ICPC Sinchon

04. Bruteforcing, Recursive function

04. 완전 탐색, 재귀 함수

2023 Summer Algorithm Camp

목차

완전 탐색

- 가능한 모든 경우의 수를 탐색하며 답을 찾는 방법

- 가장 단순하지만, 시간이 상당히 오래 걸리는 편이다.
- 보통 N이 작을 때 사용한다.
  - 시간복잡도를 유도할 수 있어야 한다.
  - 걸리는 연산량을 계산했을 때, 보통 1억을 1초로 잡음

- 탐색하는 방법
  - 반복문
  - 재귀 함수 (백트래킹)
  - 순열 순회
  - 비트마스킹
  - 등..

- 반복문 : 변수의 개수가 고정일 때, 반복문으로 모든 경우의 수에 대해 답을 구해볼 수 있다.

예시 문제

- BOJ 6131 (완전 제곱수)

- N이 주어질 때, 다음 식을 만족하는 쌍 (A, B)의 개수를 구하는 문제 (단, $1 \leq B \leq A \leq 500$, $1 \leq N \leq 1{,}000$)
  - $A^2 = B^2 + N$

- 모든 쌍을 순회하면서 위 식을 만족하는지 확인하면 된다.
- A의 범위가 500이고, B의 범위도 500이므로 가능한 쌍의 개수는 250,000보다 작다.
  - 1억보다 충분히 작으므로 시간 안에 돌아간다.

240

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n; cin >> n;
    int ans = 0;
    for (int i = 1; i <= 500; i++) {
        for (int j = i; j <= 500; j++) {
            if (i * i + n == j * j) ans++;
        }
    }
    cout << ans;
}
```

질문?

순열 순회하기

- 원소의 개수가 N인 수열의 순열을 순회하는 방법
  - 수열의 순열 개수는 N!개

- N = 3
  - 1 2 3
  - 1 3 2
  - 2 1 3
  - 2 3 1
  - 3 1 2
  - 3 2 1

2023 Summer Algorithm Camp

순열 순회하기 STL

```
#include <algorithm>


std::next_permutation(first, last);
```

- 사전 순으로 바로 다음 순열로 변경하는 함수
  - 현재 순열이 사전 순으로 가장 마지막 순열이면 false를 반환하고, 사전 순으로 가장 앞서는 순열로 변경한다.
  - 그렇지 않으면 true를 반환하고, 사전 순으로 바로 다음 순열로 변경한다.

- `vector<int> vec = {1, 2, 3, 4, 5};`
- `next_permutation(vec.begin(), vec.end());`
  - vec은 {1, 2, 3, 5, 4}로 변경된다.

- `vector<int> vec = {1, 1, 2, 2, 2};`
- `next_permutation(vec.begin(), vec.end());`
  - vec은 {1, 2, 1, 2, 2}로 변경된다.

2023 Summer Algorithm Camp

순열 순회하기 STL

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3};
    do {
        for (int i: vec) cout << i << " ";
        cout << "\n";
    } while (next_permutation(vec.begin(), vec.end()));
}
```

```
Output:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

2023 Summer Algorithm Camp

순열 순회하기 STL

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    vector<int> vec = {2, 1, 3};
    do {
        for (int i: vec) cout << i << " ";
        cout << "\n";
    } while (next_permutation(vec.begin(), vec.end()));
}
```

Output:

```
2 1 3
2 3 1
3 1 2
3 2 1
```

모든 순열을 순회하려면 처음에 배열을 정렬해야 한다.

질문?

예시 문제

- BOJ 10819 (차이를 최대로)

- 배열 A_0, A_1, ⋯, A_{N-1}이 주어지면, 다음 식의 최댓값을 구하는 문제
  - $|A_0 - A_1| + |A_1 - A_2| + \cdots + |A_{N-2} - A_{N-1}|$

- $3 \le N \le 8$이므로 O(N! * N) 풀이가 가능하다. 즉, 나올 수 있는 모든 수열에 대하여 위 식의 값을 계산해보면 된다.
  - 8! = 40320

- 배열 A를 정렬하고 next_permutation을 사용해서 모든 경우의 수에 대해 값을 계산해보면 된다.

- 시간 복잡도 계산
  - 경우의 수 : N!
  - 각 경우에 대해서 식의 값을 계산하는 데 걸리는 시간 : N
  - O(N! * N)

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> vec(n);
    for (int &i: vec) cin >> i;
    sort(vec.begin(), vec.end());

    int ans = 0;
    do {
        int curr = 0;
        for (int i = 1; i < n; i++) curr += abs(vec[i] - vec[i - 1]);
        ans = max(ans, curr);
    } while (next_permutation(vec.begin(), vec.end()));

    cout << ans;
}
```

질문?

질문?

재귀 함수

- 자기 자신을 다시 호출하는 함수
- 대표적인 예) 귀납(재귀)적으로 정의하는 팩토리얼 함수나 피보나치 함수

- 팩토리얼
  - F(1) = 1
  - F(n) = n * F(n - 1), n > 1

- 피보나치 함수
  - F(1) = F(2) = 1
  - F(n) = F(n - 1) + F(n - 2), n > 2

재귀 함수

- 주로 사용하는 곳:
  - 반복문으로 하기 어려운 작업 (길이가 N인 순열 만들기 등)
  - 귀납적으로 정의된 함수(점화식) 계산 (피보나치 등)
  - 완전 탐색

- 익숙해지기 어렵지만, 구현을 간단하게 할 수 있다는 장점이 있다.

재귀 함수

- 재귀 함수 정의를 잘해야 한다.
  - 귀납적으로 사고할 것
  - 재귀 함수는 다른 매개변수에 대해 같은 행동을 하는 함수라고 생각하자.

- 해야 하는 것
  - base case(재귀 함수의 종료 조건)를 먼저 정한다.
  - 그렇지 않을 경우(general case), 계속해서 재귀 함수를 호출한다.
  - 호출한 재귀 함수의 반환 값을 받았을 때, 어떻게 처리할지 정한다.

- 재귀 함수는 항상 올바른 결과를 반환한다고 믿고 구현한다.

2023 Summer Algorithm Camp

재귀 함수 - 팩토리얼 함수

```cpp
#include <bits/stdc++.h>

using namespace std;

int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    cout << factorial(5); // 120
}
```

재귀 함수 - 팩토리얼 함수

- factorial(n) = n * factorial(n - 1)

- “factorial(n - 1)”의 반환 값이 항상 올바르다고 믿고 factorial(n)을 구현한다.
- 즉, factorial(n)을 계산할 때, factorial(n - 1), factorial(n - 2), …, factorial(1)을 계속 따라가면서 고민하지 말자.

- 이렇게 하려면 재귀 함수의 정의를 잘 만들어야 한다.
- “귀납적으로” 설계하자.

재귀 함수

- 반복문을 사용하지 않고 배열의 합을 구하는 재귀 함수를 설계해 보자.

- 먼저, 재귀 함수를 정의해 보자.
  - f(l, r) = 배열 [l, r]의 합

- base case)
  - l == r이면 A[l]을 반환한다.

- general case)
  - mid = ⌊ (l + r) / 2⌋
  - f(l, r) = f(l, mid) + f(mid + 1, r)
  - [l, r]의 합은 [l, mid]의 합 + [mid + 1, r]의 합과 같다.

2023 Summer Algorithm Camp

재귀 함수

- **f(l, r) = f(l, mid) + f(mid + 1, r)**

- 재귀 함수의 호출 과정
  - f(1, 4)
  - f(1, 2) + f(3, 4)
  - f(1, 1) + f(2, 2) + f(3, 3) + f(4, 4)
- 을 분석하는 것은 매우 비효율적이다.

- f(l, r)의 함수가 항상 올바른 결과만을 반환한다고 “**믿는다면**”
- 위 점화식은 단순히 배열을 절반으로 쪼개서 더하는 과정이기 때문에 당연히 성립할 것이다.

재귀 함수

- 주의해야 할 것

- 무한 루프
  - 종료 조건(base case)이 필요함.
  - 재귀 함수는 점점 종료 조건에 가까워지는 방향으로 호출되어야 함.

- 반복문과 비교했을 때의 단점
  - 팩토리얼 등의 간단한 점화식은 반복문으로 구현하는 것이 더 간단할 수도 있다.
  - 일반적으로 재귀 함수는 반복문에 비해 성능이 떨어진다.
    - 메모리나 시간 측면에서 반복문보다 성능이 떨어질 수 있다. (관련한 내용은 스택 메모리 참고)
    - 컴파일러에서 최적화를 해주는 경우도 있다.

질문?

예시 문제

- BOJ 15649 (N과 M (1))

- 1부터 N까지 자연수 중 중복 없이 M개를 고른 수열을 모두 구하는 문제 (사전 순 출력)

- N = 3, M = 1
  - 1
  - 2
  - 3

- N = 4, M = 2
  - 1 2
  - 1 3
  - 1 4
  - 2 1
  - 2 3
  - …

예시 문제

- solve(choose) : 현재까지 choose개의 수를 골랐을 때, 수열의 맨 뒤에 수를 하나 추가하는 함수
  - arr[0]부터 arr[choose - 1]까지는 이미 수가 채워진 상태이다.

  - choose == M이면 M개의 수를 모두 채운 것이다.
    - 현재 배열을 출력하고 함수를 종료한다.

  - arr[choose]에 1부터 N까지의 수를 하나 추가하면 된다.
    - 수를 추가할 때는 arr[0]부터 arr[choose - 1]까지 쓰지 않은 수인지만 확인하면 된다.
    - 확인은 0부터 choose - 1까지 하나씩 순회하면서 확인하면 된다.
  - arr[choose]에 값을 추가하고 나서 solve(choose + 1)을 호출하면 된다.

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, m;
vector<int> arr(10);

void solve(int choose) {
    if (choose == m) {
        for (int i = 0; i < m; i++) cout << arr[i] << " ";
        cout << "\n";
        return;
    }

    for (int i = 1; i <= n; i++) {
        int duplicated = 0;
        for (int j = 0; j < choose; j++) if (arr[j] == i) duplicated = 1;
        if (!duplicated) {
            arr[choose] = i;
            solve(choose + 1);
        }
    }
}

int main() {
    cin >> n >> m;
    solve(0);
}
```

240

질문?

240

예시 문제

- solve 함수에서 수를 사용했는지 사용하지 않았는지 확인하는 부분
- 현재는 arr[0] .. arr[choose - 1]까지 확인하기 때문에 O(M)이 걸린다.
- 최적화할 수 없을까?

- use 배열을 만들자.
  - use[i] = 1 : i를 사용했을 때
  - use[i] = 0 : i를 사용하지 않았을 때

- O(1)에 수를 추가할 수 있는지, 없는지 확인할 수 있다.

- arr[choose] = i로 수를 추가하면, use[i] = 1을 해준다.
- solve(choose + 1)을 호출한 뒤에 use[i] = 0으로 꼭 바꿔줘야 한다.

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, m;
vector<int> arr(10), use(10);

void solve(int choose) {
    if (choose == m) {
        for (int i = 0; i < m; i++) cout << arr[i] << " ";
        cout << "\n";
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (!use[i]) {
            use[i] = 1;
            arr[choose] = i;
            solve(choose + 1);
            use[i] = 0;
        }
    }
}

int main() {
    cin >> n >> m;
    solve(0);
}
```

240

질문?

예시 문제

- BOJ 15650 (N과 M (2))

- 1부터 N까지 자연수 중 중복 없이 M개를 고르되,
- 오름차순인 수열을 모두 구하는 문제 (사전 순 출력)

- N = 3, M = 1
  - 1
  - 2
  - 3

- N = 4, M = 2
  - 1 2
  - 1 3
  - 1 4
  - 2 3
  - 2 4
  - 3 4

예시 문제

- 오름차순인 배열을 만들면 되기 때문에, solve 함수에서는 맨 뒤 원소보다 큰 수들만 넣으면 됨.
- 따라서, use 배열을 관리할 필요가 없음.

- 구현의 편의를 위해 배열의 가장 앞에는 0을 채워 둠

240

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, m;
vector<int> arr = {0};

void solve(int choose) {
    if (choose == m) {
        for (int i = 1; i <= m; i++) cout << arr[i] << " ";
        cout << "\n";
        return;
    }

    for (int i = arr.back() + 1; i <= n; i++) {
        arr.push_back(i);
        solve(choose + 1);
        arr.pop_back();
    }
}

int main() {
    cin >> n >> m;
    solve(0);
}
```

질문?

2023 Summer Algorithm Camp

백트래킹

- 쉽게 표현하면 **가지치기**

- 해를 구성해 나가다가 “더 진행해도 답을 찾는 것이 **불가능**하다면”
- 현재 경우에서 해를 찾는 것을 더 진행하지 않는 것이 백트래킹이다.

- 앞서 재귀 함수에서
  - 현재까지 구성한 해로 답을 만들 수 없다면, 다음 재귀 함수를 호출하지 않고 현재 함수를 종료한다.
  - “답이 될 수 있는 경우들만 보면서” 다음 재귀 함수를 호출한다.
- 했던 행동들도 백트래킹이라고 볼 수 있다.

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, m;
vector<int> arr(10), use(10);

void solve(int choose) {
    if (choose == m) {
        for (int i = 0; i < m; i++) cout << arr[i] << " ";
        cout << "\n";
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (!use[i]) {
            use[i] = 1;
            arr[choose] = i;
            solve(choose + 1);
            use[i] = 0;
        }
    }
}

int main() {
    cin >> n >> m;
    solve(0);
}
```

240

예시 문제

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, m;
vector<int> arr = {0};

void solve(int choose) {
    if (choose == m) {
        for (int i = 1; i <= m; i++) cout << arr[i] << " ";
        cout << "\n";
        return;
    }

    for (int i = arr.back() + 1; i <= n; i++) {
        arr.push_back(i);
        solve(choose + 1);
        arr.pop_back();
    }
}

int main() {
    cin >> n >> m;
    solve(0);
}
```

240

질문?

문제

- **필수 문제**
- BOJ 6131 (완전 제곱수)
- BOJ 20360 (Binary numbers)
  - 숫자를 이진수로 바꿨을 때, “1”이 있는 위치를 출력하는 문제
  - $13 = 2^0 + 2^2 + 2^3$
  - 답은 `0 2 3`
- BOJ 10819 (차이를 최대로)
- BOJ 15651 (N과 M (3))
- BOJ 15652 (N과 M (4))

- **연습/심화 문제**
- BOJ 1497 (기타콘서트)
  - bit masking을 사용해서도 해결할 수 있고, 재귀를 사용해서도 해결할 수 있다.
- BOJ 15649 (N과 M (1))
- BOJ 15650 (N과 M (2))
- BOJ 9095 (1, 2, 3 더하기)
- BOJ 24954 (물약 구매)

240

질문?

쉬는 시간

- 5분 후 수업 다시 시작합니다.

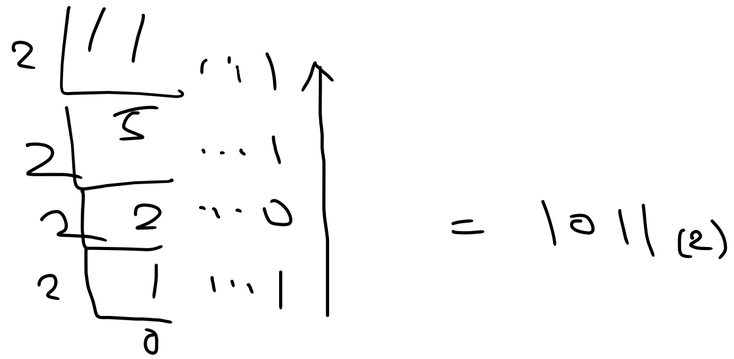
- 다룰 내용
  - 2, 10진수 간 변환 방법
  - 이진(binary) 연산자들을 다루는 방법
  - bit masking을 통해 경우의 수가 2^N인 완전 탐색을 쉽게 하는 방법

- **이 내용부터는 듣고 싶은 분만 남아서 들으시면 됩니다.**

2023 Summer Algorithm Camp

비트 마스킹 - 이진수

- 10진수에서 2진수로의 변환

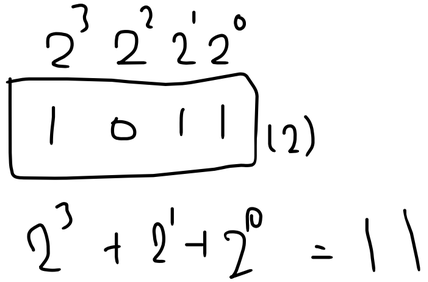


- 0이 될 때까지 수를 계속 2로 나눔
- 2로 나눈 나머지를 오른쪽에 씀
- 나머지를 반대 순서로 쓰면 이진수로 변환할 수 있음

2023 Summer Algorithm Camp

비트 마스킹 - 이진수

- 2진수에서 10진수로의 변환



- $2^n, 2^{n-1}, \cdots, 2^2, 2^1, 2^0$
- 이진수로 1인 위치들만 더하면 된다.

2023 Summer Algorithm Camp

비트 마스킹

- bitwise AND
  - `  0011`
  - `& 0110`
  - `= 0010`

- bitwise OR
  - `  0011`
  - `¦ 0110`
  - `= 0111`

- bitwise left-shift
  - 011_2 << 3_{10} = 011000_2

비트 마스킹

- bitwise left-shift 연산의 특징
  - 1 << k는 2^k와 동일하다.
  - 2^k는 이진수로 항상 100...00 형태이고, 0의 개수는 k와 같음
  - 따라서, “1”을 왼쪽으로 k만큼 밀고 나서의 값은 2^k와 같음

- 예시
  - 2^2는 이진수로 100_2임
  - 1 << 2와 같음

240

질문?

2023 Summer Algorithm Camp

비트 마스킹

- 길이가 N인 이진수가 저장할 수 있는 값의 범위
  - 0 이상 $2^N - 1$ 이하
  - 총 2^N개
  - 최댓값이 $2^N - 1$인 이유) N개의 자릿수를 모두 1로 채우면(1111...111), $2^{N-1} + 2^{N-2} + \cdots + 2^1 + 2^0 = 2^N - 1$

- 물건이 N개 있을 때, 선택할지/말지 정하는 경우의 수
  - 물건마다 독립적으로 정할 수 있으므로 $2 * 2 * \cdots * 2 = 2^N$

- 경우의 수가 2^N = 길이가 N인 이진수의 개수
  - 일대일 대응(bijective)시킬 수 있음

2023 Summer Algorithm Camp

비트 마스킹

- 길이가 N인 이진수에 대하여 각 자릿수를
  - 0이면 false
  - 1이면 true
- 로 하여 0부터 2^N - 1까지 하나의 수를 하나의 경우로 대응시킬 수 있다.

- N = 3
  - 000
  - 001
  - 010
  - 011
  - 100
  - 101
  - 110
  - 111

2023 Summer Algorithm Camp

비트 마스킹

- C++에서 k-bit인 정수 n이 주어졌을 때, 켜진 비트의 위치를 찾아보자.
  - k-bit인 정수라는 것은 이진수로 인덱스가 0부터 k-1까지 있다는 의미이다.
- i번째 비트가 켜져 있는지 판단하는 방법
  - n에서 i번째 비트 빼고 전부 끈다: n & (1 << i)

	1	1	0	1	0	1	
&	0	0	0	1	0	0	= 1 << 2
	0	0	0	1	0	0	

  - 끄고 나서의 수가 0보다 큰지 확인하면 된다.
  - 연산자 우선순위에 조심해야 한다. 비트 연산을 할 때는 항상 괄호 치는 것을 권장한다.

2023 Summer Algorithm Camp

비트 마스킹

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n = 5, k = 4; // n=101(2)
    for (int i = 0; i < k; i++) {
        if ((n & (1 << i)) > 0) {
            cout << i << " "; // 0 2
        }
    }
}
```

2023 Summer Algorithm Camp

비트 마스킹

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n = 5, k = 4; // n=101(2)
    for (int i = 0; i < k; i++) {
        if (n & (1 << i)) {
            cout << i << " "; // 0 2
        }
    }
}
```

2023 Summer Algorithm Camp

비트 마스킹

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n = 3;
    for (int i = 0; i < (1 << n); i++) {
        cout << i << ": ";
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) {
                cout << j << " ";
            }
        }
        cout << "\n";
    }
}
```

```
Output:
0:
1: 0
2: 1
3: 0 1
4: 2
5: 0 2
6: 1 2
7: 0 1 2
```

질문?

예시 문제

- BOJ 1497 (기타콘서트)

- 기타 N개가 주어지고, 기타마다 연주할 수 있는 곡의 목록들이 주어짐. (곡의 개수는 총 M개)
- 최대한 많은 곡을 연주하려 할 때, 필요한 기타의 최소 개수를 구하는 문제 (곡을 하나도 연주할 수 없으면 -1)

- 제한 : $N \leq 10$, $M \leq 50$

- N = 4, M = 5이고, 기타마다 연주할 수 있는 곡의 목록이 다음과 같다. (Y는 연주할 수 있고, N은 연주할 수 없음)
  - **YYYNN**
  - YYNNY
  - **NNNYY**
  - YNNNN
- 문제의 답은 2 (연주할 수 있는 곡의 수 : 5)

240

예시 문제

- $O(2^N NM)$ 풀이가 가능하다.

- 기타마다 사용할지, 사용하지 않을지에 대해서 결정한다.
- 기타가 N개 있으므로 경우의 수는 2^N이다.

- 사용할 기타가 정해지면 연주할 수 있는 곡의 개수를 세는 것은 쉽다.
- 기타가 최대 N개이고, 곡의 수는 M개이므로 각 경우의 수에 대해서 O(NM) 시간이 걸린다.

- 이전에 설명한 재귀 함수 등을 사용해서도 구현할 수 있으나,
- 비트 마스킹을 사용하면 “모든 경우의 수를 순회하는” 구현이 상당히 간단해진다.

예시 문제

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, m; cin >> n >> m;
    vector<string> vec(n);
    string tmp;
    for (auto &s : vec) cin >> tmp >> s;
    int ans = 0, cnt = -1; // ans: 기타 개수, cnt: 곡 개수
    for (int i = 0; i < (1 << n); i++) { // 2^N개의 bit mask를 모두 순회한다.
        vector<int> vis(m);
        int guitar = 0, song = 0; // guitar: 기타 개수, song: 곡 개수
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) { // j번째 기타를 선택하는 경우
                guitar++;
                for (int k = 0; k < m; k++) {
                    if (vec[j][k] == 'Y' && !vis[k]) vis[k] = 1, song++; // k번째 곡을 연주할 수 있음
                }
            }
        }
        if (song) {
            if (cnt < song) cnt = song, ans = guitar;
            else if (cnt == song) ans = min(ans, guitar);
        }
    }
    cout << (cnt == -1 ? -1 : ans);
}
```

240

질문?