

안녕하세요, 동계 대학생 S/W 알고리즘 특강의 열한 번째 시간인 오늘은 분할정복에 대해 다루어보도록 하겠습니다.

1. 기초 강의

동영상 강의 콘텐츠 확인 > 9. 분할정복

Link:

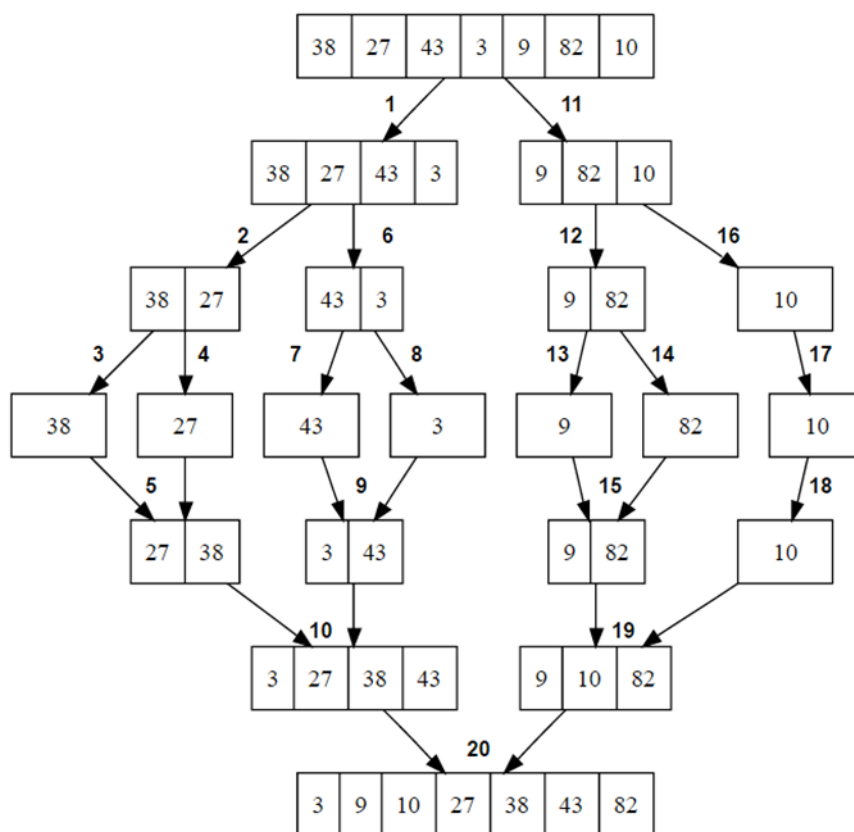
https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXyLsaRGwDFARs

※ 출석은 강의 수강 내역으로 확인합니다.

2. 실전 강의

2.1 Merge Sort

분할 정복 개념을 사용한 대표적인 정렬 알고리즘이며, 다음 순서에 따라 진행됩니다.



1. (분할) 정렬되지 않은 리스트를 절반으로 잘라 두 개의 리스트로 나눈다.
2. (정복) 생성된 두개의 리스트를 Merge Sort 알고리즘을 재귀 호출하여 정렬한다.
3. (통합) 정렬된 두개의 리스트를 다시 하나의 정렬된 리스트로 Merge 한다.

Merge Sort 는 항상 $O(N \cdot \log N)$ 의 시간 복잡도를 보장하기 때문에 안정적인 수행속도가 보장되어야 할 때 많이 사용되며, 같은 값끼리 정렬 전후의 순서가 유지되는 Stable Sort 입니다. 하지만 정렬된 두 배열을 합치기 위해 배열의 크기만큼 메모리 공간을 추가로 사용한다는 단점이 있습니다. (추가 공간을 사용하지 않는 In-Place Merge Sort 도 존재합니다! In-Place Merge Sort 는 연결 리스트(Linked List)를 사용해서 구현 합니다.)

```
#include <cstring>
```

```
constexpr size_t MAX_N = 100000;
```

```
int a[MAX_N], buffer[MAX_N];
```

```
void merge(int* const begin, int* const mid, int* const end) {
    int *begin1 = begin, *end1 = mid;
    int *begin2 = mid, *end2 = end;
    int* result = buffer;
    while (begin1 != end1 && begin2 != end2) {
        *result++ = *begin1 <= *begin2 ? *begin1++ : *begin2++;
    }
    while (begin1 != end1) *result++ = *begin1++;
    while (begin2 != end2) *result++ = *begin2++;
    std::memcpy(begin, buffer, sizeof(int) * (end - begin));
}
```

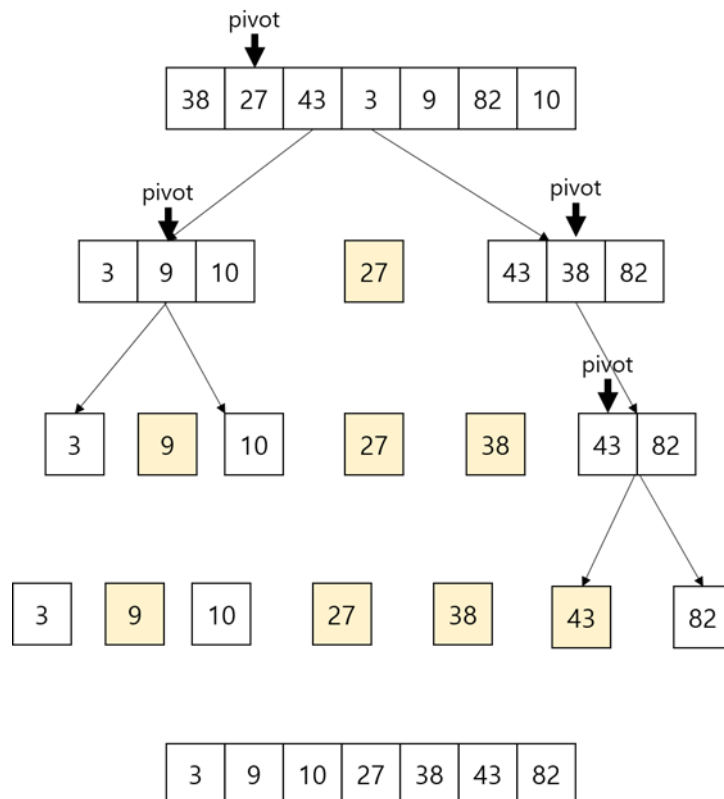
```
void merge_sort(int* const begin, int* const end) {
    if (end - begin <= 1) return;
    int* const mid = begin + (end - begin) / 2;
    merge_sort(begin, mid);
    merge_sort(mid, end);
    merge(begin, mid, end);
}
```

2.2 Quick Sort

Quick Sort 는 Merge Sort 와 함께 분할 정복의 대표적인 정렬 알고리즘 이며, 다음 순서에 따라 진행됩니다.

1. 리스트 내에 임의의 원소를 선택하여 pivot 이라 명합니다.
2. pivot 을 기준으로 pivot 보다 작은 원소는 좌측으로, pivot 보다 큰 원소는 우측으로 이동합니다.

3. pivot 을 제외하고 좌측 리스트와 우측 리스트를 Quick Sort 알고리즘을 재귀 호출하여 정렬합니다.



pivot 으로 선택된 원소는 반드시 정렬되며 다음 재귀 호출에서 제외되기 때문에 어떻게 pivot 을 선택하더라도 올바르게 정렬된다는 것은 보장됩니다. Quick Sort 는 추가 공간을 사용하지 않기 때문에 In-Place Sort 입니다.

Quick Sort 는 Merge Sort 와 다르게 pivot 이 어떻게 선택 되느냐에 따라 분할 횟수가 달라집니다. 평균적으로는 $O(N \cdot \log N)$ 의 시간복잡도를 가지지만, pivot 으로 리스트 내에서 가장 작은/큰 값을 계속 고른다면 최악의 경우 $O(N^2)$ 의 시간 복잡도를 가집니다.

pivot 만 잘 고른다면 Quick Sort 는 현대 컴퓨터에서 가장 빠른 정렬 방법 중 하나입니다. 따라서 많은 프로그래밍 언어의 표준 라이브러리의 정렬 함수는 Quick Sort 의 높은 성능은 가져오는 대신 최악의 경우를 피하기 위해, Quick Sort 와 기타 정렬 방법을 섞은 Hybrid Sort 를 사용합니다. 예를 들어 g++의 `std::sort` 는 Quick Sort 를 사용하다가 재귀가 깊어지면 Heap Sort 를 사용하고, 배열의 길이가 충분히 짧다면 Insertion Sort 를 사용하는 Intro Sort 방식을 사용합니다.

아래 구현은 랜덤으로 pivot 을 잡는 Quick Sort 입니다.

```
#include <algorithm>
#include <random>
```

```
constexpr size_t MAX_N = 1e6;

int a[MAX_N];
std::mt19937 engine(std::random_device {}());
std::uniform_int_distribution<ptrdiff_t> generator(0, PTRDIFF_MAX);

int* partition(int* begin, int* end) {
    std::iter_swap(begin + generator(engine) % (end - begin), end - 1);
    const int pivot = *--end;
    int* i = begin;
    while (begin != end) {
        if (*begin < pivot) {
            std::iter_swap(i++, begin);
        }
        ++begin;
    }
    return i;
}

void quick_sort(int* const begin, int* const end) {
    if (end - begin <= 1) return;
    int* const mid = partition(begin, end);
    quick_sort(begin, mid);
    quick_sort(mid, end);
}
```

2.3 거듭제곱 계산

X^{1024} 을 어떻게 계산할 수 있을까요. 가장 간단한 방법은 $X \cdot X \cdot X \dots X$ 이렇게 1024 번 곱하는 것입니다. 하지만 $X^{1024} = X^{512} \cdot X^{512}$ 임을 이용하면, X 를 512 번 곱해서 X^{512} 를 구한 다음에 그 둘을 곱하는 방법으로 X^{1024} 를 구할 수 있습니다. 계산이 절반으로 줄어듭니다. 마찬가지로 X^{512} 도 같은 방법으로 계산을 계속 절반으로 줄여나갈 수 있습니다.

이러한 분할정복 방식으로 X^p 을 $O(\log P)$ 시간에 구할 수 있습니다.

$X^p \% 2^{64}$ 을 구하는 코드입니다.

```
unsigned long long power(unsigned long long x, unsigned long long p) {
    if (p == 0) return 1;
```

```

    unsigned long long res = power(x, p / 2);
    res *= res;
    if (p & 1) res *= x; // p 가 홀수면 x 를 따로 한 번 곱해준다.
    return res;
}

```

아래 코드는 비재귀로 구현한 것입니다.

```

unsigned long long power(unsigned long long x, unsigned long long p) {
    unsigned long long res = 1;
    while (p) {
        if (p & 1) res *= x;
        x *= x;
        p >>= 1;
    }
    return res;
}

```

3. 기본 문제

- 염라대왕의 이름정렬
- 사탕 분배