Optimized C++
Chapter 5 - Algorithms

Time Cost of Algorithms

- 정의 : 알고리즘을 연산하는 데 필요한 비용. 주로 Big-O 표기법을 사용한다.

- Big-O 표기법
  - 알고리즘의 실행 시간을 표기하는 방법
  - 다음과 같이 알고리즘을 분류한다.
    - O(1), 상수 시간 – 실행 시간이 상수일 때 표기한다. 입력 값의 개수에 영향을 받지 않는다.
    - $O(\log_2 n)$ – 실행 시간이 입력 크기의 로그에 비례해서 늘어날 때 표기한다.
    - O(n), 선형 시간 – 실행 시간이 입력 크기에 비례해서 늘어날 때 표기한다.
    - $O(n log n)$ – 초선형 알고리즘(superlinear algorithm)이라 부르며 속도는 선형 알고리즘과 다항식 알고리즘의 중간쯤 된다.
    - $O(n^c)$ – 입력 크기가 늘어나면 실행 시간이 빠르게 늘어나며, 다항식 알고리즘(polynomial alogorithm)이라고 부른다.
    - $O(c^n)$ – 다항식 알고리즘보다도 실행 속도가 빠르게 느려지며, 지수 알고리즘이라고 부른다.
    - O(n!) – 가장 느린 알고리즘으로 개쓰레기. 팩토리얼 알고리즘이라 부른다.

Best-Case, Average And Worst-Case Time Cost

- 최선, 평균, 최악 케이스의 실행 시간도 생각하면서 알고리즘을 짜야한다. 왜냐하면 정렬이 되지 않은 데이터에 대한 최악의 케이스에는 성능이 정말 나 쁘지만 이미 정렬된 데이터가 들어왔을 때는 적용하기에 매우 좋은 것도 있기 때문이다.

- 최대값 구하는 알고리즘 두 개를 살펴보자.
  ① 배열의 모든 원소를 하나씩 확인하면서 가장 큰 수를 계속 기록한 다음, 확인이 끝나고 나면 그 값을 반환하는 방법.

  ② 각 값을 다른 모든 값과 비교하는 방법.

Best-Case, Average And Worst-Case Time Cost

- 최대값 구하는 알고리즘 두 개를 살펴보자.
  ① 배열의 모든 원소를 하나씩 확인하면서 가장 큰 수를 계속 기록한 다음, 확인이 끝나고 나면 그 값을 반환하는 방법.

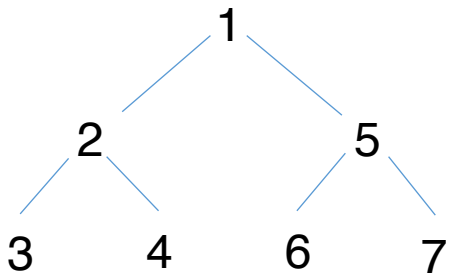
  ② 각 값을 다른 모든 값과 비교하는 방법.

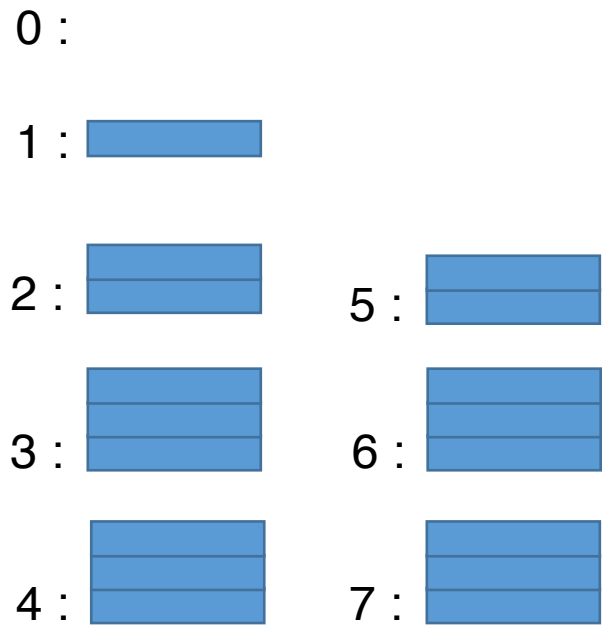
```
int CompareToMax(int* arr, int n)
{
    int curMax, i;

    if (n <= 0)
        return -1;

    curMax = arr[0];

    for (i = 1; i < n; i++)
    {
        if (arr[i] > curMax)
            curMax = arr[i];
    }
    return curMax;
}
```

```
int CompareToAll(int* arr, int n)
{
    int i, j;
    bool isMax;

    for (i = n - 1; i > 0; i--)
    {
        isMax = true;
        for (j = 0; j < n; n++)
        {
            if (arr[j] > arr[i])
            {
                isMax = false;
            }
        }

        if (isMax) break;
    }

    return arr[i];

}
```

Other Costs

-
  - 하드웨어가 제한적일 경우 데이터 저장 비용도 고려해야 한다.
  - 이진 트리의 순회를 재귀 함수로 구현하면 실행 시간은 O(n)이지만, 저장 비용은 $\log_2 n$ 만큼 필요함.





Other costs

- 병렬화 되었을 때 빠른 알고리즘들은 프로세서의 수가 중요하다.

- 따라서 코어가 한정된 일반적인 CPU가 아닌 다른 장치(예 : GPU)를 이용한다.

Time Cost of Searching Algorithms

-
  - 선형 검색의 비용은 O(n) 으로 비싸긴 하지만 일반적인 속도이다.
  - 이진 검색의 비용은$O(\log_2 n)$으로 빠른 편에 속한다.
  - 보간 검색은 $O(loglogn)$으로 이진 검색보다 개선된 속도를 자랑한다.
    - 보간 검색 – 이진 검색의 업그레이드 버전으로 데이터의 양과 찾고자 하는 데이터 위치의 비율을 이용하여 검색하는 방법.
  - 해쉬 테이블의 검색은 O(1)로 가장 빠른 속도를 자랑한다. 링크드 리스트로 해쉬 테이블을 만들면 길이가 고정되지 않는 대신 검색 속도가 최악의 경우 O(n)이 된다. 하지만 길이가 고정된 배열로 만들면 검색 속도가 O(1)이 된다.

All Search Are Equal n Is Small

- 테이블 개수가 1 일 때 모든 알고리즘의 검색 속도는 동일하다.

- 그러나 검색 수가 늘어나면 늘어날수록 속도 차이는 급격하게 벌어진다.

table size	Linear	Binary	Hash
1	1	1	1
2	1	2	1
4	2	3	1
8	4	4	1
16	8	5	1
24	13	6	1
32	16	6	1

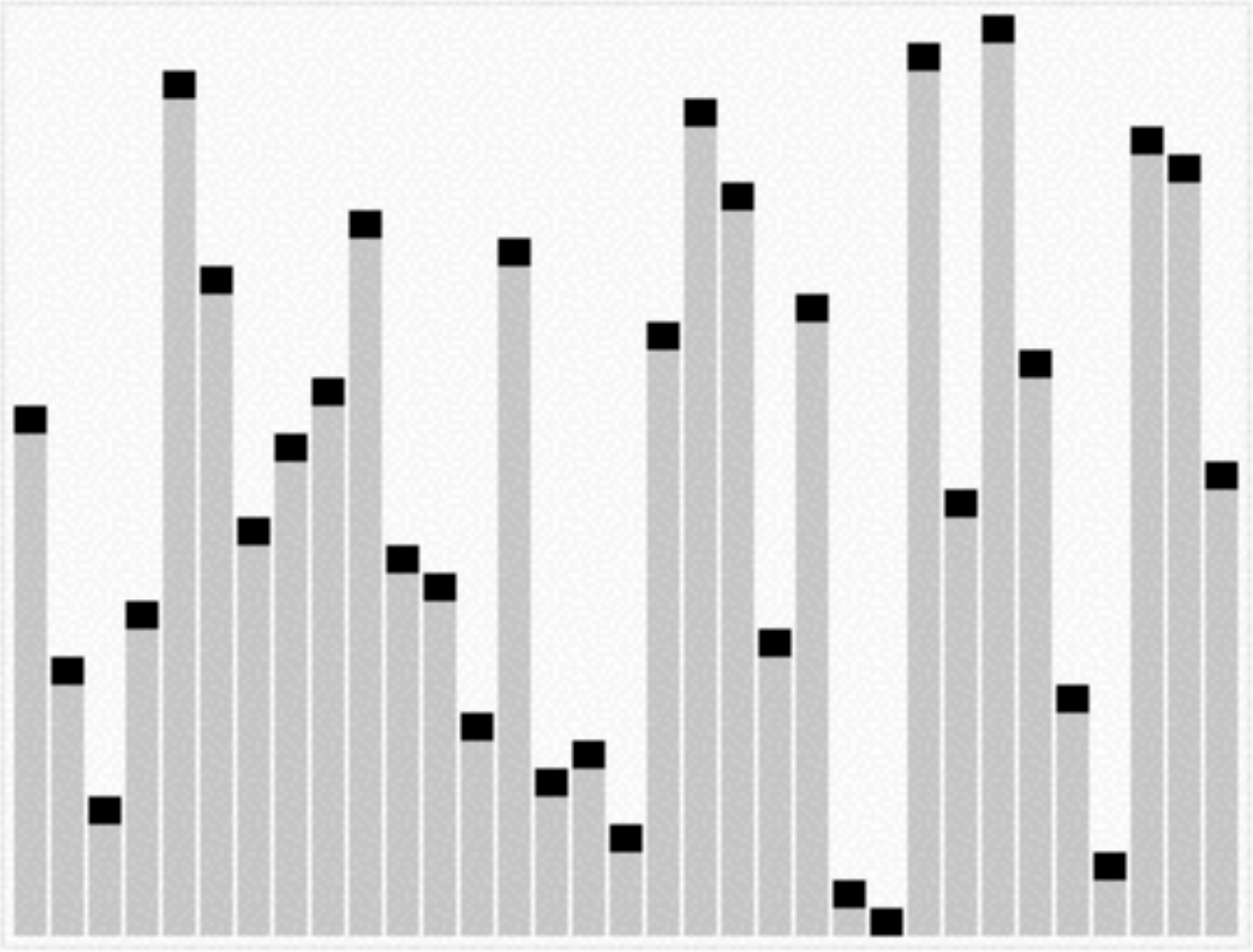
삽입정렬

- k 번째 원소를 1부터 k-1까지 비교해 적절한 위치에 넣고 하나씩 밀어내는 방식

6 5 3 1 8 7 2 4

퀵 정렬

- 하나의 피벗을 잡아 그것보다 작은건 앞으로 빼고, 피벗보다 작은것 큰것으로 나누어 다시 각각 피벗을 잡아 정렬하여 각각의 크기가 0이나 1이 될때까지 반복



병합정렬

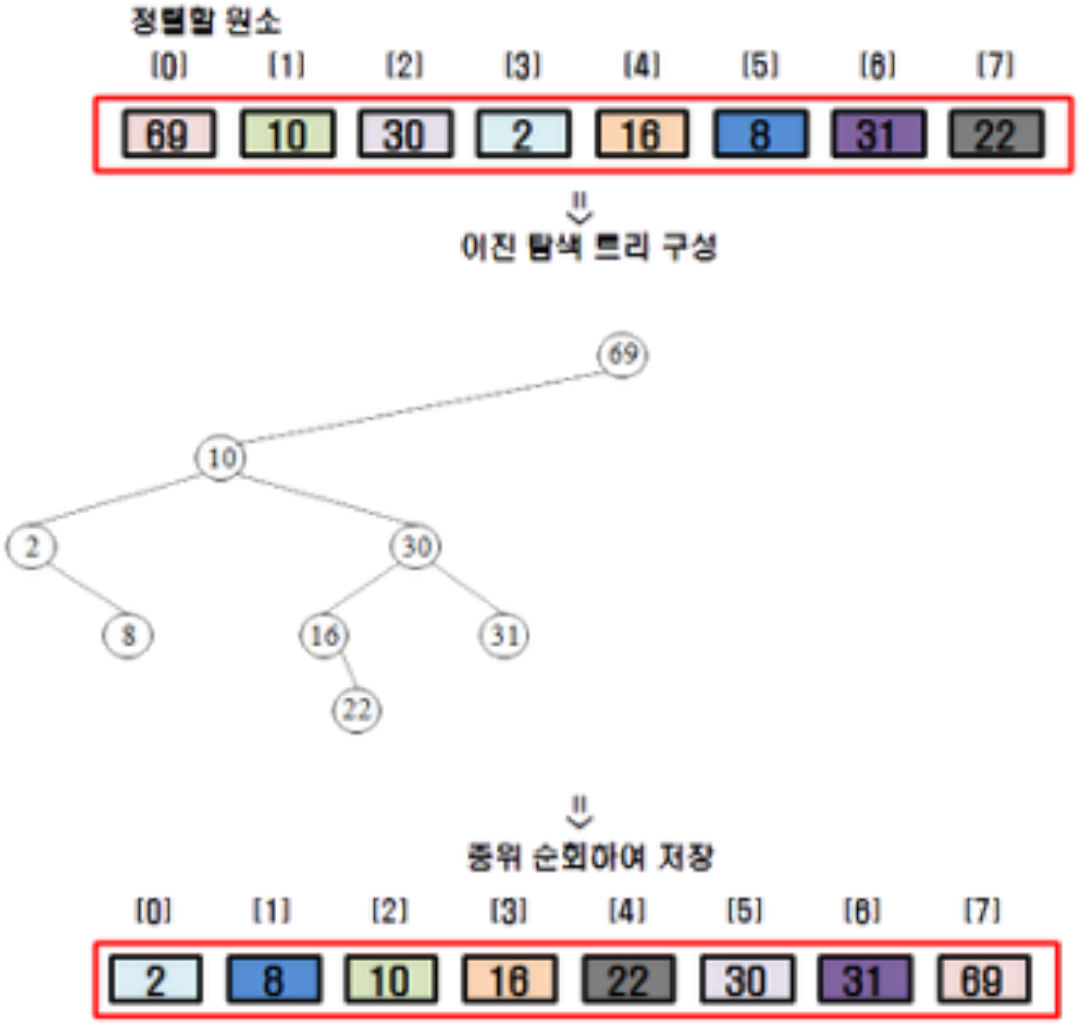
- 원소의 개수가 1 또는 0 이 될때까지 두부분으로 자른뒤 자른순서의 역순으로 비교

6 5 3 1 8 7 2 4

트리정렬

- 정렬할 원소들을 이진트리로 구성하고 중위 순회 방법을 이용하여 꺼낸다.

정렬과정(오름차순 기준)



효율적인 정렬 알고리즘

- 기수정렬
- flash sort
- 언제나 quiksort가 빠른건 아니다.

정렬알고리즘의 시간 비용

- 대부분 평균 성능이 동일 하지만 가장 좋은 케이스와 최악의 성능 그들이 소비하는 공간의 차이가 있다.

종류	최상	평균	최하	소비 공간
삽입 정렬	n	n^2	n^2	1
퀵 정렬	n log n	n log n	n^2	log n
병합 정렬	n log n	n log n	n log n	1
트리 정렬	n log n	n log n	n log n	n
힙 정렬	n log n	n log n	n log n	1
Timsort	n	n log n	n log n	n
Introsort	n log n	n log n	n log n	1

최악의 경우를 가진 정렬 알고리즘을 바꿔라

- Quicksort
- 첫번째 또는 마지막 요소를 피벗으로 지정할 경우 성능 불량
- 이미 정렬된 배열이나 거의 정렬된 배열을 정렬할 경우 성능 불량

- 정교한 Quicksort 구현
- 무작위로 피벗을 선택
- 추가로 많은 사이클을 소비 하여 중앙값을 계산하고이를 초기 피벗으로 사용하기

- 입력 데이터를 정확히 알기
- 모른다면 병렬정렬, 트리정렬, 힙정렬 사용 해도 큰 성능 불량이 없음

알고있는 입력데이터의 특성을 이용해라

- 정렬이 되있거나 거의 정렬이 되있을때는 순차정렬이 우수한 성능을 가진다.

- Trimsort는 평균적인 상황일때도 뛰어나다.(python의 표준 정렬방식)

- Introsort는 quicksort와 heapsort의 혼합
- heapsort로 최악의 경우를 보장하고, quicksort 효율적인 구현을 이용하여 평균 실행시간 감소

최적화 패턴

Precomputation(1)

- **정의** : 핫 스팟에 도달하기 전에 연산을 수행

- Precomputation은 계산할 값이 컨텍스트에 종속되지 않는 범위까지만 가능하다.
  -> int sec_per_day = 60 * 60 * 24;

- int sec_per_weekend = (date_end - date_beginning + 1) * 60 *60 *24;
  -> 2로 교체

Precomputation(2)

- c++ 컴파일러는 컴파일러의 내장형 규칙과 연산자 우선순위를 사용하여 상수 값을 자동으로 부호화

- 특정 인수가 있는 템플릿 함수 호출은 컴파일 시간에 따라 평가된다. 컴파일러는 인수가 상수 일 때 효율적인 코드 생성

Lazy Computation (1)

- **정의** : 계산의 결과값이 필요할 때까지 늦추는 기법

- **Two-part computation**

  → 오브젝트를 생성할 때 초기화를 위해 생성자에 다 코드를 넣기보다는
  생성자에는 오브젝트를 만들기 위한 최소한에 코드만 넣고
  오브젝트 초기화 함수를 따로 두어서 초기화가 필요할 때 호출함.

- **Copy-on-write(COW) = Implicit Sharing = Shadowing**

  → 어떤 다이나믹 오브젝트를 복사할 때, 데이터 자체를 복사하지 않고
  레퍼런스를 이용함. 만약 수정이 필요하다면 새로운 오브젝트를
  만들어서 복사함.
  (수정이 일어나기 전까지 복사를 하지 않음)

Lazy Computation (2)

- **Example - Copy-on-write(COW)**

```
std::string x("Hello");
std::string y = x;  // x and y use the same buffer
y += ", World!";            // now y uses a different buffer // x still uses the same old buffer
```

  - *std::string y = x*
    → y는 레퍼런스를 이용하여 x와 **같은 버퍼**를 공유

  - *y += “, World”*
    → y의 문자열이 추가됨으로써 **새로운 버퍼**가 만들어지고
    기존에 가지고 있던 “Hello”를 버퍼에 복사하고 “, World!”를 추가

Batching

- **정의 :** 같은 작업을 하는 데이터끼리 모아서 처리

- **Buffred Output**
  → 버퍼가 꽉 차거나 라인의 끝 혹은 파일의 끝을 만날 때까지
  버퍼에 입력 데이터를 계속 쌓음.
- **Converting an unsorted array into a heap**
  → 효율적인 알고리즘을 위해 정렬되지 않은 배열을
  힙으로 바꾸는 것도 Batching의 방법.
- **Multithreaded Task Queue**
  → 리소스를 효율적으로 사용하기 위해 멀티쓰레드에서
  Queue를 이용하여 작업 - EX) Input/Output Completion Port

Caching (1)

- **정의 :** 계산의 결과를 재사용하는 기법

- **Complier Caching**
  → 컴파일러는 반복되어서 나오는 결과를 캐싱함.
- **Cache Memory**
  → 메모리와 프로세서에 접근 속도를 높이기 위해 중간에
  캐시 메모리를 넣음.
- **std::string**
  → C-스타일 문자열과는 다르게 std::string은 문자열 길이를
  필요할 때 마다 매번 계산하지 않고 캐싱함.
- **Thread Pool**
  → 미리 쓰레드를 여러 개 생성하여 쓰레드 생성 비용을 없앰.
- **Dynamic Programming**
  → 계산 결과를 캐싱함으로써 재귀 관계를 가진 계산의 속도를 올려줌.

Caching (2)

- **Example – Caching**

- **Complier Caching**

```
a[i][j] = a[i][j] + c;

auto p = &a[i][j];
*p = *p + c;
```

a[i][j]를 캐싱

- **Dynamic Programming**

```
int fibonacci(int n)
{
    if(n==1 || n==2) return 1;
    else return fibonacci(n-1) + fibonacci(
n-2);
}
```

→

```
int fibonacci(int n)
{
    int sum;
int n1 = 1; int n2 = 1;
    if(n==1 || n==2) return 1;
    else{
        for(int i=2; i<n; i++){
            sum = n1 + n2;
            n1 = n2;
            n2 = sum;
        }
        return sum;
    }
}
```

→ 피보나치 수열을 Dynamic Programming으로 변환한 모습

Specialization

- **정의** : 특별한 상황에서 요구되지 않는 비싼 계산 부분들을 제거하는 방법

- **std::swap()** (C++11 이전)

→ std::swap함수는 인자를 복사하여 구현되었지만
C++11부터 **move semantics**를 이용하여 더 효율적으로 바뀜.

- **std::string()**

→ std::string()은 문자열의 길이를 동적으로 변화시킬 수 있지만 고정된 길이에
문자열을 사용하는 상황이면 std::string()보다는 C스타일의 문자열을
사용하는게 성능이 더 좋음.

Taking Bigger Bites

- **ㅎ정의** : 반복 오버헤드를 줄이기 위해 데이터를 더 많이 받는 것

- **사용자 ←→ 운영체제**

  → 사용자와 OS 간에 데이터를 주고 받을 때 더 큰 데이터를 주고 받음으로써
  시스템 콜 오버헤드를 줄일 수 있음.

- **Move or Clear Buffers**

  → Buffer를 지우거나 이동시킬 때 바이트 단위가 아닌
  더 큰 단위(word or longword)로 처리함으로써 성능을 향상시킬 수 있음.

- **Compare strings**

  → 빅엔디안 머신에서 문자열 비교를 **더 큰 단로(word or long word)**하면 성능이 올라감.
  (리틀엔디안 머신에서는 이 방식은 위험)

- **Thread**

  → 쓰레드를 깨웠을 때 **더 많은 작업을 수행시켜서** 반복되는 '쓰레드 깨우는 오버헤드'를 절약할 수 있음.

- **A maintenance task**

  → 루프를 돌면서 매번 지속적으로 수행하는 일들을 10단위 혹은 100단위 때 마다 처리하게 해서 횟수를 줄임.

Hinting

- **정의** : 힌트를 이용하여 연산량을 줄이는 기법

- **std::map**

→ 삽입하는 방식을 **emplace_hint**와 같은 함수를 이용하여
삽입 속도를 빠르게 할 수 있음.

Optimizing the Expected Path

- **정의** : 예상되는 경로를 최적화하는 것

- **If- else if – else**

→ 무작위로 순서로 테스트할 때 if문을 지나칠 때 마다 절반의 테스트가 계산되어지는데 이때 시간을 많이 잡아먹는 경우를 찾게 되면 그 테스트를 먼저 수행함.

Hashing

- **정의** : 어떤 데이터를 빨리 찾을 수 있도록 직접 접근할 수 있는
  짧은 길이의 값이나 키로 변환하는 것.

- **Comparsion for equality**

  → 두 입력 값의 해시 값을 비교하는 알고리즘을 최적화 해야함.

  → 최적화를 위해 **Double-Checking** 사용

  → 일반적으로 입력 값에 대한 해시 값은 캐싱이 되어짐.

Double-Checking

- Double-Checking은 비용이 적게/많이 드는 비교를 둘 다 사용

- 캐싱을 이용해 값이 있는지 체크하고 없다면
  값을 가져와서 해쉬값을 계산.

- 문자열의 경우 바이트 단위로 비교해야 되지만 그전에
  먼저 길이를 비교해서 빠르게 처리 가능.

- 두 값의 해싱 값을 비교해서 다르면 입력 값은 다른 거고 같다면
  두 값을 바이트 단위로 비교해야 함.