((x & -x) == x) return dp[x] = pref;

    // 모든 문자열 조합 모두 순회
    dp[x] = INF;
    for (int i = (x - 1) & x; i > 0; i = (i - 1) & x) {
        int curr = solve(n, i) + solve(n, x ^ i) - pref;
        dp[x] = (dp[x] > curr) ? curr : dp[x];
    }

    return dp[x];
}
```

문자열 조합 x를 2가지 조합으로 나누는 solve함수를 문자열 조합의 문자열 개수가 1개가 되어 calc_pref 함수가 실행될 때까지 호출합니다.

ex) n=3, x=7로 이진수로는 111이면
i가 110부터 001까지 순회하고,
x^i는 XOR관계로, 001부터 110까지 순회합니다.
조합의 문자열이 1개일 때까지 재귀 함수를 호출합니다.

| i | x^i | 문자열 조합 x의 문자열 | curr |
|---|---|---|---|
| 110, | 001 | {bc, cd}, {ab} | 5 |
| 101, | 010 | {ab, cd}, {bc} | 6 |
| 100, | 011 | {cd}, {ab, bc} | 5 |
| 011, | 100 | {ab}, {bc, cd} | 5 |
| 010, | 101 | {bc}, {ab, cd} | 6 |
| 001, | 110 | {ab, bc}, {cd} | 5 |

2가지 조합의 알파벳 개수 합에서 중복된 알파벳 개수 pref를 뺀 값이 curr로, 여태껏 계산한 curr 중 최솟값을 선택하여 dp[x]에 저장합니다.

ex) n=3, x=7=111(이진수), pref = 0(ab, bc, cd 모두에 중복된 알파벳 없음), dp[7]=5(curr 중 가장 작은 값이 해당 조합의 최솟값)

```
// 문자열 2가지 조합으로 나눕니다.
int solve(int n, int x) {
    // 한 번 계산한 dp는 다시 계산하지 않고 이전 계산 결과 활용
    if (dp[x] != -1) return dp[x];

    // 문자열 조합 x에서 모두 중복되는 알파벳 개수를 계산합니다.
    int pref = calc_pref(n, x);

    // x가 2의 제곱수인지 확인
    // => 문자열 1개만 포함한 조합인 경우,
    //    해당 문자열의 알파벳 개수 리턴
    if ((x & -x) == x) return dp[x] = pref;

    // 모든 문자열 조합 모두 순회
    dp[x] = INF;
    for (int i = (x - 1) & x; i > 0; i = (i - 1) & x) {
        int curr = solve(n, i) + solve(n, x ^ i) - pref;
        dp[x] = (dp[x] > curr) ? curr : dp[x];
    }

    return dp[x];
}
```

문자열 조합 x에서 중복 알파벳 개수를 계산하는 calc_pref 함수 작성합니다.

중복 알파벳 개수를 저장할 임시 배열 temp를 INF로 최댓값 초기화합니다. 알파벳별로 문자열마다 해당 알파벳이 존재하는 최소 개수를 저장합니다.

반복문을 이용해 i번째 문자열이 포함되는지 확인하고, i번째 문자열을 포함하면 temp와 cnt[i] 비교하여 작은 값이 중복되는 알파벳 개수이므로 temp 배열에 알파벳 인덱스에 맞추어 저장합니다.

ex) ab, bc, cd 문자열 모두 포함하는 경우 => 전부 0
ab 문자열 확인 => temp[0]=1, temp[1]=1 …
bc 문자열 확인 => temp[0]=0, temp[1]=1 …
cd 문자열 확인 => temp[0]=0, temo[1]=0 …

len은 temp 배열 전부 합한 값이 문자열 조합 x에서 중복되는 알파벳 개수입니다.

```
// 문자열 조합 x에서 중복 알파벳 개수 계산합니다.
int calc_pref(int n, int x) {
   int len = 0;        // 중복 알파벳 개수
   int temp[26];  // 알파벳별로 최소로 존재하는 개수 저장

   // temp배열을 최댓값으로 초기화합니다.
   for (int i = 0; i < 26; i++) {
      temp[i] = INF;
   }

   for (int i= 0; i < n; i++) {
      if (x & (1 << i))                    // i번째 문자열 포함 확인합니다.
         for (int j = 0; j < 26; j++)
            // temp[j]와 cnt[i][j]중 작은 값 선택합니다.
            temp[j] = (temp[j] > cnt[i][j]) ? cnt[i][j] : temp[j];
   }

   // temp 전부 더합니다.
   for (int i = 0; i < 26; i++)
      len += temp[i];

   return len;
}
```

# JAVA 소스코드

```java
import java.util.*;

public class Solution {
    static int MAXN = 16;          // 최대 문자열 개수
    static int MAXL = 1001;        // 최대 문자열 길이
    static int INF = 999999999;    // 최댓값

    static int cnt[][] = new int[MAXN][26]; // 알파벳 개수
    static int dp[] = new int[1<<MAXN]; // 최대 2^16 필요

    public static int solution(int n, char[][] str)
    {
        Arrays.fill(dp, -1);       // dp -1로 초기화

        for(int i=0; i<n; i++)
        {
            // 알파벳 개수 저장
            for(int j=0; j<str[i][j]; j++)
            {
                cnt[i][str[i][j]-'a']++;
            }
        }

        return solve(n, (1<<n)-1)+1; // solve(2^n-1)실행 +1
}
```

```java
// 문자열 2가지 조합으로 나눔
public static int solve(int n, int x)
{
    // 이전 결과 활용
    if (dp[x] != -1) return dp[x];

    // 문자열 조합 x에서 모두 중복되는 알파벳 개수 연산
    int pref = calc_pref(n, x);

    // x가 2의 제곱수인지 확인
    // =>문자열 1개만 포함한 조합
    //   해당 문자열 개수 반환
    if ((x&-x) == x) return dp[x] = pref;

    // 모든 문자열 조합 모두 순회
    dp[x] = INF;
    for (int i = (x - 1) & x; i > 0; i = (i - 1) & x) {
        int curr = solve(n, i) + solve(n, x^i) - pref;
        dp[x] = (dp[x] > curr) ? curr : dp[x];
    }

    return dp[x];
}
```

```java
// 중복 알파벳 개수 계산
public static int calc_pref(int n, int x)
{
  int len = 0; // 중복 알파벳 개수
  int temp[] = new int[26]; // 알파벳별로 최소로 존재하는 개수 저장

  // 최댓값으로 초기화
  for (int i = 0; i < 26; i++) {
    temp[i] = INF;
  }

  for (int i = 0; i < n; i++) {
    if ((x & (1 << i)) != 0) // 문자열 포함 확인
      for (int j = 0; j < 26; j++)
        // temp[j]와 cnt[i][j] 중 작은 값 선택
        temp[j] = (temp[j] > cnt[i][j]) ? cnt[i][j] : temp[j];
    }

  // temp 전부 합함
  for (int i = 0; i < 26; i++)
     len += temp[i];

  return len;
}
```