

#2. 정렬과 이분 탐색

나정휘

<https://justicehui.github.io/>

목차

여러가지 정렬 알고리즘

정렬 STL

기수 정렬

이분 탐색

결정 문제와 최적화 문제

여러가지 정렬 알고리즘

버블 정렬

버블 정렬 알고리즘

- 서로 인접한 두 원소를 비교해서 정렬하는 알고리즘

- 5 3 4 2 1 (5 > 3 이므로 교환)	3 2 1 4 5 (3 > 2 이므로 교환)
- 3 5 4 2 1 (5 > 4 이므로 교환)	2 3 1 4 5 (3 > 1 이므로 교환)
- 3 4 5 2 1 (5 > 2 이므로 교환)	2 1 3 4 5 (3 < 4 이므로 교환 X)
- 3 4 2 5 1 (5 > 1 이므로 교환)	2 1 3 4 5 (4 < 5 이므로 교환 X)
- 3 4 2 1 5 (첫 번째 순회 끝)	2 1 3 4 5 (세 번째 순회 끝)
- 3 4 2 1 5 (3 < 4 이므로 교환 X)	2 1 3 4 5 (2 > 1 이므로 교환)
- 3 4 2 1 5 (4 > 2 이므로 교환)	1 2 3 4 5 (2 < 3 이므로 교환 X)
- 3 2 4 1 5 (4 > 1 이므로 교환)	1 2 3 4 5 (3 < 4 이므로 교환 X)
- 3 2 1 4 5 (4 < 5 이므로 교환 X)	1 2 3 4 5 (4 < 5 이므로 교환 X)
- 3 2 1 4 5 (두 번째 순회 끝)	1 2 3 4 5 (네 번째 순회 끝)

- i번째 순회에서 i번째로 큰 값이 N-i+1번째로 이동
- 따라서 N-1번만 순회하면 정렬 끝

버블 정렬

```
● ● ●

#include <bits/stdc++.h>
using namespace std;

int N, A[1010];
void BubbleSort(){
    for(int i=1; i<N; i++){
        for(int j=1; j<N; j++){
            if(A[j] > A[j+1]) swap(A[j], A[j+1]);
        }
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    BubbleSort();
    for(int i=1; i<=N; i++) cout << A[i] << "\n";
}
```

선택 정렬

선택 정렬 알고리즘

- 가장 작은 원소를 맨 앞에 있는 원소와 교환하면서 정렬하는 알고리즘
 - 5 3 4 **1** 2 가장 작은 1을 찾아서 첫 번째 원소와 교환
 - **1** 3 4 5 **2** 이미 정렬된 첫 번째 원소를 제외하고 가장 작은 원소인 2를 찾아서 두 번째 원소와 교환
 - **1** **2** 4 5 **3** 1~2번째 원소 제외하고 가장 작은 원소인 3을 찾아서 세 번째 원소와 교환
 - **1** **2** **3** 5 **4** 1~3번째 원소 제외하고 가장 작은 원소인 4를 찾아서 네 번째 원소와 교환
 - **1** **2** **3** **4** 5 정렬 끝
- i번째 단계가 끝나면 1..i번째로 작은 수가 올바른 자리로 이동함
- 따라서 N-1단계까지 수행하면 정렬 끝

선택 정렬

```
● ● ●

#include <bits/stdc++.h>
using namespace std;

int N, A[1010];
void SelectionSort(){
    for(int i=1; i<N; i++){
        int pos = i;
        for(int j=i+1; j<=N; j++){
            if(A[pos] > A[j]) pos = j;
        }
        swap(A[i], A[pos]);
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    SelectionSort();
    for(int i=1; i<=N; i++) cout << A[i] << "\n";
}
```

삽입 정렬

삽입 정렬 알고리즘

- 앞에 있는 원소부터 차례대로 보면서, 올바른 자리를 찾아서 집어넣는 알고리즘
 - 5 3 4 1 2 가장 앞에 있는 5만 고려했을 때는 잘 정렬된 상태
 - 3 5 4 1 2 두 번째 원소인 3을 적당한 곳에 삽입해서 1~2번째 원소를 정렬된 상태로 조정
 - 3 4 5 1 2 세 번째 원소인 4를 적당한 곳에 삽입해서 1~3번째 원소를 정렬된 상태로 조정
 - 1 3 4 5 2 네 번째 원소인 1을 적당한 곳에 삽입해서 1~4번째 원소를 정렬된 상태로 조정
 - 1 2 3 4 5 다섯 번째 원소인 2를 적당한 곳에 삽입해서 1~5번째 원소를 정렬된 상태로 조정
- i번째 단계가 끝나면 1..i번째 원소는 정렬된 상태로 바뀜
- 따라서 N단계까지 수행하면 정렬 끝

삽입 정렬

```
● ● ●

#include <bits/stdc++.h>
using namespace std;

int N, A[1010];
void InsertionSort(){
    for(int i=2; i<=N; i++){
        int pos = i, key = A[i];
        while(pos > 1 && A[i] < A[pos-1]) pos--;
        for(int j=i; j>pos; j--) A[j] = A[j-1];
        A[pos] = key;
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    InsertionSort();
    for(int i=1; i<=N; i++) cout << A[i] << "\n";
}
```

삽입 정렬

```
● ● ●

#include <bits/stdc++.h>
using namespace std;

int N, A[1010];
void InsertionSort(){
    for(int i=2; i<=N; i++){
        for(int j=i-1; j>=1; j--){
            if(A[j] > A[j+1]) swap(A[j], A[j+1]);
            else break;
        }
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    InsertionSort();
    for(int i=1; i<=N; i++) cout << A[i] << "\n";
}
```

시간 복잡도 계산

시간 복잡도

- 최악의 경우, 최선의 경우, 평균의 경우로 나눠서 분석
 - 문제 풀 때는 주로 최악의 경우만 고려하는 경우가 많음
- 최악의 경우: 연산이 최대한 많이 수행되도록 하는 악의적인 입력이 들어왔을 때의 연산 횟수
- 최선의 경우: 연산이 최대한 적게 수행되도록 하는 입력에서의 연산 횟수
- 평균의 경우: 가능한 모든 입력에서의 연산 횟수의 평균
 - 정렬 알고리즘에서는 가능한 $N!$ 가지의 순열에서의 연산 횟수의 합을 구한 다음 $N!$ 으로 나눈 것
 - 랜덤으로 만들어진 데이터에서 연산 횟수의 기댓값으로 생각해도 됨

시간 복잡도 계산

최악의 경우

- 기본 연산
 - 두 수를 비교하는 연산
 - 두 수의 위치를 교환하는 연산
- 버블 정렬
 - 배열을 한 번 순회할 때마다 비교는 정확히 $N-1$ 번, 교환은 최대 $N-1$ 번
 - 이 과정을 $N-1$ 번 반복하므로 $O(N^2)$
- 선택 정렬
 - i 번째 단계에서 $i..N$ 번째 원소 중 최솟값을 찾을 때 비교는 정확히 $N-i$ 번, 이후 교환 1번
 - 이 과정을 $i=1..N-1$ 일 때 수행하므로 $O(N^2)$
- 삽입 정렬
 - i 번째 단계에서 i 번째 원소는 최대 $i-1$ 번 이동 가능, 따라서 비교와 교환 모두 최대 $i-1$ 번
 - 이 과정을 $i=2..N$ 일 때 수행하므로 $O(N^2)$

시간 복잡도 계산

최선의 경우

- 버블 정렬
 - N-1번의 순회를 모두 수행하면 항상 $O(N^2)$
 - 하지만 이미 배열이 정렬되어 있으면 더 이상 순회를 할 필요가 없음
 - 즉, 한 번의 순회에서 교환이 한 번도 발생하지 않으면 그대로 종료해도 됨
- 처음부터 배열이 정렬되어 있으면 한 번의 순회만 사용해 정렬 가능
- 따라서 최선의 경우에는 $O(N)$



```
int N, A[1010];
void BubbleSort(){
    for(int i=1; i<N; i++){
        bool changed = false;
        for(int j=1; j<=N-i; j++){
            if(A[j] > A[j+1]) swap(A[j], A[j+1]), changed = true;
        }
        if(!changed) break;
    }
}
```

시간 복잡도 계산

최선의 경우

- 선택 정렬
 - 항상 $N-1$ 개의 단계를 거쳐야 하므로 최선의 경우에도 $O(N^2)$
- 삽입 정렬
 - 삽입 정렬의 교환 횟수는 inversion의 개수와 동일함
 - $\text{inversion} := x < y$ 이면서 $A[x] > A[y]$ 인 순서쌍 (x, y)
 - 따라서 inversion의 개수가 0일 때, 즉 배열이 이미 정렬되어 있을 때가 최선의 경우
 - 매 단계마다 원소가 한 번도 이동하지 않으므로 $O(N)$

시간 복잡도 계산

평균의 경우

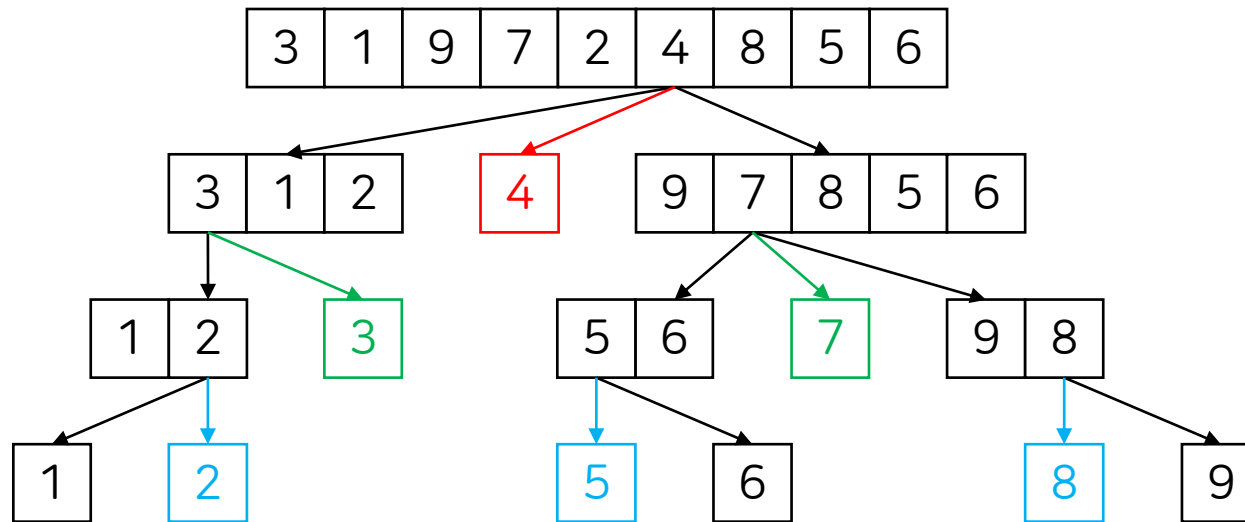
- 삽입 정렬

- 길이가 N인 모든 N! 가지의 순열에서 inversion 개수의 총합은?
- 모든 순열의 inversion 개수의 합 = $1 \leq x < y \leq N$ 인 x y에 대해, x가 y보다 뒤에 나오는 순열 개수의 합
- x가 y보다 뒤에 나오는 순열의 개수 = $\binom{N}{2} \times (N - 2)! = \frac{N!}{2}$
- x, y를 고정하는 방법의 수 = $\binom{N}{2} = \frac{N(N-1)}{2}$
- 따라서 모든 순열에서 inversion 개수의 총합은 $\frac{N!N(N-1)}{4}$
- 평균적으로 $\frac{N(N-1)}{4}$ 개이므로 평균 시간 복잡도는 $O(N^2)$

퀵 정렬

퀵 정렬 알고리즘

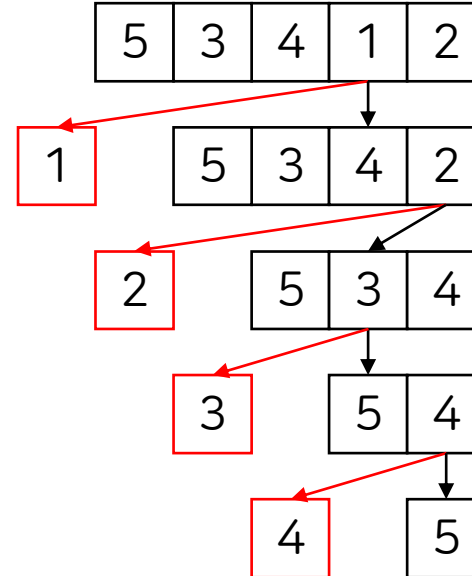
- 아무 원소를 하나 선택해서 pivot이라고 잡은 다음
- pivot보다 작은 값은 왼쪽, 큰 값은 오른쪽으로 옮기고
- 왼쪽과 오른쪽을 각각 재귀적으로 정렬



퀵 정렬

퀵 정렬 시간 복잡도

- 최선의 경우
 - 매번 $(n-1)/2 : 1 : (n-1)/2$ 로 분할되는 경우
 - $T(n) = 2T(n/2) + c*n$
 - pivot 기준 좌우로 나누는 과정에서 $\Theta(n)$ 만큼의 연산이 필요함
 - $T(n) = O(n \log n)$
- 최악의 경우
 - 매번 $0 : 1 : n-1$ 로 분할되는 경우
 - $T(n) = T(n-1) + c*n$
 - $T(n) = O(n^2)$



퀵 정렬

퀵 정렬 시간 복잡도

- 평균의 경우

- pivot이 i 번째 데이터이면, 분할 후 왼쪽에 $i-1$ 개, 오른쪽에 $n-i$ 개의 데이터가 있음
 - $T(n) = T(i-1) + T(n-i) + c*n$
 - $i = 0..n-1$ 일 때의 합을 n 으로 나눈 값이 평균 시간 복잡도
- $T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(i) + T(n-i) + cn = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + cn$
- 양변에 n 을 곱하면 $nT(n) = 2 \sum_{i=0}^{n-1} T(i) + cn^2$
- n 대신 $n-1$ 을 대입하면 $(n-1)T(n-1) = 2 \sum_{i=0}^{n-2} T(i) + c(n-1)^2$
- $nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$
 - c 는 상수이므로 n 이 커질수록 영향이 작아지기 때문에 없애도 무방함
- 위 식을 $nT(n)$ 에 대해 정리하면 $nT(n) = (n+1)T(n-1) + 2cn$
- 양변을 $n(n+1)$ 로 나누면 $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$

퀵 정렬

퀵 정렬 시간 복잡도

- 평균의 경우

- $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$
- n 대신 $n, n-1, n-2, \dots, 2$ 를 대입하면
 - $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$
 - $\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$
 - $\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$
 - ...
 - $\frac{T(3)}{4} = \frac{T(2)}{3} + \frac{2c}{4}$
 - $\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$
- 잘 정리하면 $\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{n+1} \frac{1}{i} \approx \frac{T(1)}{2} + 2c \ln(n+1)$ 이고, $T(1)$ 은 상수
- 따라서 $\frac{T(n)}{n+1} = O(\log n), T(n) = O(n \log n)$

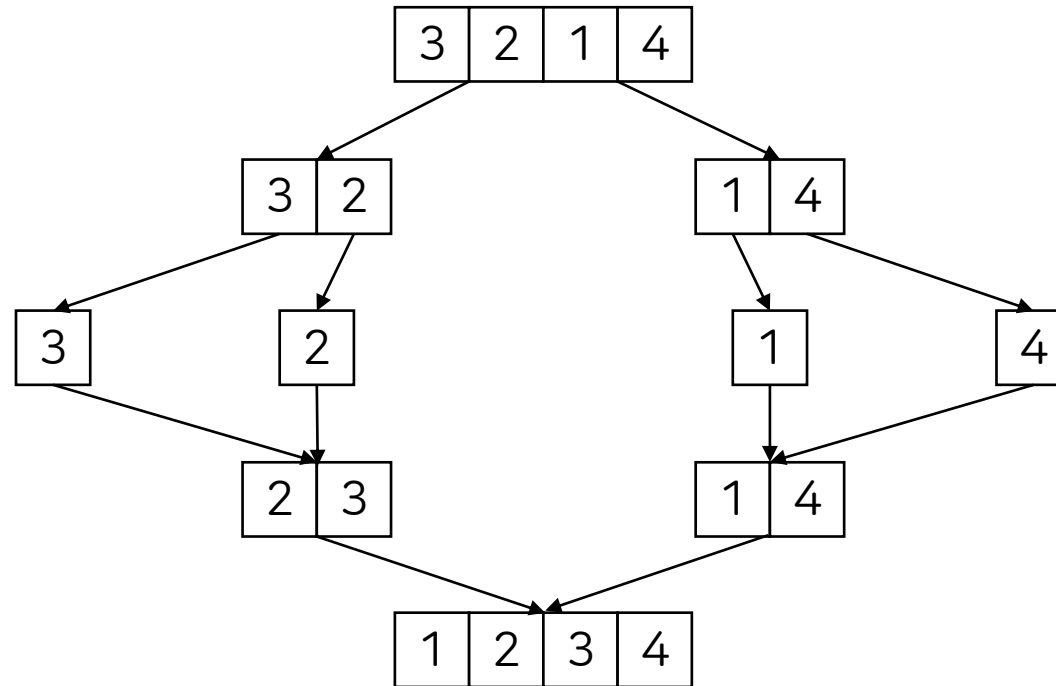
합병 정렬

합병 정렬 알고리즘

- 배열의 구간 $[l, r]$ 을 정렬하는 알고리즘
- $l = r$ 이면 직접 해결
 - 원소가 하나인 배열은 이미 정렬된 상태
- $l < r$ 이면 더 작은 문제로 나눠서 해결
 - m 을 l 과 r 의 중간 지점이라고 할 때
 - $[l, m]$ 과 $[m+1, r]$ 을 각각 정렬한 다음 (분할)
 - 정렬된 두 배열을 정렬된 하나의 배열로 합병 (정복)

합병 정렬

합병 정렬 알고리즘 (애니메이션)



합병 정렬

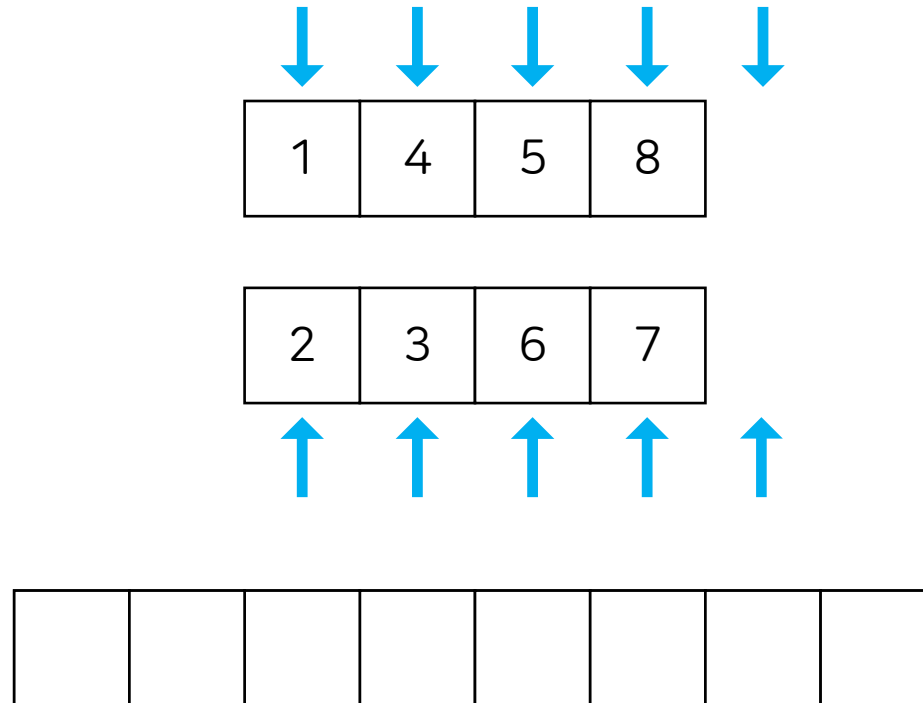
합병 정렬 알고리즘

- 정렬된 두 배열 A, B를 정렬된 하나의 배열 V로 합병하는 방법
 - 두 배열 모두에서 가장 작은 값은 A[0] 또는 B[0]
 - 둘 중 더 작은 값을 V[0]에 저장
 - $A[0] \leq B[0]$ 이었다면 $V[0] = A[0]$
 - A[0]을 제외했을 때 가장 작은 값은 A[1] 또는 B[0]
 - 둘 중 더 작은 값을 V[1]에 저장
 - 반복
 - 두 배열에서 각각 가장 작은 값을 가리키는 인덱스를 저장해야 함

합병 정렬

합병 정렬 알고리즘 (애니메이션)

- 정렬된 두 배열 A, B를 정렬된 하나의 배열 V로 합병하는 방법



합병 정렬

합병 정렬 알고리즘

- 정렬된 두 배열 A, B를 정렬된 하나의 배열 V로 합병하는 방법
 - 매번 두 화살표 중 하나는 한 칸 뒤로 이동
 - 두 화살표는 합쳐서 $r-l+1$ 번 뒤로 이동
- 따라서 배열의 길이를 n 이라고 하면 시간 복잡도는 $O(n)$

합병 정렬

합병 정렬 시간 복잡도

- n 을 정렬해야 하는 배열의 길이($= r-l+1$)라고 하면
- $T(1) = O(1)$
- $T(n) = 2T(n/2) + O(n) = O(n \log n)$
 - 가로 길이 n , 세로 길이 $\log n$ 인 직사각형의 넓이와 동일

n							
n/2				n/2			
n/4		n/4		n/4		n/4	
n/8	n/8	n/8	n/8	n/8	n/8	n/8	n/8
...							
1	1	1	1	1	1	1	1

합병 정렬

합병 정렬 알고리즘



```
constexpr int MAX_N = 1'000;

void Merge(int A[], int s, int m, int e){
    static int tmp[MAX_N];
    int i = s, j = m + 1, idx = s;
    while(i <= m && j <= e){
        if(A[i] < A[j]) tmp[idx++] = A[i++];
        else tmp[idx++] = A[j++];
    }
    while(i <= m) tmp[idx++] = A[i++];
    while(j <= e) tmp[idx++] = A[j++];
    for(int k=s; k<=e; k++) A[k] = tmp[k];
}

void MergeSort(int A[], int s, int e){
    if(s == e) return;
    int m = (s + e) / 2;
    MergeSort(A, s, m);
    MergeSort(A, m+1, e);
    Merge(A, s, m, e);
}
```

정렬 알고리즘의 장단점

지금까지 배운 정렬 알고리즘

이름	최선	평균	최악	추가 공간
버블 정렬	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
퀵 정렬	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
합병 정렬	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

정렬 알고리즘의 장단점

상황에 따른 정렬 알고리즘 선택

- 배열이 이미 대부분 정렬되어 있다면?
 - 버블 정렬이나 삽입 정렬이 좋을 수 있음
- 일반적인 상황이라면?
 - 합병 정렬이나 퀵 정렬이 좋음
 - 비교 연산 횟수는 합병 정렬이 더 적지만, 실제 실행 시간은 퀵 정렬이 더 빠름
 - 캐시히트 등의 영향
- 최악의 경우까지 고려해야 한다면?
 - 퀵 정렬은 최악의 경우에 $O(n^2)$ 이므로 합병 정렬을 사용하는 것이 좋음
- 위치를 교환하는 것에 비해 두 값을 비교하는 연산이 훨씬 오래 걸린다면?
 - 비교 연산 횟수는 퀵 정렬보다 합병 정렬이 더 적기 때문에 합병 정렬이 좋음
- 공간이 부족한 환경이라면?
 - 합병 정렬은 $O(n)$ 만큼의 추가 공간이 필요하므로 다른 알고리즘을 사용하는 것이 좋음

정렬 알고리즘의 장단점

C++ std::sort의 구현

- 실제로 C++에서 제공하는 정렬 함수는 여러 정렬 알고리즘을 섞어서 사용함
 - 일단 퀵 정렬 수행
 - 분할된 덩어리의 크기가 16 이하가 되면 더 이상 정렬하지 않고 그대로 놔둠
 - 재귀 깊이가 $2 \log n$ 을 넘어가면 힙 정렬 수행
 - 힙 정렬은 $O(1)$ 만큼의 추가 공간을 사용해 최악의 경우에도 $O(n \log n)$ 시간에 정렬
 - 평균적으로 퀵 정렬, 합병 정렬에 비해 조금 느림
 - 여기까지 오면 대부분 정렬되어 있는 상황이지만, 크기가 16 이하인 몇몇 구간은 정렬이 안 되어 있을 수 있음
 - 배열 전체에 대해 삽입 정렬 수행
 - 각 원소는 최대 16칸 이동하므로 $O(n)$ 에 정렬 가능
 - 최악의 경우에도 $O(n \log n)$ 을 보장함

정렬 STL

정렬 STL

비교 기반 정렬

- 아래 두 가지 연산만 이용해서 원소를 정렬하는 방법
 - 배열의 두 원소 $A[i]$ 와 $A[j]$ 중 어떤 원소가 더 앞에 와야 하는지 알아내는 연산 (비교 연산)
 - 배열의 원소를 이동/복사하는 연산
- 비교 연산
 - 나올 수 있는 결과는 3가지
 - $A[i]$ 가 먼저 와야 한다 / $A[j]$ 가 먼저 와야 한다 / 둘 중 뭐가 먼저 와도 상관없다
 - 둘 중 어느 것이 먼저 오더라도 상관없는 경우, 두 원소가 “동등하다”라고 부름
- “두 원소 중 더 먼저 와야 하는 원소”를 구할 수만 있다면, 정렬하고자 하는 대상의 타입/성질은 중요하지 않음

정렬 STL

비교 기반 정렬 - 예시

- 정수 배열의 오름차순 정렬
 - $A[x] < A[y]$ 이면 $A[x]$ 가 먼저 와야만 함
 - $A[x] = A[y]$ 이면 둘 중 어느 것이 먼저 오더라도 상관없음
 - bubble_sort 함수에서는 A의 값을 직접 보지 않는다는 것에 주목하자(추상화).

```
int N, A[1010];

bool must_i_come_before_j(int i, int j){
    return A[i] < A[j];
}

void bubble_sort(){
    for(int i=0; i<N; i++){
        for(int j=1; j<N; j++){
            if(must_i_come_before_j(j, j-1)){
                swap(A[j-1], A[j]);
            }
        }
    }
}
```

정렬 STL

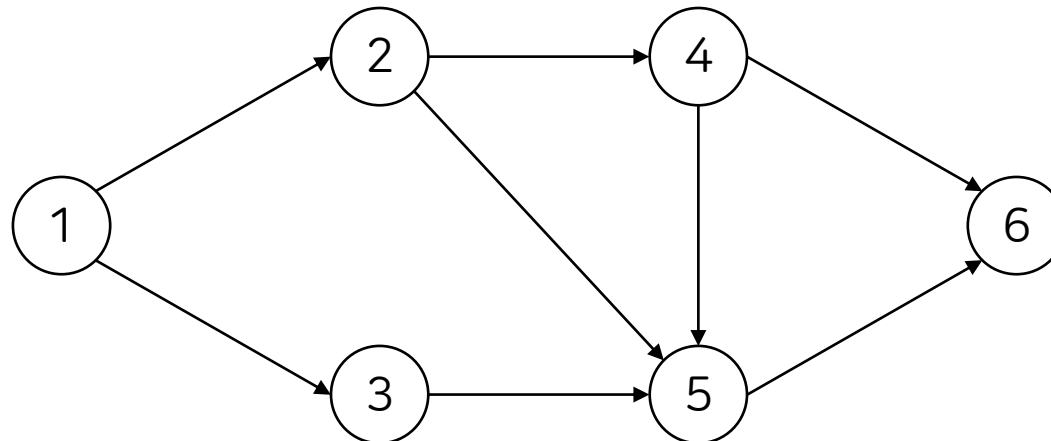
비교 기반 정렬 - 예시

- 방향 그래프의 위상 정렬

- 위상 정렬: $A \rightarrow B$ 간선이 있으면 A가 B보다 먼저 등장하도록 정점을 정렬하는 문제
- 비교 연산
 - 만약 A에서 B로 가는 경로가 존재한다면, A가 B보다 먼저 등장해야 함
 - A에서 B로 가는 경로와 B에서 A로 가는 경로 모두 없으면, 둘 중 어느 것이 먼저 등장해도 상관없음

- 예시

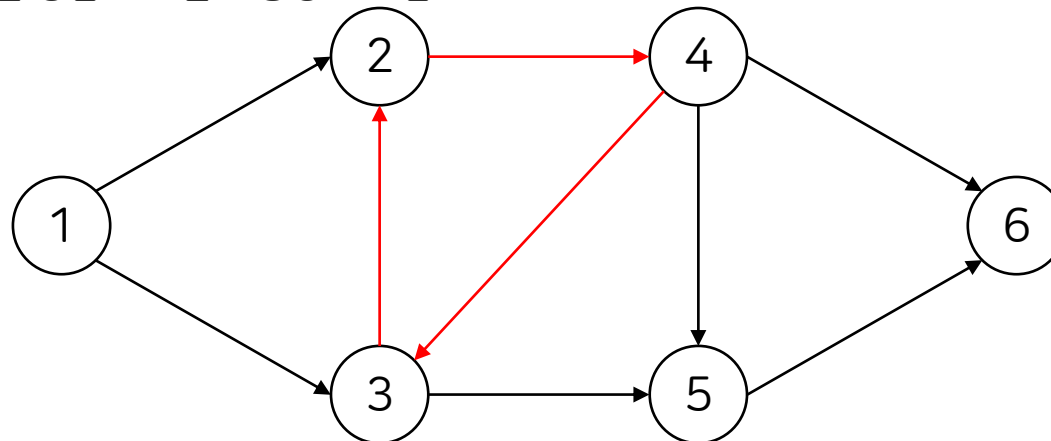
- 2번 정점에서 6번 정점으로 가는 경로 존재 $\rightarrow$ 2는 6보다 먼저 등장해야 함
- 3번 정점과 4번 정점 사이에는 서로 경로가 존재하지 않음 $\rightarrow$ 둘 중 어느 것이 먼저 등장하더라도 상관없음



정렬 STL

비교 기반 정렬 - 예시

- 방향 그래프의 위상 정렬
 - 위상 정렬: $A \rightarrow B$ 간선이 있으면 A가 B보다 먼저 등장하도록 정점을 정렬하는 문제
 - 비교 연산
 - 만약 A에서 B로 가는 경로가 존재한다면, A가 B보다 먼저 등장해야 함
 - A에서 B로 가는 경로와 B에서 A로 가는 경로 모두 없으면, 둘 중 어느 것이 먼저 등장해도 상관없음
 - 만약 사이클이 있다면...
 - 2번 정점은 4번 정점보다 먼저 등장해야 함
 - 4번 정점은 3번 정점보다 먼저 등장해야 함
 - 3번 정점은 2번 정점보다 먼저 등장해야 함
 - ...
 - ??



정렬 STL

비교 기반 정렬

- 원소를 정렬하기 위해서는 비교 연산이 어떤 조건을 만족해야 할까?
- `bool compare(a, b)`: a가 b보다 반드시 먼저 등장해야 하면 `true`, 그렇지 않으면 `false`를 반환하는 함수
 - 값이 같은 두 원소는 둘 중 어느 것이 먼저 오더라도 상관 없음
 - 즉, `compare(a, a)`는 `false`를 반환해야 함
 - 비교 결과에 사이클이 있으면 안 됨
 - 즉, `compare(a, b)`와 `compare(b, a)`가 모두 `true`면 안 됨
 - 비교 결과는 일관적이어야 함
 - 즉, `compare(a, b)`와 `compare(b, c)`가 모두 `true`라면, `compare(a, c)` 또한 `true`를 반환해야 함
 - “동등한 원소”는 정말로 동등해야 함
 - 즉, a와 b가 동등하고, b와 c가 동등하다면, a와 c 또한 동등해야 함
 - $eq(x, y) = !compare(x, y) \text{ and } !compare(y, x)$ 라고 정의하면
 - `eq(a, b)`와 `eq(b, c)`가 모두 참일 때 `eq(a, c)`가 성립해야 함

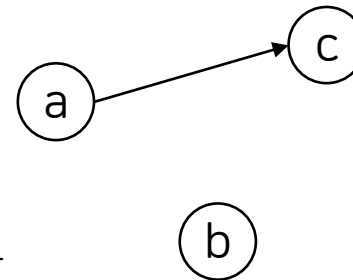
정렬 STL

비교 기반 정렬 - 비교 함수

- `bool compare(T a, T b)`
 - a가 b보다 반드시 앞에 나와야 하면 `true`, 아니면 `false`
 - `compare(a, a) = false`
 - `compare(a, b) = true` 이면 `compare(b, a) = false`
 - `compare(a, b) = compare(b, c) = true` 이면 `compare(a, c)` 또한 `true`
 - `eq(a, b) = eq(b, c) = true` 이면 `eq(a, c)` 또한 `true`
 - 단, `eq(x, y) = !compare(x, y) and !compare(y, x)`

- 예시

- 정수의 오름차순 정렬
 - `compare(a, b) := a < b`
 - $a \leq b$ 는 첫 번째 조건에 위배되므로 불가능
- 방향 그래프의 위상 정렬
 - 네 번째 조건에 위배되므로 불가능
 - 위상 정렬을 해결하기 위한 방법은 그래프 이론 시간에 배울 수 있음



- C++에는 비교 함수를 이용해 정렬하는 `std::sort`, `std::stable_sort` 라는 함수가 있음

정렬 STL

std::sort

- sort(first, last): 시작 주소, 끝 주소
 - 기본적으로 오름차순 정렬
 - 구조체/클래스의 경우 < 연산이 정의되어 있어야 함
- sort(first, last, comp): 시작 주소, 끝 주소, 비교 함수
- $O(N \log N)$

```
#include <bits/stdc++.h>
using namespace std;

bool Compare(int a, int b){
    return a > b;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int A[5] = {5, 3, 1, 4, 2};

    // 정렬할 범위의 첫 주소, 마지막 주소의 한 칸 뒤 (반열린 구간)
    sort(A, A+5); // 비교 함수 지정 안 하면 오름차순 정렬
    for(int i=0; i<5; i++) cout << A[i] << " \n"[i+1==5];

    sort(A, A+5, Compare); // 비교 함수 사용, 내림차순
    for(int i=0; i<5; i++) cout << A[i] << " \n"[i+1==5];

    // 람다식으로 비교 함수 구현, 오름차순
    sort(A, A+5, [](int a, int b){ return a < b; });
    for(int i=0; i<5; i++) cout << A[i] << " \n"[i+1==5];
}
```

정렬 STL

std::stable_sort

- stable sort와 unstable sort
 - stable sort: 동등한(비교 불가능한) 원소는 기존 순서 유지
 - unstable sort: 동등한 원소들의 순서는 마음대로 배치
- std::sort는 unstable sort
- stable sort를 사용하고 싶다면 std::stable_sort 사용

```
● ● ●

#include <bits/stdc++.h>
using namespace std;

struct Point{
    int x, y;
};

bool Compare(Point a, Point b){
    return a.x < b.x;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    Point A[3] = { {1, 3}, {1, 2}, {0, 1} };
    stable_sort(A, A+3, Compare);
    // x 좌표가 같은 점들은 기존 순서 유지
    for(int i=0; i<3; i++) cout << A[i].x << " " << A[i].y << "\n";
}
```

기수 정렬

기수 정렬

기수 정렬

- 비교 기반 정렬은 최악의 경우에 $O(n \log n)$ 보다 빠를 수 없다는 것이 증명되어 있음
- 만약 정렬을 $O(n \log n)$ 보다 빠르게 수행하고 싶다면?
 - 입력 데이터에 대한 정보(e.g., 데이터의 성질 또는 분포)를 더 갖고 있거나
 - 비교 기반 정렬이 아닌 다른 방식을 사용해야 함
- 기수 정렬은 입력 데이터가 d자리 정수일 때 $O(dn)$ 시간에 정렬하는 알고리즘

기수 정렬

기수 정렬 알고리즘

- 낮은 자리부터 차례대로 정렬하는 방식
- $A = \{329, 657, 457, 839, 436, 720, 355\}$
- 1단계. A를 일의 자리 기준으로 정렬, 일의 자리가 같다면 stable하게 정렬
 - $A = \{720, 355, 436, 657, 457, 329, 839\}$
- 2단계. A를 십의 자리 기준으로 정렬
 - $A = \{720, 329, 436, 839, 355, 657, 457\}$
 - 십의 자리가 같은 경우 기존 순서를 유지 → A의 원소는 100으로 나눈 나머지로 정렬됨
- 3단계. A를 백의 자리 기준으로 정렬
 - $A = \{329, 355, 436, 457, 657, 720, 839\}$
 - 백의 자리가 같은 경우 기존 순서를 유지 → A의 원소는 1000으로 나눈 나머지로 정렬됨
 - 끝

기수 정렬

기수 정렬 알고리즘

- 각 단계를 어떻게 해야 $O(n)$ 에 처리할 수 있을까?
- 목표: 특정 자리를 기준으로 stable sort를 $O(n)$ 시간에 처리
 - 10개의 큐를 생성
 - A의 원소를 차례대로 보면서, 각 자리에 해당하는 큐에 원소를 삽입
 - 0번 큐부터 9번 큐까지 차례대로 보면서, 큐에 있는 원소를 순서대로 가져오면 됨

```
int N, A[1010101];
void RadixSort(int d){
    queue<int> que[10];
    for(int iter=0, p=1; iter<d; iter++, p*=10){
        // iter = 0이면 A[i] / 1 % 10 = 일의 자리 기준으로 정렬
        // iter = 1이면 A[i] / 10 % 10 = 십의 자리 기준으로 정렬
        // iter = 2이면 A[i] / 100 % 10 = 백의 자리 기준으로 정렬

        for(int i=0; i<N; i++) que[A[i]/p%10].push(A[i]);
        int idx = 0;
        for(int i=0; i<10; i++){
            while(!que[i].empty()){
                A[idx++] = que[i].front();
                que[i].pop();
            }
        }
    }
}
```

기수 정렬

기수 정렬 알고리즘

- 굳이 10진법으로만 볼 필요는 없음
- 진법이 클수록 iteration 횟수 감소하지만, 너무 크면 큐를 선언하느라 메모리를 많이 쓰게 될 수 있음
- 65536진법을 사용하면 int 범위에서 2번, long long 범위에서 4번 만에 정렬할 수 있음
- 큐 대신 동적 배열을 사용해도 됨

```
int N, A[1010101];

void RadixSort_32bit(){
    constexpr int BASE = 65536;
    vector<int> vec[BASE];

    // first iteration
    for(int i=0; i<N; i++) vec[A[i] % BASE].push_back(A[i]);
    for(int i=0, idx=0; i<BASE; i++){
        copy(vec[i].begin(), vec[i].end(), A+idx);
        idx += vec[i].size();
        vec[i].clear();
    }

    // second iteration
    for(int i=0; i<N; i++) vec[A[i] / BASE].push_back(A[i]);
    for(int i=0, idx=0; i<BASE; i++){
        copy(vec[i].begin(), vec[i].end(), A+idx);
        idx += vec[i].size();
        vec[i].clear();
    }
}
```

이분 탐색

이분 탐색

업/다운 게임

- 철수는 1 이상 N 이하인 자연수 X 를 하나 선택한다.
- 영희는 철수가 생각한 X 를 맞혀야 한다.
- 영희가 어떤 수 Y 를 말하면, 철수는 아래와 같은 대답을 한다.
 - $X > Y$ 라면: Up
 - $X < Y$ 라면: Down
 - $X = Y$ 라면: Accept
- 영희의 질문 횟수를 최소화 시키자.

이분 탐색

영희의 전략

- 정답은 $[1, N]$ 사이에 있음
 - 중간 지점 $M_1 = \text{floor}((1+N)/2)$ 선택
 - 정답이 M_1 보다 작다면 구간을 $[1, M_1-1]$ 로 조정
 - 정답이 M_1 보다 크다면 구간을 $[M_1+1, N]$ 으로 조정
 - 정답이 존재하는 구간을 $[S_2, E_2]$ 라고 하자
- 정답은 $[S_2, E_2]$ 사이에 있음
 - 중간 지점 $M_2 = \text{floor}((S_2+E_2)/2)$ 를 선택
 - 정답이 M_2 보다 작다면 구간을 $[S_2, M_2-1]$ 로 조정
 - 정답이 M_2 보다 크다면 구간을 $[M_2+1, E_2]$ 로 조정
- 반복

이분 탐색

영희의 전략

- 정답이 존재할 수 있는 구간이 매번 절반으로 감소함
 - 최악의 경우에도 $\log_2 N + 1$ 번의 질문으로 답을 구할 수 있음
- 이게 최악의 경우에 질문 횟수를 최소화하는 전략일까?
 - YES
 - 상대자 논증을 이용해 증명할 수 있음

이분 탐색

정렬된 배열 A 가 주어진다. K 가 A 의 몇 번째 원소인지 구해라.

- $A = \{1, 2, 3, 4, 5, 6, 7\}, K = 3$
- $A = \{1, 2, 3, 4, 5, 6, 7\}, K = 3$
- $A = \{1, 2, 3, 4, 5, 6, 7\}, K = 3$

시간 복잡도

- 기본 연산: 배열의 원소와 어떤 수를 비교하는 것
- 기본 연산의 수행 횟수: 최악의 경우 $\log_2 N + 1$ 번
 - 매번 정답이 존재할 수 있는 구간의 크기가 절반씩 감소
- 따라서 최악의 경우 비교 횟수는 $O(\log N)$

이분 탐색

BOJ 1920. 수 찾기

- N개의 수로 구성된 배열이 주어짐
- X가 주어지면 배열에 값이 X 원소가 존재하는지 판별하는 작업을 M번해야 함
- 매번 모든 배열의 원소를 확인하면 $O(NM)$
- 배열을 정렬한 다음 매번 이분 탐색
 - 정렬 $O(N \log N)$
 - 이분 탐색 M번 $O(M \log N)$
 - 전체 시간 복잡도는 $O((N+M) \log N)$

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N, M, A[101010];

bool Find(int v){
    int l = 1, r = N;
    while(l <= r){
        int m = (l + r) / 2;
        if(v == A[m]) return true;
        else if(v < A[m]) r = m - 1;
        else l = m + 1;
    }
    return false;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    sort(A+1, A+N+1);
    cin >> M;
    for(int i=1; i<=M; i++){
        int t; cin >> t;
        cout << Find(t) << "\n";
    }
}
```

결정 문제와 최적화 문제

결정 문제와 최적화 문제

결정 문제와 최적화 문제

- 결정 문제: YES/NO로 답할 수 있는 문제
 - 가중치가 K 이하인 해가 존재하는가?
- 최적화 문제: 최댓값/최솟값을 구하는 문제
 - 가능한 최소 가중치는 얼마인가?
- 어떤 문제가 더 어려울까?
 - 최적화 문제가 결정 문제보다 쉬울 수 없음
 - 최적화 문제를 해결할 수 있으면 무조건 결정 문제를 해결할 수 있음

결정 문제와 최적화 문제

결정 문제와 최적화 문제

- 결정 문제의 예시: 배열 A의 최댓값이 3 이하인지 판별하라.
- 최적화 문제의 예시: 배열 A의 최댓값을 구해라.
- 최적화 문제를 푸는 함수 $\text{optimization}(A)$ 가 있으면
- 결정 문제를 푸는 함수 $\text{decision}(A) := \text{optimization}(A) \leq 3$

결정 문제와 최적화 문제

결정 문제의 풀이를 이용해 최적화 문제를 해결

- 배열 A의 최댓값을 구하는 최적화 문제 → 배열 A의 최댓값이 K 이하인지 판별하는 결정 문제
- $K = 0, 1, 2, \dots$ 에 대해 결정 문제를 해결하면 됨
- 이렇게 보면 비효율적인 것 같지만...

결정 문제와 최적화 문제

결정 문제의 풀이를 이용해 최적화 문제를 해결

- 배열 A의 최댓값을 구하는 최적화 문제 → 배열 A의 최댓값이 K 이하인지 판별하는 결정 문제
- $K = 0, 1, 2, \dots$ 에 대해 결정 문제를 해결하면 됨
- 이렇게 보면 비효율적인 것 같지만...
- 배열의 최댓값을 10이라고 하자
 - $K = 0, 1, 2, \dots, 9$ 일 때는 NO
 - $K = 10, 11, 12, \dots$ 일 때는 YES
 - 결정 문제의 답이 NNN...NNYYYY...YY 꼴이므로 가장 먼저 등장하는 Y의 위치를 찾으면 됨
 - 이분 탐색

결정 문제와 최적화 문제

결정 문제의 풀이를 이용해 최적화 문제를 해결

- 배열의 최댓값을 10이라고 하자
 - $K = 0, 1, 2, \dots, 9$ 일 때는 NO
 - $K = 10, 11, 12, \dots$ 일 때는 YES
 - 결정 문제의 답이 NNN...NNYYY...YY 꼴이므로 가장 먼저 등장하는 Y의 위치를 찾으면 됨
- 이분 탐색
 - 정답이 존재하는 구간 $[S, E]$ 와 중간 지점 $M = \text{floor}((S+E)/2)$ 에 대해
 - $\text{decision}(M)$ 이 참이면(최댓값이 M 이하이면) 정답은 $[S, M]$ 에 존재
 - $\text{decision}(M)$ 이 거짓이면(최댓값이 M 초과이면) 정답은 $[M+1, E]$ 에 존재

결정 문제와 최적화 문제

매개 변수 탐색 (Parametric Search)

- 조건을 만족하는 최댓값을 구하는 문제
 - 결정 문제의 답이 $YYY...YYNNN...NN$ 꼴일 때 Y 가 등장하는 마지막 위치를 구하는 문제
- 조건을 만족하는 최솟값을 구하는 문제
 - 결정 문제의 답이 $NNN...NNYYYY...YY$ 꼴일 때 Y 가 등장하는 첫 번째 위치를 구하는 문제
- 이분 탐색을 이용해 해결할 수 있음
 - 최댓값을 구하는 문제
 - 정답이 존재하는 구간 $[S, E]$ 와 중간 지점 M 에 대해
 - $\text{decision}(M)$ 이 참이면 정답은 $[M, E]$ 에 존재
 - $\text{decision}(M)$ 이 거짓이면 정답은 $[S, M-1]$ 에 존재
 - 최솟값을 구하는 문제
 - 정답이 존재하는 구간 $[S, E]$ 와 중간 지점 M 에 대해
 - $\text{decision}(M)$ 이 참이면 정답은 $[S, M]$ 에 존재
 - $\text{decision}(M)$ 이 거짓이면 정답은 $[M+1, E]$ 에 존재

결정 문제와 최적화 문제



```
int Minimization(){
    int s = 1, e = N;
    while(s < e){
        int m = (s + e) / 2; // floor((s + e) / 2)
        if(Decision(m)) e = m;
        else s = m + 1;
    }
    return e;
}

int Maximization(){
    int s = 1, e = N;
    while(s < e){
        int m = (s + e + 1) / 2; // ceil((s + e) / 2)
        if(Decision(m)) s = m;
        else e = m - 1;
    }
    return s;
}
```

결정 문제와 최적화 문제

BOJ 2805. 나무 자르기

- N개의 나무가 있고, i번째 나무의 높이는 $A[i]$ 임
- 높이가 H인 지점에서 절단기를 사용하면 높이가 H 이상인 나무에서 $A[i] - H$ 만큼 가져갈 수 있음
- 나무를 M 이상 가져가기 위해 필요한 높이의 최댓값
- $f(h) := h$ 에서 절단기를 사용했을 때 가져가게 되는 나무의 양
- $f(h) \geq M$ 을 만족하는 가장 큰 h를 구하는 문제

결정 문제와 최적화 문제

BOJ 2805. 나무 자르기

- N 개의 나무가 있고, i 번째 나무의 높이는 $A[i]$ 임
- 높이가 H 인 지점에서 절단기를 사용하면 높이가 H 이상인 나무에서 $A[i] - H$ 만큼 가져갈 수 있음
- 나무를 M 이상 가져가기 위해 필요한 높이의 최댓값
- $f(h) := h$ 에서 절단기를 사용했을 때 가져가게 되는 나무의 양
- $f(h) \geq M$ 을 만족하는 가장 큰 h 를 구하는 문제
- h 가 증가하면 $f(h)$ 는 단조 감소함
- $\text{decision}(h) := f(h) \geq M$ 으로 정의하면 결정 문제의 답은 YYY...YYNNN...NN 꼴
- Y가 등장하는 마지막 지점을 구하는 문제

결정 문제와 최적화 문제



```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N, M, A[101010];

ll Get(int h){
    ll res = 0;
    for(int i=1; i<=N; i++) res += max(0, A[i] - h);
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++) cin >> A[i];

    int l = 0, r = 1e9;
    while(l < r){
        int m = (l + r + 1) / 2;
        if(Get(m) >= M) l = m; // M만큼 가져갈 수 있으면 정답은 l 이상
        else r = m - 1; // M만큼 가져갈 수 없으면 정답은 r 미만
    }
    cout << l;
}
```

결정 문제와 최적화 문제

BOJ 15810. 풍선 공장

- N명의 스태프가 M개의 풍선을 만들어야 함
- i번째 스태프는 A[i]분마다 풍선을 만들 수 있음
- M개의 풍선이 모두 만들어지는 데 걸리는 최소 시간을 구하는 문제
- $f(m) := m$ 분 동안 만들 수 있는 풍선의 최대 개수 = $\text{sum}(\text{floor}(m / A[i]))$
- $f(m) \geq M$ 을 만족하는 가장 작은 m을 구하는 문제
- m이 증가하면 $f(m)$ 은 단조 증가함
- $\text{decision}(m) := f(m) \geq M$ 으로 정의하면 결정 문제의 답은 NNN...NNYYYY..YY 꼴
- Y가 등장하는 첫 번째 지점을 구하는 문제

결정 문제와 최적화 문제



```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N, M, A[1010101];

ll Get(ll m){
    ll res = 0;
    for(int i=1; i<=N; i++) res += m / A[i];
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++) cin >> A[i];

    ll l = 0, r = 1e12;
    while(l < r){
        ll m = (l + r) / 2;
        if(Get(m) >= M) r = m;
        else l = m + 1;
    }
    cout << r;
}
```

결정 문제와 최적화 문제

BOJ 2417. 정수 제곱근

- N이 주어지면 $\text{ceil}(\sqrt{N})$ 을 구하는 문제



```
#include <bits/stdc++.h>
using namespace std;
using ull = unsigned long long;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    ull N; cin >> N;
    ull l = 0, r = (1LL << 32) - 1;
    while(l < r){
        ull m = (l + r) / 2;
        if(m * m >= N) r = m;
        else l = m + 1;
    }
    cout << l;
}
```

요약

- 여러가지 정렬 알고리즘
 - 버블 정렬(거품 정렬): 인접한 두 원소를 차례로 비교
 - 선택 정렬(교환 정렬): 최솟값(혹은 최댓값)을 찾고 위치를 교환
 - 삽입 정렬: 정렬된 부분의 위치에 삽입함
 - 퀵 정렬: 기준 원소로 나눈 후 기준의 위치를 잡고 나머지를 정렬
 - 합병 정렬: 무조건 나눈 후 정렬된 부분열을 합병하여 정렬
 - 기수 정렬: 자릿수의 역순으로 정렬
- 정렬 STL
 - 동등성을 별도의 함수로 정의함(추상화)
 - 정렬 STL의 예: 위상 정렬(선후를 정할 수 없는 두 원소가 있음, 부분 순서)
- 이분 탐색
 - 탐색 공간을 반씩 줄여 나가며 탐색하는 방법
 - 결정 문제와 최적화 문제: 결정 문제로 최적화 문제를 푸는 경우 이분 탐색 기법을 사용할 수 있음