SAMSUNG 삼성SDS

SW 아카데미 알고리즘 특강

Contents

오늘의 원리

- 알고리즘은 완전탐색에서 시작한다.
- 알고리즘을 이용하기 위해서는 복잡한 구조의 코드를 정확하게 구현할 수 있는 구현력이 필요하다.
- 시간복잡도란 작동하는 알고리즘의 수행시간을 정량화하는 것을 뜻하며, 일반적으로 Worst Case를 가정하고 계산한다.

학습목표

- 완전탐색 및 복잡한 구현 코드를 작성할 수 있다.
- 알고리즘을 구현하기 전, 미리 구현하고자 하는 알고리즘의 공간복잡도와 시간복잡도를 계산하여 문제의 제약조건 내에 수행될 수 있는 알고리즘인지 판단할 수 있다.

알고리즘

- **알고리즘(Algorithm) 이란?**

고대 페르시아의 수학자 알콰리즈미(Al-Khwarizmi)에서 유래
문제를 해결하기 위한 여러 동작들의 모임

- **알고리즘의 조건**
  - 정밀성 : 변하지 않는 명확한 작업 단계를 가져야 한다.
  - 유일성 : 각 단계마다 명확한 다음 단계를 가져야 한다.
  - 입력 : 정의된 입력을 받아들일 수 있어야 한다.
  - 출력 : 답으로 출력을 내보낼 수 있어야 한다.
  - 유한성 : 특정 수의 작업 이후에 정지해야 한다.
  - 일반성 : 정의된 입력들에 일반적으로 적용할 수 있어야 한다.

알고리즘과 개발자

- **각자의 알고리즘**

개발자는 눈앞의 문제를 해결하기 위해 각자의 알고리즘을 작성한다.
좋은 알고리즘을 만들어내고 문제를 잘 해결하기 위해서는 크게 3가지가 필요하다.

- **구현력**

  본인이 생각한 알고리즘을 빠르고 정확하게 구현할 수 있는 능력
  구현력 향상 => 연습 + 연습 + 연습 + 연습 ...

- **지식(이론, 알고리즘)**

  자료구조, 그래프 이론, 수학적 지식 등
  문제를 해결하기 위해 이용하는 잘 알려진 이론들과 알고리즘들에 대한 이해
  지식 향상 => 책, 강의 등을 이용하여 이론 학습 및 이해

- **응용력**

  습득한 지식들 및 다양한 테크닉들을 응용하여
  해결해야 할 문제에 적용하고 검증하는 능력
  응용력 향상 => 양질의 문제 풀이 + 모자란 지식 학습 + 또 문제 풀이 + 또 공부...

기초 탐색 - 깊이 우선 탐색의 이해와 활용

- **깊이 우선 탐색(Depth First Search)**

- 그래프 탐색 방법의 한가지
- 한 경로로 탐색 하다 특정 상황에서 최대한 깊숙하게 들어가서 확인 후 다시 돌아가 다른 경로로 탐색하는 방식
- 일반적으로 재귀함수를 이용하여 구현, Stack을 이용하여 구현하기도 함
- 구조상 Stack Overflow에 유의해야 함
- 그래프 알고리즘에서 기초가 되는 탐색 방식으로 반드시 숙지가 필요함
- DFS 활용 : 단절선 찾기, 단절점 찾기, 위상정렬, 사이클 찾기 등

기초 탐색 - 깊이 우선 탐색의 이해와 활용

- **DFS 구현(WHITE : 방문 전, GRAY : 방문 중, BLACK : 방문 완료)**

```
for each vertex u ∈ G.V
    u.color = WHITE
    u.parent = NIL

for each vertex u ∈ G.V
    if u.color == WHITE
        DFS(G,u)

DFS(G,u)
    u.color = GRAY
    for each v ∈ G.Adj[u]
        if v.color == WHITE
        v.parent = u
        DFS(G,v)
    u.color = BLACK
```

기초 탐색 - 너비 우선 탐색의 이해와 활용

- **너비 우선 탐색(Breadth First Search)**

- 그래프 탐색 방법의 한가지
- 시작 노드에서 시작하여 인접한 노드를 먼저 탐색하는 방식
- 일반적으로 Queue 자료구조를 이용하여 구현
- 그래프 알고리즘에서 기초가 되는 탐색 방식으로 반드시 숙지가 필요함
- BFS 활용 : 최단경로 찾기, 위상정렬 등

기초 탐색 - 너비 우선 탐색의 이해와 활용

- **BFS 구현(WHITE : 방문 전, GRAY : 방문 중, BLACK : 방문 완료)**

```
BFS(G, s)
   for each vertex u ∈ G.V
      u.color = WHITE
      u.parent = NIL
   s.color = GRAY
   s.parent = NIL
   Q = Ø
   ENQUEUE(Q, s)
   while Q ≠ Ø
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]
         if v.color == WHITE
            v.color = GRAY
            v.parent = u
            ENQUEUE(Q, v)
      u.color = BLACK
```

기초 탐색 - 재귀 함수를 이용한 완전 탐색

- **완전탐색의 설계와 구현**
  - 알고리즘은 완전탐색에서부터 시작
  - 완전탐색의 시간복잡도가 크다면 개선하여 시간복잡도를 작게 만드는 것이 목표
  - 완전탐색을 설계 및 구현 할 수 있다는 것은 문제의 정의와 조건을 정확하게 파악한 것
  - 완전탐색의 구현을 위해 많이 사용되는 재귀 함수를 자유롭게 다룰 수 있어야 함

- **재귀 함수의 구조**

```
int recursion(Parameters)
{
    if(Termination Condition)
    {
        return;
    }
    else
    {
        while(Search Condition)
        {
            recursion(Parameters);
        }
    }
}
```

[재귀함수 구현]

1. 하나의 문제를 여러 부분문제로 나누기
2. 각 부분 문제를 정의하기 위한 상태의 정보를 설계
3. 각 부분 문제의 상태가 원하는 해인지 판별할 조건 설계
4. 각 부분 문제에서 다음 하위문제를 탐색할 조건 설계

정렬의 특성

- **정렬된 데이터의 특성**

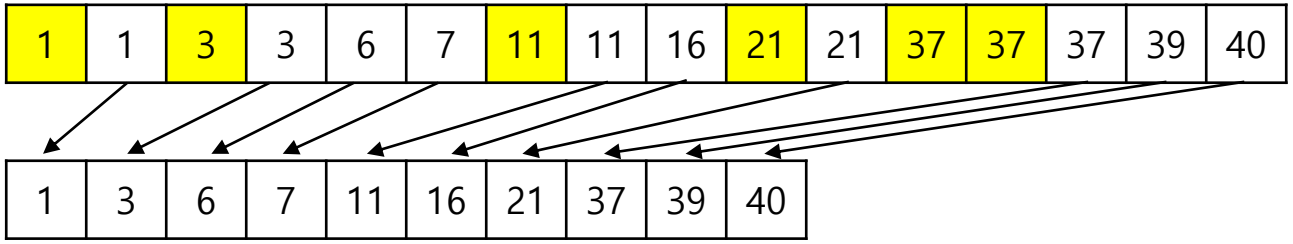
- 정렬은 데이터의 크기 정보(값)를 위치 정보(인덱스)로 확장시킴

- 정렬된 데이터에서 어떤 위치의 데이터 값을 알면,
  그 데이터 앞의 데이터와 뒤의 데이터의 크기를 유추 할 수 있음
- 정렬된 데이터에서 동일 값을 가지는 데이터는 반드시 인접함
- 데이터 중 K 번째로 작은(큰) 값에 빠르게 접근 가능
- 데이터의 속성값이 여러 가지인 경우 정렬의 기준이 여러 가지가 될 수 있음

정렬의 응용

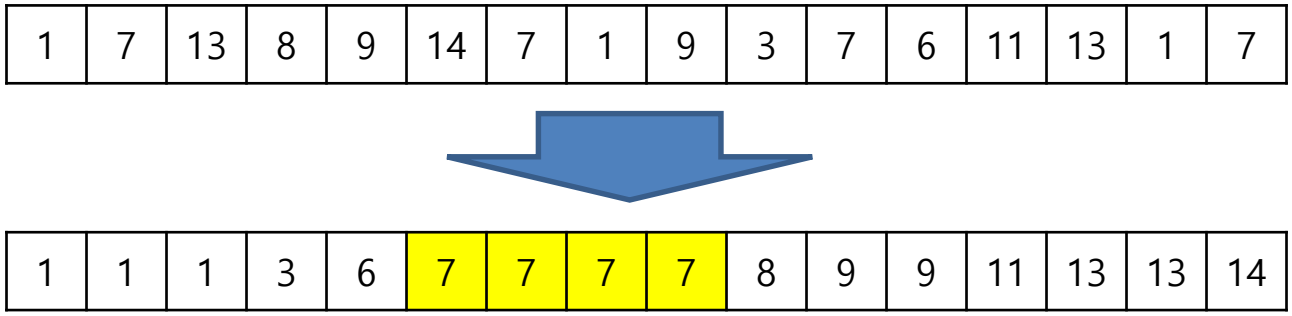
- **유일성 검사 / 중복 제거**

  - 인접한 데이터와의 비교만으로 유일성 검사 및 중복 제거가 가능



- **빈도 구하기**

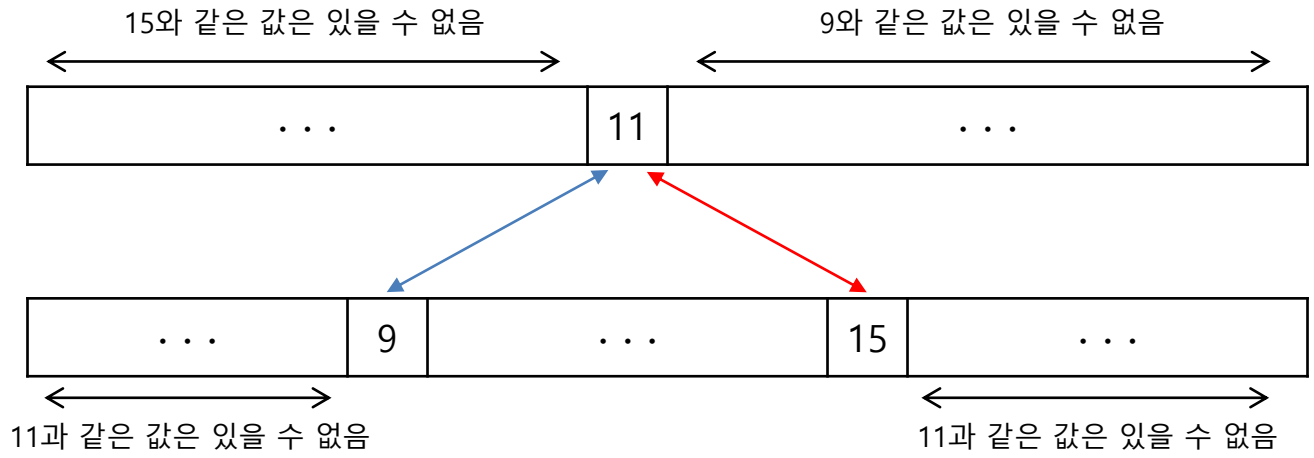
  - 정렬된 데이터를 한번씩만 확인하면 각 숫자의 빈도 수를 확인 가능



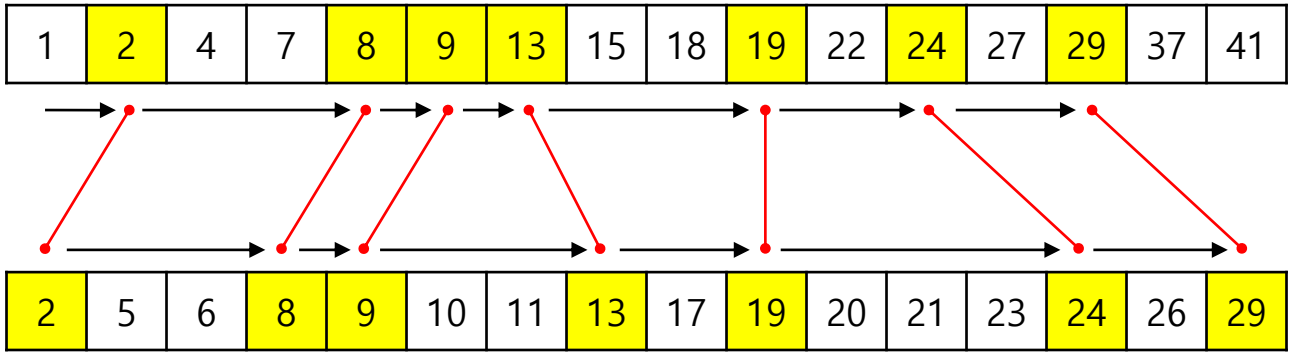
정렬의 응용

- **합집합 / 교집합 구하기**

- 정렬된 두 데이터 집합에서도 정렬된 데이터의 특징을 이용



- 2 pointers 알고리즘을 이용



정렬의 응용

- **이분 탐색**

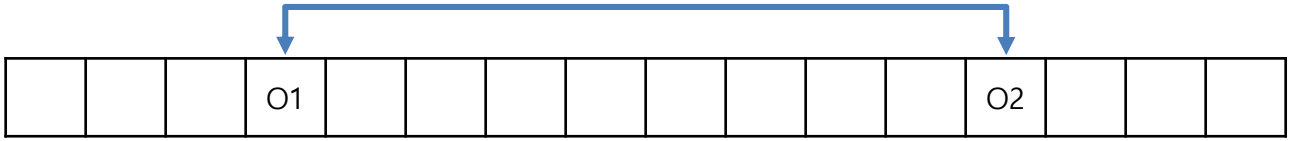
  - 찾고자 하는 데이터의 범위를 ½로 줄여나가며 탐색

표준 라이브러리를 활용한 정렬 이용하기

- **Compare 함수의 이해**

- 표준 라이브러리의 정렬은 Compare 함수를 이용하여 정렬 기준을 재정의하여 사용
- Compare 함수는 2개의 데이터의 위치에 대하여 Swap 여부를 결정하는 함수
- Compare 함수의 인자 중 앞에 있는 데이터가 위치적으로 앞에 있는 데이터

O1 데이터와 O2 데이터의 위치를 서로 바꾸어야 하는지 확인



- Compare 함수가 리턴하는 값에 따라 두 데이터의 Swap 여부 결정

	Java	C++

Compare 함수	`static class comp implements Comparator<Object>{`
	`  @Override`
	`  public int compare(Object o1, Object o2) {`
	`  }`
	`}`
		`bool comp(const Object & o1, const Object & o2)`
		`{`
		`}`

Return value	음수 : 두 데이터를 바꾸지 않음 0 : 두 데이터가 동일함 양수 : 두 데이터를 바꿈	true : 두 데이터를 바꾸지 않음 False : 두 데이터를 바꿈

표준 라이브러리를 활용한 정렬 이용하기

- **표준 라이브러리를 활용한 정렬**

- 표준 라이브러리의 정렬 사용법을 잘 숙지해야 함
- 각 언어별 API 문서를 참조하여 정확한 사용법 숙지 필요

- **C++**

```
#include <algorithm>

template <class RandomAccessIterator> void std::sort(RandomAccessIterator first,
RandomAccessIterator last);
template <class RandomAccessIterator, class Compare> void std::sort(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

- **Java**

```
static void sort(int[] a)
static void sort(int[] a, int fromIndex, int toIndex)
static <T> void sort(T[] a, Comparator<? super T> c)
static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
```

사전문제

다음 코드들을 수행하는데 필요한 연산 횟수를 구하시오.

(1)

```
#include <stdio.h>
int N, MAT[5001][5001], D[5001][5001];
int main() {
        scanf("%d", &N);
        for(int i = 1; i <= N; i++) {
                for(int j = 1; j <= N; j++) {
                        scanf("%d", &MAT[i][j]);
                }
        }
        for(int i = 1; i <= N; i++) {
                for(int j = 1; j <= N; j++) {
                        D[i][j] = D[i-1][j] + D[i][j-1] - D[i-1][j-1] + MAT[i][j];
                }
        }
        printf("%d\n",D[N][N]);
        return 0;
}
```

사전문제

(2)

```
#include <stdio.h>
int N, M, num[100001], ans;
int main(){
        scanf("%d %d", &N, &M);
        for(int i = 1; i <= N; i++){
                scanf("%d", &num[i]);
        }
        ans = -1;
        for(int i = 1; i <= N; i++){
                if(num[i] == M){
                        ans = i;
                        break;
                }
        }
        printf("%d\n",ans);
        return 0;
}
```

사전문제

(3)

```
#include <stdio.h>
int result, idx, N, M;
int num[5001][5001], ans[5000 * 5000 + 1];
int main(){
	scanf("%d %d", &N, &M);
	for(int i = 1; i <= N; i++){
		for(int j = 1; j <= N; j++){
			scanf("%d", &num[i][j]);
		}
	}
	idx = 1;
	for(int i = 1; i <= N; i++){
		for(int j = 1; j <= N; j++){
			num[i][j] += 2;
			if(num[i][j] == M) ans[idx++] = i;
		}
	}
	for(int i = 1; i < idx; i++) result += ans[i];
	printf("%d\n",result);
	return 0;
}
```

문제 접근법

기본적으로 반복문의 범위나 중첩되어있는 반복문의 개수를 보고 판단한다.

사전문제

(1) 첫 번째 반복문을 살펴보면 특정 i값에 대해 j가 N번 `scanf("%d", &MAT[i][j]);` 구문을 반복한다.
i값 역시 1부터 N까지 반복되므로 $N \times N$번 연산을 한다.
두 번째 반복문도 마찬가지로 $N \times N$번의 연산을 하므로 총 연산 횟수는 $2 \times N \times N$번이다.

(2) 첫 번째 반복문은 N번의 연산을 진행하고, 두 번째 반복문은 최소 1번, 최대 N번의 연산을 한다.
따라서 총 연산횟수는 최소 $N+1$번, 최대 $N+N$번이 된다.

(3) 크게 2개의 이중 반복문과 한 개의 단일 반복문으로 나눌 수 있다.
첫 번째 이중 반복문은 $N \times N$번, 두 번째 이중 반복문도 $N \times N$번의 연산을 하고,
마지막 반복문은 최소 0번, 최대 N^2 번의 연산을 한다.
따라서 총 연산횟수는 최소 $2N^2$번, 최대 $3N^2$번이 된다.

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **시간복잡도(Time Complexity)의 정의**

시간복잡도란 작동하는 알고리즘의 수행시간을 정량화하는 것을 뜻한다.

- **시간복잡도 계산**

알고리즘의 시간복잡도는 연산의 횟수 $T(N)$을 구하는 방법이 주로 쓰인다. 통상적으로 **1억(10^8)번의 연산당 1초의 시간이 걸린다**고 간주하여 알고리즘의 수행시간을 예측한다.

우측 예제는 N개의 숫자에서
M이 어디에 있는지 찾는 알고리즘이다.
최선의 경우(Best Case) 1번만에,
최악의 경우(Worst Case) N번만에,
평균적으로(Average Case) $\frac{N}{2}$번만에
찾을 수 있다.

$T(N)$은 최악의 경우인 연산횟수를 나타내며
따라서 $T(N) = N$임을 알 수 있다.

```
#include <stdio.h>
int N, M, num[100001], ans;
int main(){
        scanf("%d %d", &N, &M);
        for(int i = 1; i <= N; i++){
                scanf("%d", &num[i]);
        }
        ans = -1;
        for(int i = 1; i <= N; i++){
                if(num[i] == M){
                        ans = i;
                        break;
                }
        }
        printf("%d\n",ans);
        return 0;
}
```

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **시간복잡도(Time Complexity)의 표기법**

시간복잡도를 표기하는 방법에는 빅-오, 빅-오메가, 세타 표기법 등 다양한 방법이 있다.
빅-오(Big-Oh, $O(N)$) 표기법은 **Worst Case**의 연산 횟수를 나타낸 표기법이고,
빅-오메가(Big-Omega, $\Omega(N)$)표기법은 **Best Case**,
빅-세타(Big-Theta, $\theta(N)$) 표기법은 **Average Case** 연산횟수를 나타낸 표기법이다.

이 중, 일반적으로 시간복잡도를 표현할 땐 빅-오 표기법(Big-O Notation)을 사용한다.

사전문제 3번을 살펴보면 연산의 최대 횟수는 $T(N) = 3N^2$으로 나타낼 수 있다.
빅-오 표기법은 $T(N)$에서 최고차항의 **차수만**을 표기하는 표기법으로써, $O(N^2)$으로 표기한다.

연습문제

다음 코드들의 시간복잡도를 빅-오 표기법으로 나타내시오.

(1)

```
#include <stdio.h>
int N;
int main(){
        scanf("%d", &N);
        if(N < 10) printf("%d\n", N / 2);
        return 0;
}
```

연습문제

(2)

```
#include <stdio.h>
int N, num[11], visit[11];
void backtrack(int order){
        if(order > N){
                for(int i = 1; i <= N; i++) printf("%d ", num[i]);
                printf("\n");
        }
        else{
                for(int i = 1; i <= N; i++){
                        if(visit[i] == 0){
                                visit[i] = 1;
                                num[order] = i;
                                backtrack(order+1);
                                visit[i] = 0;
                        }
                }
        }
        return ;
}
int main(){
        scanf("%d", &N);
        for(int i = 1; i <= N; i++) visit[i] = 0;
        backtrack(1);
        return 0;
}
```

연습문제

(3)

```
#include <stdio.h>
int C[100004], n, a, D[100004], cand, sign, ans;
int main() {
	scanf("%d %d", &n, C + 1);
	for (int i = 2; i <= n; i++) {
		scanf("%d", C + i);
		D[i - 1] = C[i] - C[i - 1];
		if (D[i - 1] < 0) D[i - 1] *= -1;
	}
	ans = 0;
	cand = 0;
	sign = 1;
	for (int i = 2; i < n; i++) {
		cand += sign*D[i];
		sign *= -1;
		if (cand < 0) cand = 0;
		if (cand > ans) ans = cand;
	}
	printf("%d", ans);
}
```

연습문제

(4)

```
#include <stdio.h>
int arr1[100004], arr2[100004], arr3[200010], N;
int main(){
        scanf("%d",&N);
        for(int j=1;j<=N;j++) scanf("%d",arr1+j); // 오름차순 입력
        for(int j=1;j<=N;j++) scanf("%d",arr2+j); // 오름차순 입력
        int i1=1,i2=1,idx=1;
        for(;i1<=N;i1++){
                for(;i2<=N;i2++){
                        if(arr1[i1] > arr2[i2]) arr3[idx++] = arr2[i2];
                        else break;
                }
                arr3[idx++] = arr1[i1];
        }
        for(;i2<=N;i2++) arr3[idx++] = arr2[i2];
        for(int i=1;i<idx;i++) printf("%d ",arr3[i]);
}
```

연습문제

(5)

```
#include <stdio.h>
int N, M, num[100001], low, hi, ans, mid;
int main(){
        scanf("%d %d", &N, &M);
        for(int i = 1; i <= N; i++) scanf("%d", &num[i]); // 오름차순 입력
        low = 1;
        hi = N;
        ans = -1;
        while(low <= hi){
                mid = (low + hi) / 2;
                if(num[mid] < M) low = mid + 1;
                else {
                        ans = mid;
                        hi = mid - 1;
                }
        }
        printf("%d", ans);
        return 0;
}
```

연습문제 접근법

(1) 기본적으로 반복문의 범위나 재귀함수의 구조를 살펴 시간복잡도를 예측한다.

(2) 반복문이 중첩되어 있더라도 시간복잡도가 두 반복문 범위의 곱이 아닐 수 있음에 유의한다.

연습문제

(1) $O(1)$

(2) backtrack 함수를 살펴보면 order가 N보다 작은 경우, 반복문을 실행하며 visit 배열의 값을 하나씩 1로 바꾸며 다음 backtrack 함수를 호출한다. 따라서 함수가 호출될 때마다 범위는 N부터 1씩 줄어들며, order가 N이 되는 순간(backtrack 함수가 N번 호출될 때) 함수는 종료하게 된다. $O(N!)$

(3) 최초 N개의 입력값을 받고, 다음 반복문에서 N번의 연산을 수행하게 된다. $O(N)$

(4) 4번째 반복문을 살펴보면 임의의 i1에 대해 변수 i2는 arr[i1] > arr[i2] 조건문에 따라 반복연산을 수행한다. 다만 유의할 것은 반복문이 호출될 때, 변수 i2가 매번 초기화되는 것이 아니라 기존의 값을 유지한 채로 반복문을 진행한다는 점이다. $O(N)$

(5) while문을 살펴보면 low > hi 조건을 만족하는 순간 반복문이 종료된다. 또한 반복문이 실행될 때마다 low와 hi의 중앙값에 가까운 값으로 갱신됨을 알 수 있다. $O(log\,N)$ 하지만 첫번째 반복문이 N개의 값을 입력받는 동안 N번의 연산이 일어나므로 총 시간복잡도는 $O(N)$.

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **시간복잡도(Time Complexity)의 종류**

- 상수형($O(1)$) : 길이가 N인 배열에서 M번째 배열값을 출력(단, $M < N$)

- 로그형($O(logN)$) : N개의 정렬된 수열에서 이분탐색을 통해 특정 숫자를 탐색
  우선순위 큐(Priority Queue) 또는 힙(Heap) 에서의 원소 삽입, 삭제연산
  모듈러 거듭제곱을 빠르게 연산하기

- 선형($O(N)$) : 정렬되지 않은 길이가 N 인 배열에서 가장 작은 수를 탐색

- 선형로그형 ($O(N\,log\,N)$) : 힙 정렬(Heap Sort), 병합 정렬(Merge Sort), 퀵 정렬의 Average Case

- 2차형($O(N^2)$) : 버블 정렬(Bubble Sort), 삽입 정렬(Insert Sort), 퀵 정렬의 Worst Case

- 3차형($O(N^3)$) : 플로이드-워셜 알고리즘(Floyd-Warshall Algorithm)

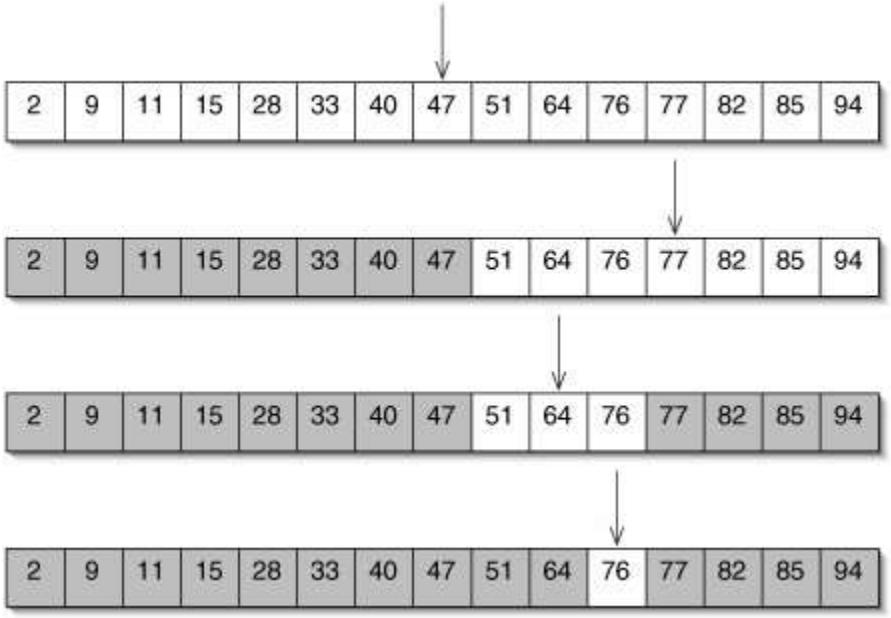
- 지수형($O(2^N)$) : 번호가 매겨진 N 개의 동전을 던졌을 때 나올 수 있는 경우를 출력하는 함수

- 팩토리얼형($O(N!)$) : 1부터 N 까지의 숫자를 나열할 수 있는 모든 방법을 출력하는 함수

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **이분탐색(Binary Search)의 시간복잡도**

오름차순으로 정렬된 N개의 수열에서 특정 수를 이분탐색으로 찾는 경우 **최대 탐색 횟수**를 살펴보자.



탐색을 진행 할 수록 수의 범위는 $\frac{1}{2}$로 줄어들고, 결국 범위가 한 개만 남았을 때 답을 찾는 경우가 최악의 경우이다. 따라서 시간복잡도를 x라 하면 아래와 같은 식을 얻을 수 있다.

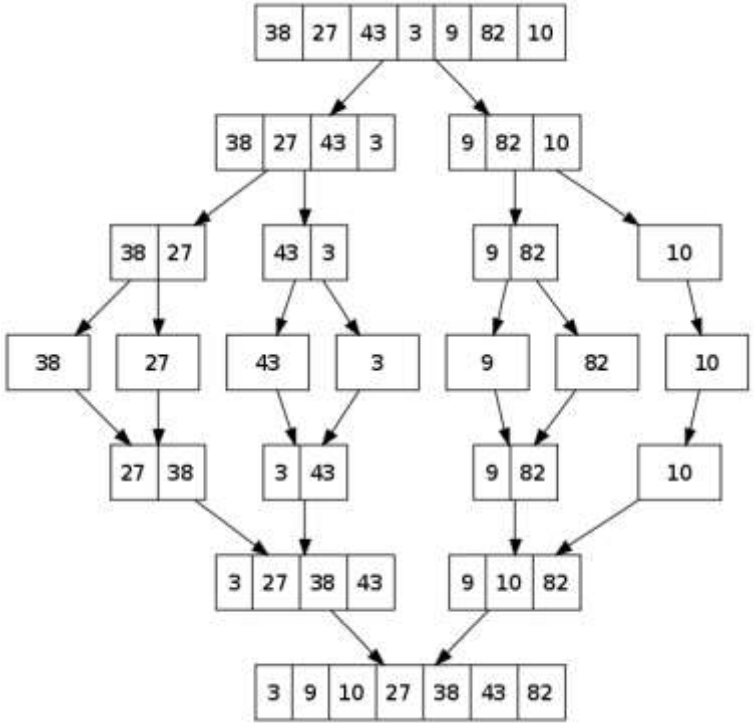
$$N \times \left(\frac{1}{2}\right)^x = 1 \Leftrightarrow N = 2^x \Leftrightarrow x = log\, N$$

(※ 주어진 배열을 3등분 해서 탐색하는 삼분탐색의 시간복잡도는 어떻게 될지 생각해보자.)

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **병합 정렬(Merge Sort)의 시간복잡도**

N개의 수열을 병합 정렬로 오름차순 정렬할 때, **최대 탐색 횟수**를 살펴보자.



앞서 언급한 이분탐색처럼 N개의 원소를 분할할 때 N번의 분할연산이 필요하다.
분할된 원소들을 병합 할 땐 최대 N개의 원소의 순서를 비교해야 하고, $\log_2 N$번의 병합연산이 필요하다.
따라서, $N + N\log_2 N$의 연산이 필요하므로 시간복잡도는 $O(NlogN)$이 된다.

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **거듭제곱을 빠르게 연산하기**

A^P mod C 를 계산하기 위해서는 A를 P번 곱해야 하므로 시간복잡도는 O(P)가 된다. 하지만 거듭제곱하려는 횟수 P가 매우 클 경우(1억 이상) 제한된 시간 내에 연산이 어려우므로 더 빠른 연산방법이 필요하다.

간단한 예를 들어 7^{21}을 아래와 같이 표현할 수 있다.

$7^{21} = 7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7*7$

$7^{21} = 7^{16}*7^4*7$ $(21 = 16 + 4 + 1)$

$= 7^{16}*(7^8)^0*7^4*(7^2)^0*7$ $(21 = 2^4 + 2^2 + 2^0)$

첫 번째 방법은 7을 21번 곱하는 방식으로 표현하였고, 두 번째 방법은 지수를 2의 거듭제곱 형태로 나누어 표현한 것이다. 즉, 거듭제곱해야 하는 지수를 2진수로 나타낸 것과 같다.

$$21_{(10)} = 10101_{(2)}$$

자릿수	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
값	1	0	1	0	1

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **거듭제곱을 빠르게 연산하기**

10진수를 2진수로 변환하는 과정을 응용하여 지수가 매우 큰 거듭제곱을 $O(\log_2(\text{지수}))$의 시간복잡도로 구할 수 있다.

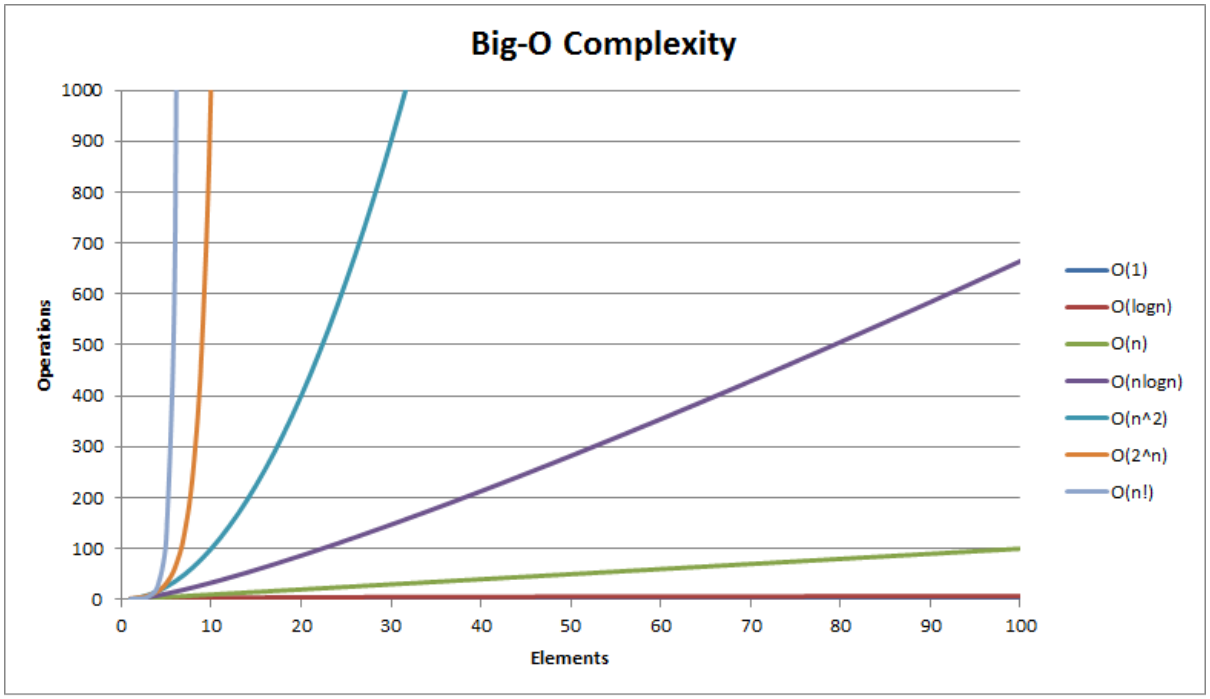
$$21_{(10)} = 10101_{(2)}$$

				2의 거듭제곱 지수	거듭제곱 결과
2	21	--	1 (2^0)	7^1	7^1
2	10	--	0 (2^1)	$7^2(=7^1*7^1)$	7^1
2	5	--	1 (2^2)	$7^4(=7^2*7^2)$	$7^1 * 7^4$
2	2	--	0 (2^3)	$7^8(=7^4*7^4)$	$7^1 * 7^4$
2	1	--	1 (2^4)	$7^{16}(=7^8*7^8)$	$7^1 * 7^4 * 7^{16}$
	0				

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **시간복잡도(Time Complexity)의 비교**

N의 범위가 커질 수록 시간복잡도별 알고리즘의 수행시간의 차이도 커짐을 알 수 있다.



이를 통해 계획하고 있는 알고리즘의 수행시간과 문제에서 주어진 제한시간을 미리 비교하여 시간 내에 수행이 가능한지 판단하는 능력을 키워야 한다.

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **공간복잡도(Space Complexity)**

공간복잡도란 알고리즘의 메모리 사용량에 대한 분석결과로, 알고리즘 문제를 해결하기 위해서 사용하는 메모리의 크기를 말한다.

아래 표는 각 자료형 별 메모리 크기를 나타낸 표이다. (32bit 기준)

자료형	키워드		메모리 크기
	C / C++	JAVA	
문자형	char	char	C/C++ : 1 Byte JAVA : 2 Bytes
정수형	short	short	2 Bytes
	int	int	4 Bytes
	long long	long	8 Bytes
부호없는 문자형	unsigned char	없음	1 Byte
부호없는 정수형	unsigned short		2 Bytes
	unsigned int		4 Bytes
	unsigned long long		8 Bytes
부동 소수형	float	float	4 Bytes
	double	double	8 Bytes

시간복잡도(Time Complexity)와 공간복잡도(Space Complexity)

- **비트연산**
  - 정수형 자료형에 사용 가능

연산자	명칭	기능	예시
~	NOT	비트의 값을 반대로 뒤집기	~00001101 = 11110010
&	AND	양쪽 비트가 모두 1일 때만 1	00001101 & 00000011 = 00000001
|	OR	양쪽 비트 중 하나라도 1이면 1	00001101 | 00000011 = 00001111
^	XOR	양쪽 비트 값이 서로 다를 때 1	00001101 ^ 00000011 = 00001110
<<	L-shift	비트를 왼쪽으로 이동	00001101 << 00000011 = 01101000
>>	R-shift	비트를 오른쪽으로 이동	00001101 >> 00000011 = 00000001
>>>	UR-shift	(Java 전용) 부호 비트를 무시하고 비트를 오른쪽으로 이동	(8bit 정수형을 기준 예시) 11111011(-5) >> 00000001 = 11111101 11111011(-5) >>> 00000001 = 01111101

- Flag와 Mask를 이용한 원하는 비트 설정하기
  - ▶ flag |= mask : mask에 1로 설정된 비트를 켜기
  - ▶ flag &= ~mask : mask에 1로 설정된 비트를 끄기
  - ▶ flag ^= mask : mask에 1로 설정된 비트를 토글(켜진 것은 끄고, 꺼진 것은 켜기)

연습문제

다음 코드들의 시간복잡도와 공간복잡도를 구하시오.

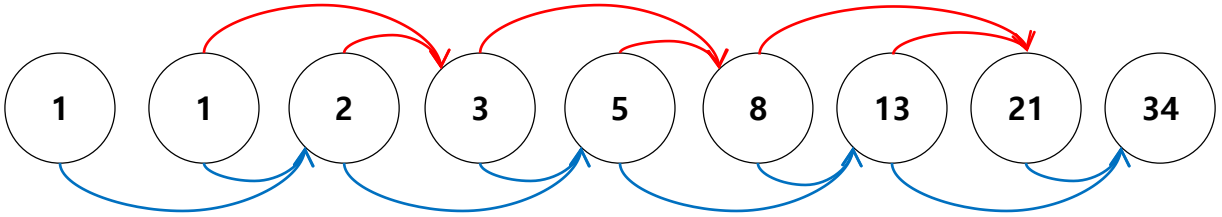
```
#include <stdio.h>
int N, M, num[5001][5001];
int main() {
	scanf("%d %d", &N, &M);
	for(int i = 1; i <= N; i++) {
		// N개의 정수가 오름 차순으로 입력
		for(int j = 1; j <= N ; j++) {
			scanf("%d", &num[i][j]);
		}
	}
	int ans = 0;
	for(int i = 1; i <= N; i++) {
		int cand = 0, low = 1, hi = N;
		while(low <= hi) {
			int mid = (low + hi) / 2;
			if(num[i][mid] < M) low = mid + 1;
			else {
				cand = mid;
				hi = mid - 1;
			}
		}
		ans += cand;
	}
	printf("%d", ans);
	return 0;
}
```

연습문제

(1) 시간복잡도 : $O(N^2)$, 공간복잡도 : N^2

피보나치 수열(Fibonacci Sequence)

피보나치 수열이란 다음 항이 이전 두 개의 항의 합으로 표현되는 수열을 뜻하며 다음과 같은 순서로 진행된다.



피보나치 수열의 N번째 항을 구하는 다양한 방법들에 대해 알아보도록 하자.

일반적인 재귀함수를 사용하는 방법 $\left(O(2^N)\right)$

아래와 같은 재귀함수를 구현하여 피보나치 수열의 값을 구할 수 있다.

f(int n){
    if(n<=2) return 1;
    else return $f(n-1)+f(n-2)$;
}

n이 5일 때의 함수 호출 과정과 호출 횟수를 살펴보면 다음과 같다.

1 2 3 4 5 6 7

$$f(5)=f(4)+f(3)=\big(f(3)+f(2)\big)+f(3)=\Big(\big(f(2)+f(1)\big)+f(2)\Big)+f(3)$$

8 9

$$=\big((1+1)+1\big)+\big(f(2)+f(1)\big)=\big((1+1)+1\big)+(1+1)=5$$

n이 5일 때 9번의 f함수를 호출하게 되며, 피보나치 수열을 구하기 위해서 호출 해야 하는 함수의 횟수는 n이 증가할수록 기하급수적으로 증가하게 된다.

반복문을 사용하는 방법 $(O(N))$

아래와 같은 반복문을 구현하여 피보나치 수열의 값을 구할 수 있다.

```
g(n){
    a = b = c = 1;
    for(int i=3;i<=n;i++){
        c = a + b;
        a = b;
        b = c;
    }
    return c;
}
```

n번째 피보나치 수열의 값은 시간복잡도 $O(n)$으로 구해낼 수 있다. 따라서 k개의 서로 다른 피보나치 수열의 값을 구하기 위해선 $O(nk)$의 시간이 소요된다.

동적계획법을 사용하는 방법 $(O(N))$

아래와 같은 반복문을 구현하여 피보나치 수열의 값을 구할 수 있다.

```
h(n){
    D[1] = D[2] = 1;
    for(int i=3;i<=n;i++){
        D[i] = D[i − 1] + D[i − 2];
    }
    return D[n];
}
```

n번째 피보나치 수열의 값은 시간복잡도 $O(n)$으로 앞에서 살펴본 반복문과 동일한 속도로 구할 수 있으나, n보다 작은 다른 피보나치 수열의 값을 추가적으로 구해야 할 경우 $O(1)$의 시간복잡도로 빠르게 구해낼 수 있다.

행렬(Matrix)을 사용하는 방법 $(O(log\ N))$

피보나치 수열 $f(n)$의 일반항은 $f(n) = f(n-1) + f(n-2)$이므로 아래와 같은 행렬식으로 나타낼 수 있다.

$$\begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix}$$

따라서 위 식은

$$\begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-2) \\ f(n-3) \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} f(2) \\ f(1) \end{pmatrix}$$

으로 나타낼 수 있다. 즉, 시간복잡도 $O(\log_2 N)$만에 피보나치 수열의 값을 구할 수 있다.

(예) $f(7)$를 구하는 경우

$$\begin{pmatrix} f(7) \\ f(6) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^5 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 & 5 \\ 5 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 13 \\ 8 \end{pmatrix}$$ 이다.

따라서 $f(7) = 13$임을 알 수 있다.

SAMSUNG 삼성SDS

감사합니다