#04-2. 정렬 알고리즘

나정휘

https://justiceHui.github.io/

목차

버블 정렬

버블 정렬 알고리즘

- 서로 인접한 두 원소를 비교해서 정렬하는 알고리즘
  - 5 3 4 2 1 (5 > 3 이므로 교환)
  - 3 5 4 2 1 (5 > 4 이므로 교환)
  - 3 4 5 2 1 (5 > 2 이므로 교환)
  - 3 4 2 5 1 (5 > 1 이므로 교환)
  - 3 4 2 1 5 (첫 번째 순회 끝)

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

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

  - 2 1 3 4 5 (2 > 1 이므로 교환)
  - 1 2 3 4 5 (2 < 3 이므로 교환 X)
  - 1 2 3 4 5 (3 < 4 이므로 교환 X)
  - 1 2 3 4 5 (4 < 5 이므로 교환 X)
  - 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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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";
}
```

질문?

시간 복잡도 계산

시간 복잡도

- 최악의 경우, 최선의 경우, 평균의 경우로 나눠서 분석
  - 사실 문제 풀 때는 최악의 경우만 생각해도 됨

- 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-1번, 교환은 최대 N-1번
  - 이 과정을 N-1번 반복하므로 $O(N^2)$
- 선택 정렬
  - i번째 단계에서 i..N번째 원소 중 최솟값을 찾을 때 비교는 정확히 N-i번, 이후 교환 1번
  - 이 과정을 i=1..N-1일 때 수행하므로 $O(N^2)$
- 삽입 정렬
  - i번째 단계에서 i번째 원소는 최대 i-1번 이동 가능, 따라서 비교와 교환 모두 최대 i-1번
  - 이 과정을 i=2..N일 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-1개의 단계를 거쳐야 하므로 최선의 경우에도 $O(N^2)$
- 삽입 정렬
  - 삽입 정렬의 교환 횟수는 inversion의 개수와 동일함
    - 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 = $1 \le x < y \le N$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 $O(N^2)$

질문?

정당성 증명

반복문 불변식 (loop invariant)

- 수학적 귀납법을 이용해 알고리즘의 정당성을 증명하는 방법 중 하나
- 반복문이 진행됨에 따라 정답에 가까워져 가고, 반복문이 종료되면 정답을 구한다는 것을 증명

- 증명 과정
  - 반복문에 진입하는 시점에 불변식이 성립함을 증명
  - 반복문 내부가 불변식을 깨뜨리지 않는다는 것을 증명
    - 반복문 블록이 시작할 때 불변식이 성립한다면 블록이 종료되었을 때도 불변식이 성립함을 증명
  - 반복문이 종료되었을 때 불변식이 성립하면 올바른 정답을 구한다는 것을 증명

정당성 증명

반복문 불변식 (loop invariant)

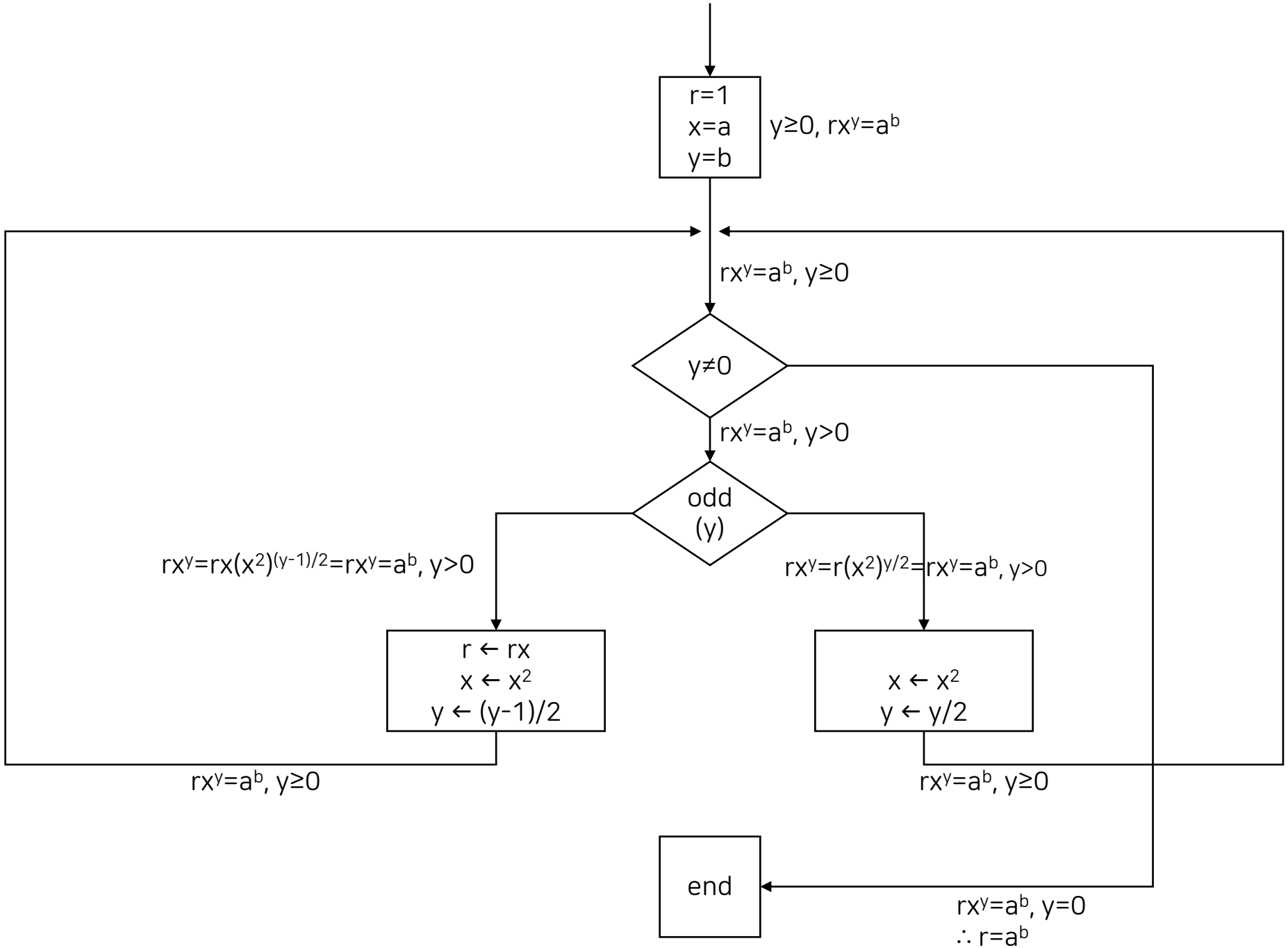
- 수학적 귀납법을 이용해 알고리즘의 정당성을 증명하는 방법 중 하나
- 반복문이 진행됨에 따라 정답에 가까워져 가고, 반복문이 종료되면 정답을 구한다는 것을 증명

- 증명 과정
  - 반복문에 진입하는 시점에 불변식이 성립함을 증명
  - 반복문 내부가 불변식을 깨뜨리지 않는다는 것을 증명
    - 반복문 블록이 시작할 때 불변식이 성립한다면 블록이 종료되었을 때도 불변식이 성립함을 증명
  - 반복문이 종료되었을 때 불변식이 성립하면 올바른 정답을 구한다는 것을 증명

정당성 증명

반복문 불변식 (loop invariant)

```
ll Pow(ll a, ll b){
    ll res = 1, x = a, y = b;
    // res * pow(x, y) = pow(a, b)
    while(y > 0){
        if(y % 2 == 0){
            // res * pow(x, y) = res * pow(x*x, y/2)
            x *= x; y /= 2;
            // res * pow(x, y) = pow(a, b)
        }
        else{
            // res * pow(x, y) = (res * x) * pow(x*x, (y-1)/2)
            res *= x;
            x *= x; y /= 2;
            // res * pow(x, y) = pow(a, b)
        }
        // 따라서 y % 2에 관계 없이 res * pow(x, y) = pow(a, b) 유지
    }
    // res * pow(x, y) = pow(a, b)
    // y = 0이므로 pow(x, y) = 1
    // 따라서 res * 1 = res = pow(a, b)
    return res;
}
```


r=1
x=a
y=b
y≥0, rx^y=a^b
rx^y=a^b, y≥0
y≠0
rx^y=a^b, y>0
odd
(y)
rx^y=rx(x^2)^(y-1)/2=rx^y=a^b, y>0
rx^y=r(x^2)^y/2=rx^y=a^b, y>0
r ← rx
x ← x^2
y ← (y-1)/2
x ← x^2
y ← y/2
rx^y=a^b, y≥0
rx^y=a^b, y≥0
end
rx^y=a^b, y=0
∴ r=a^b

정당성 증명

삽입 정렬 알고리즘의 정당성 증명

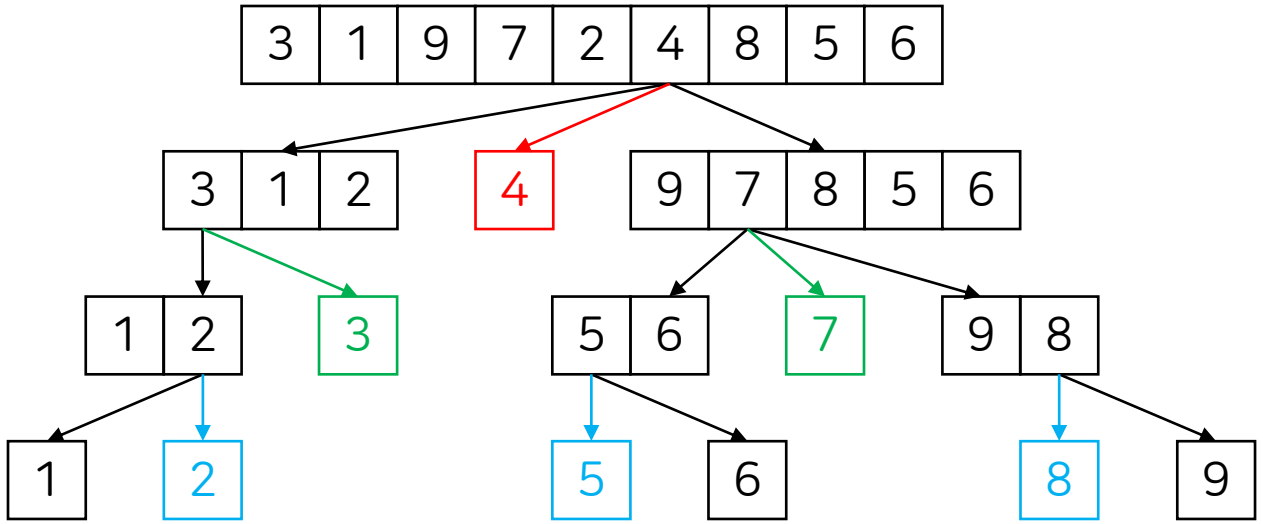
```
int N, A[1010];
void InsertionSort(){
    for(int i=2; i<=N; i++){
        // 불변식 i-1. A[1..i-1]은 이미 정렬된 상태
        for(int j=i; j>1; j--){
            // 불변식 j-1. A[j+1..i]은 정렬된 상태
            // 불변식 j-2. A[1..i]는 A[j]를 제외하면 정렬된 상태
            if(A[j-1] > A[j]) swap(A[j-1], A[j]);
            else break;
        }
    }
}
```

질문?

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

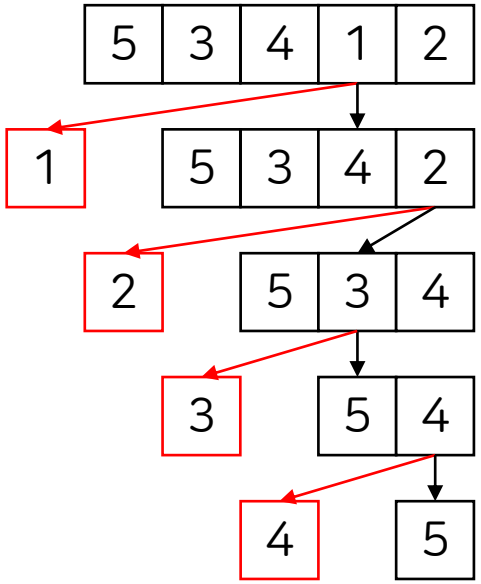


퀵 정렬

퀵 정렬 시간 복잡도

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

- 최악의 경우
  - 매번 0 : 1 : n-1로 분할되는 경우
  - T(n) = T(n-1) + c*n
  - T(n) = $O(n^2)$



퀵 정렬

퀵 정렬 시간 복잡도

- 평균의 경우
  - pivot이 i번째 데이터이면, 분할 후 왼쪽에 i-1개, 오른쪽에 n-i개의 데이터가 있음
    - T(n) = T(i-1) + T(n-i) + c*n
    - i = 0..n-1 일 때의 합을 n으로 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 n을 곱하면 $nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2$
  - n대신 $n-1$을 대입하면 $(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2$
  - $nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$
    - c는 상수이므로 n이 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 $n, n-1, n-2, \cdots, 2$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 $\frac{T(n)}{n+1} = O(\log n), T(n) = O(n\log n)$

질문?

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)
합병 정렬	O(n log n)	O(n log n)	O(n log n)	O(n)
퀵 정렬	O(n log n)	O(n log n)	$O(n^2)$	O(log 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 n을 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

질문?