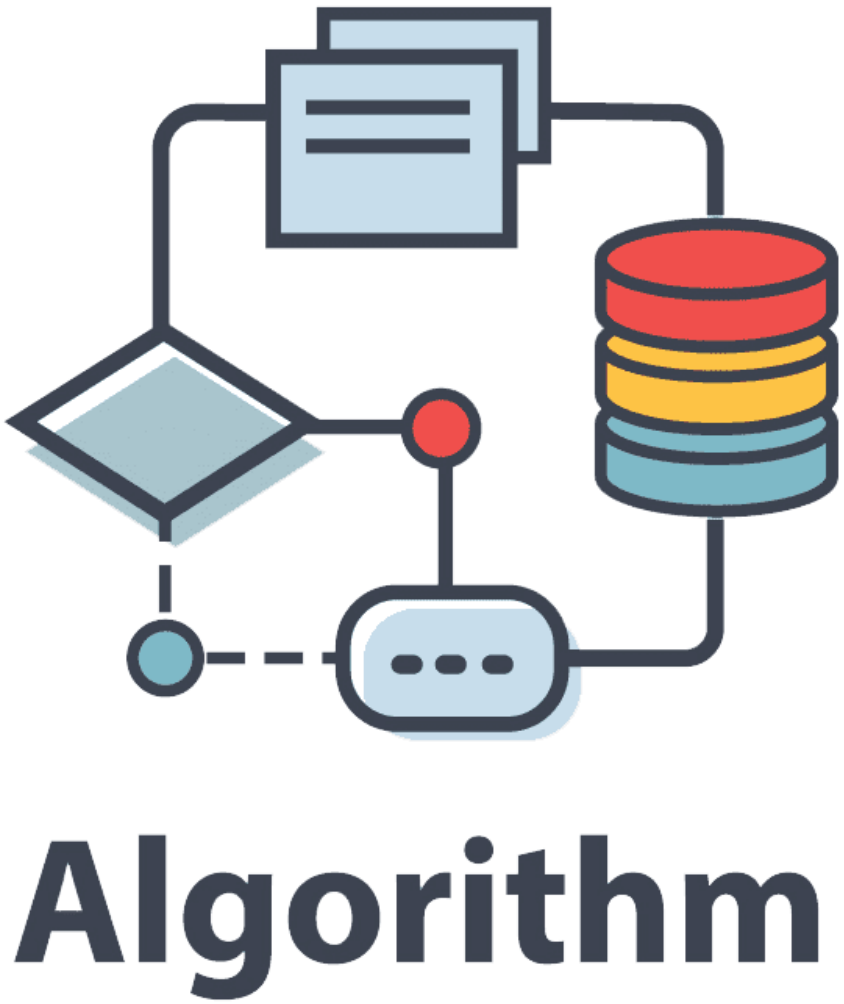
알고리즘 성능 분석 및 개선

김영현



알고리즘 특징

1. 입력 : 정의된 입력
2. 출력 : 답을 출력
3. 정밀성 : 변하지 않는 명확한 작업단계
4. 유일성 : 각 단계마다 명확한 다음 단계
5. 유한성 : 유한번에 작업으로 종료.
6. 타당성 : 구현성, 실용성

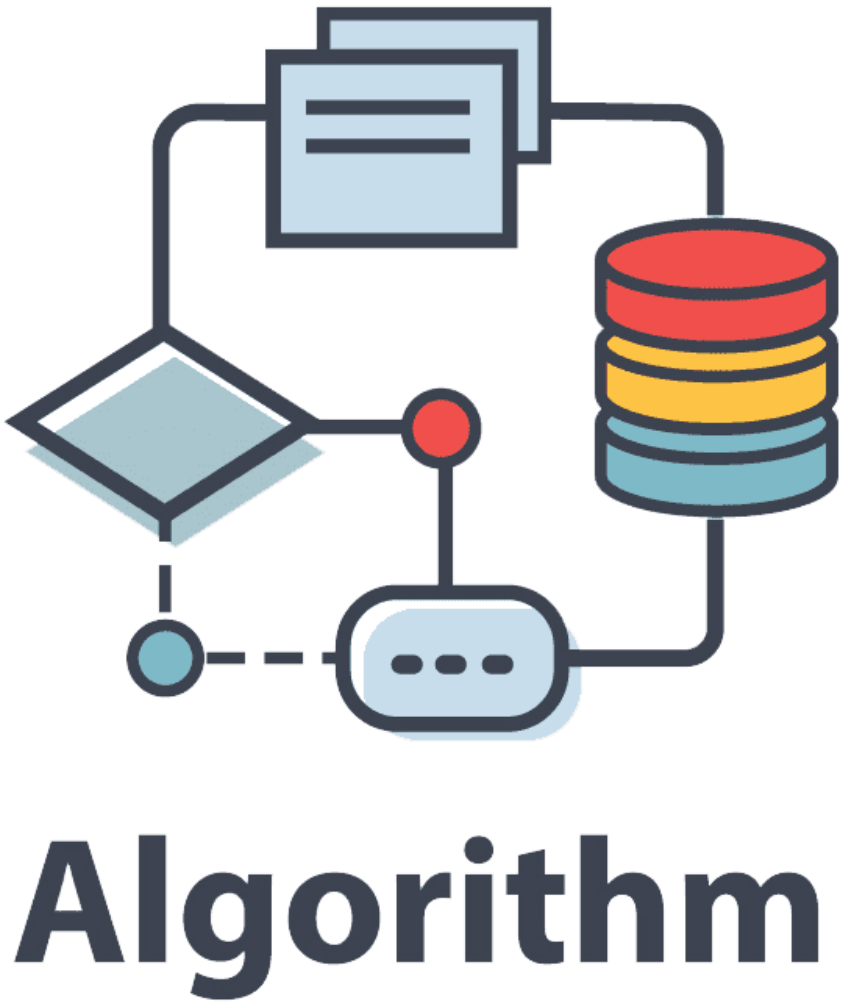
알고리즘 특징

특징

1. 입력 : 정의된 입력
2. 출력 : 답을 출력
3. 정밀성 : 변하지 않는 명확한 작업단계
4. 유일성 : 각 단계마다 명확한 다음 단계
5. 유한성 : 유한번에 작업으로 종료.
6. 타당성 : 구현성, 실용성

콜라츠 추측

- $T(n) = \begin{cases} 3n+1 \ \ if(n\ is\ odd) \\ \frac{n}{2} \ \ if\ (n\ is\ even) \end{cases}$
- 입력 : 자연수
- 출력 : 몇 번 만에 1로 떨어지는지.
- 유일성, 정밀성 :
- 유한성...?

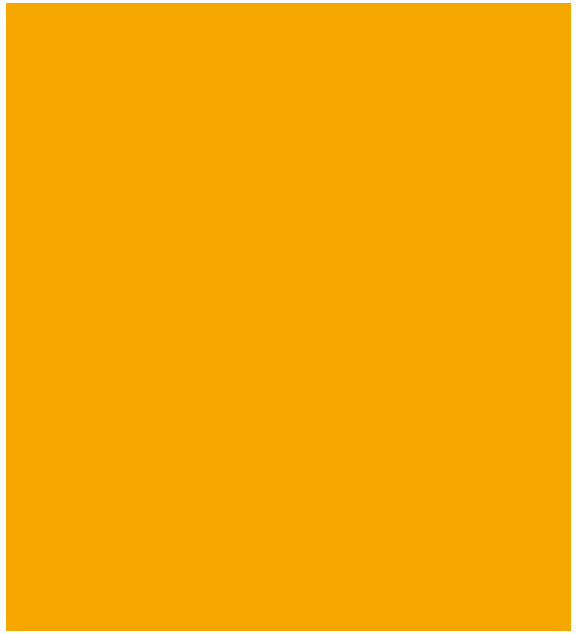


알고리즘 분석

- 시간 복잡도
- 문제 해결하는데 걸리는 시간과 입력 간의 함수 관계

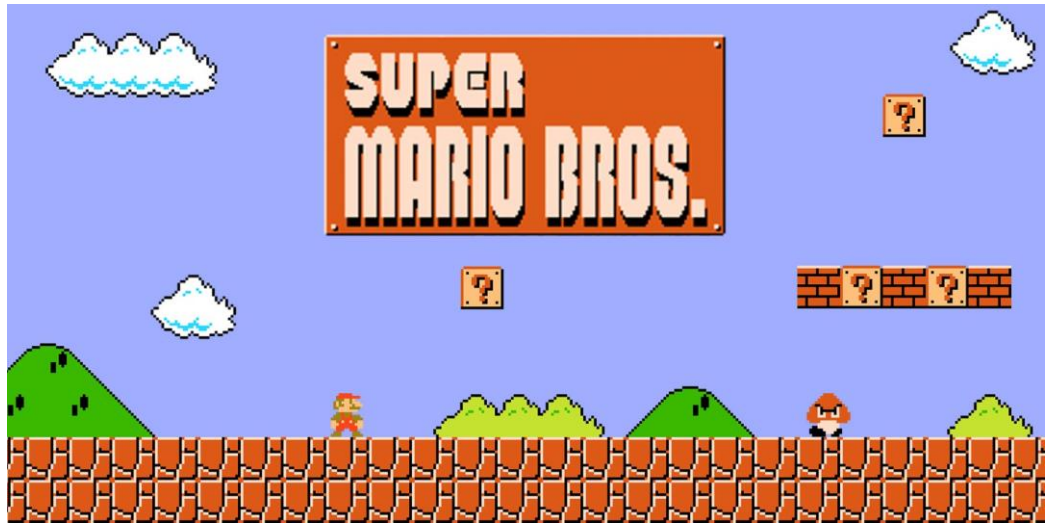
- 공간 복잡도
- 문제 해결하는데 필요한 메모리과 입력 간의 함수 관계

과거 프로그래밍과 현재 프로그래밍



과거 프로그래밍과 현재 프로그래밍

Super Mario Bros.



닌텐도 패밀리 컴퓨터

- CPU : 1.79 MHz
- 화소수 : 256x240
- RAM : 2kb

- 게임 용량 : 40kb

과거 프로그래밍과 현재 프로그래밍

League of Legends

최소 사양

- CPU : 2.93 GHZ, Dual Core
- 화소수 : 1024x768
- RAM : 2GB
- VRAM : 1GB
- 게임 용량 : 16GB

과거 프로그래밍과 현재 프로그래밍

프로그래밍 최적화 – 연산 최적화

Inverse square root

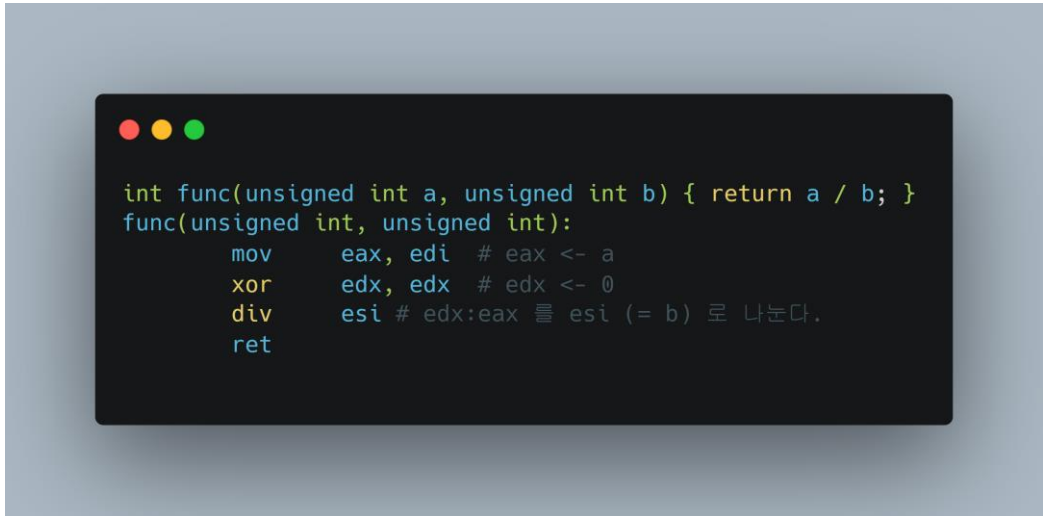
```
float inverse_square_root(float number)
{
    return 1.0f / sqrt(number)
}
```

Quake III Arena
- Fast Inverse Square Root

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

프로그래밍 최적화 – 연산 최적화

정수 나눗셈



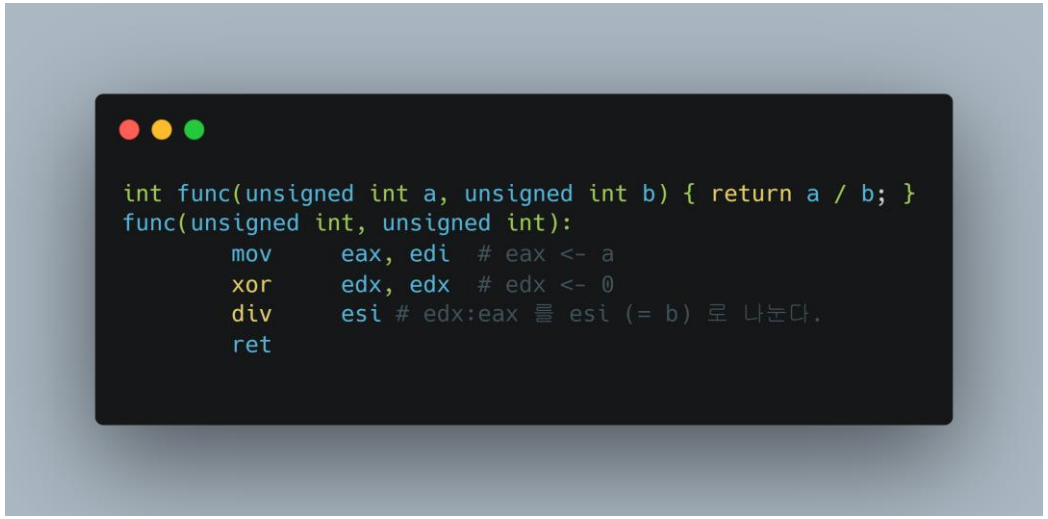
```
int func(unsigned int a, unsigned int b) { return a / b; }
func(unsigned int, unsigned int):
        mov     eax, edi  # eax <- a
        xor     edx, edx  # edx <- 0
        div     esi # edx:eax 를 esi (= b) 로 나눈다.
        ret
```

특정한 정수 나눗셈

```
int func(unsigned int d) { return d / 100; }
func(unsigned int):
        mov     eax, edi
        imul    rax, rax, 1374389535
        shr     rax, 37
        ret
```

프로그래밍 최적화 – 연산 최적화

정수 나눗셈



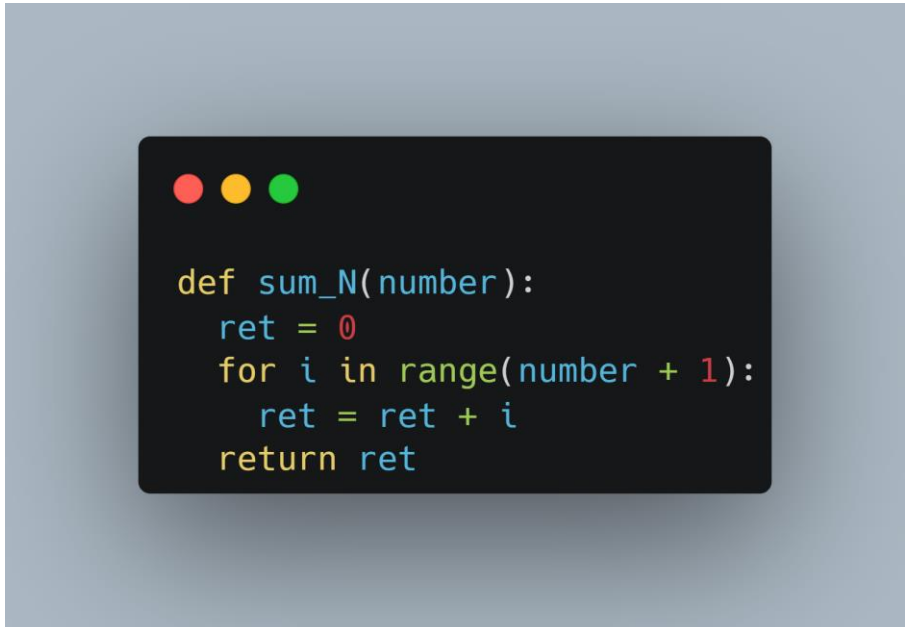
```
int func(unsigned int a, unsigned int b) { return a / b; }
func(unsigned int, unsigned int):
        mov     eax, edi  # eax <- a
        xor     edx, edx  # edx <- 0
        div     esi # edx:eax 를 esi (= b) 로 나눈다.
        ret
```

특정한 정수 나눗셈

```
int func(unsigned int d) { return d / 100; }
func(unsigned int):
        mov     eax, edi
        imul    rax, rax, 1374389535
        shr     rax, 37
        ret
```

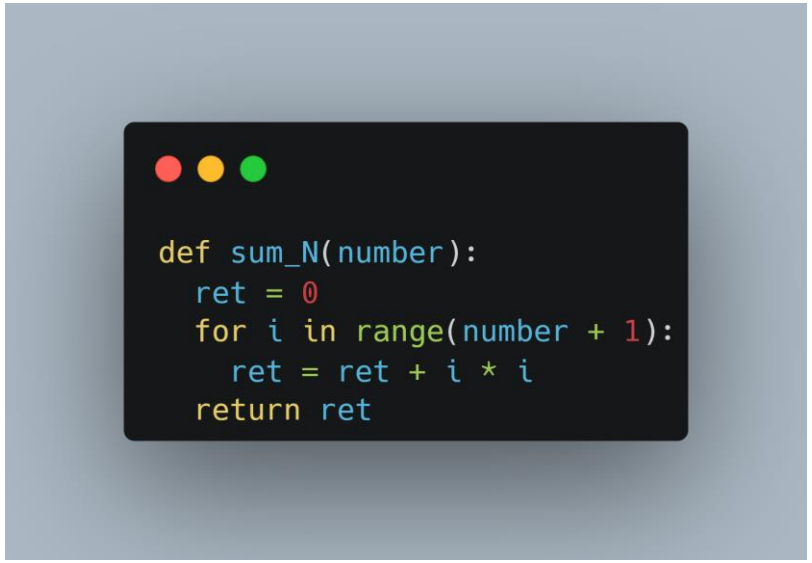
프로그래밍 최적화 – 알고리즘 최적화

1부터 N까지 덧셈



```python
def sum_N(number):
  ret = 0
  for i in range(number + 1):
    ret = ret + i
  return ret
```

1부터 N까지 제곱 덧셈



```python
def sum_N(number):
  ret = 0
  for i in range(number + 1):
    ret = ret + i * i
  return ret
```

프로그래밍 최적화 – 알고리즘 최적화

1부터 N까지 덧셈

```
def sum_N(number):
  ret = number * (number + 1) // 2
  return ret
```

1부터 N까지 제곱 덧셈

```
def sum_N(number):
  ret = number * (number + 1) * (2 * number + 1) // 6
  return ret
```

정렬 알고리즘

알고리즘 특징

1. 입력 : 정의된 입력
2. 출력 : 답을 출력

정렬 알고리즘

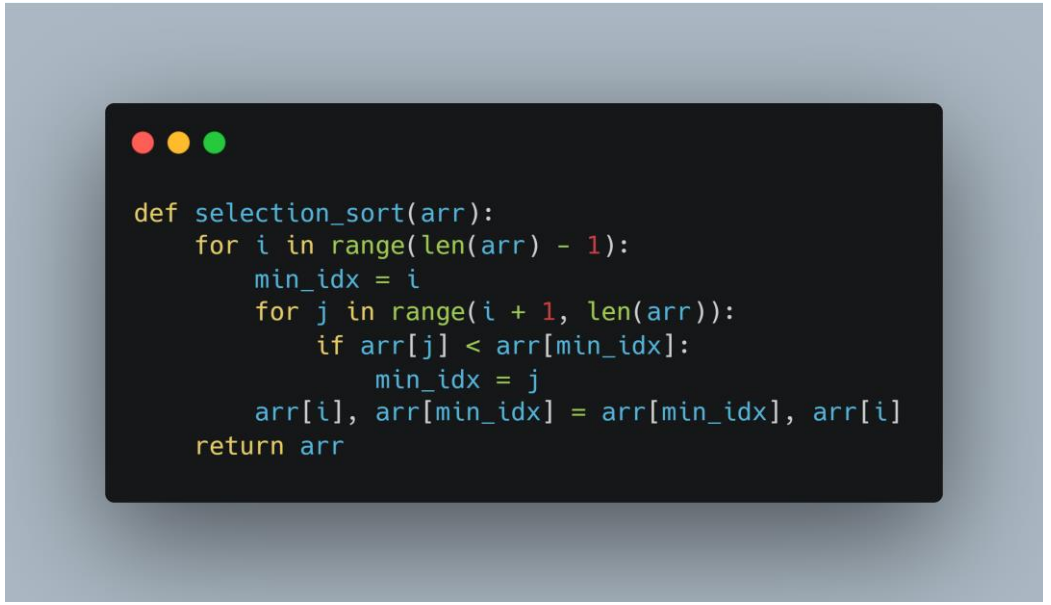
1. 입력 : 비교 가능한 객체 컨테이너 입력
2. 출력 : 정렬된 컨테이너 반환

정렬 알고리즘 – 선택 정렬

선택 정렬

1. 인덱스 맨 앞부터 가장 작은 값을 찾는다
2. 가장 작은 값을 찾으면 그 인덱스와 바꾼다
3. 다음 인덱스에서 위 과정을 마지막 인덱스까지 반복한다.

Python Code



```python
def selection_sort(arr):
    for i in range(len(arr) - 1):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

정렬 알고리즘 – 선택 정렬 vs 내장 함수

선택 정렬

1. 인덱스 맨 앞부터 가장 작은 값을 찾는다
2. 가장 작은 값을 찾으면 그 인덱스와 바꾼다
3. 다음 인덱스에서 위 과정을 마지막 인덱스까지 반복한다.

Python Code

```python
def selection_sort(arr):
    for i in range(len(arr) - 1):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

정렬 알고리즘 – 선택 정렬 vs 내장 함수

직접 구현한 선택 정렬

1. Pycarhm 실행

내장함수 sorted

- Pycharm 실행

정렬 알고리즘 – 선택 정렬 vs 내장 함수

직접 구현한 선택 정렬

- 선택 정렬

내장함수 sorted

- Tim sort
  - Hybrid (Merge Sort, Insertion Sort
  - Stable Sorting Algorithm

정렬 알고리즘 – 병합 정렬

병합 정렬

- 존 폰 노이만(John von Neumann) 제안
- 정복 분할(Divide and Conquer) 알고리즘

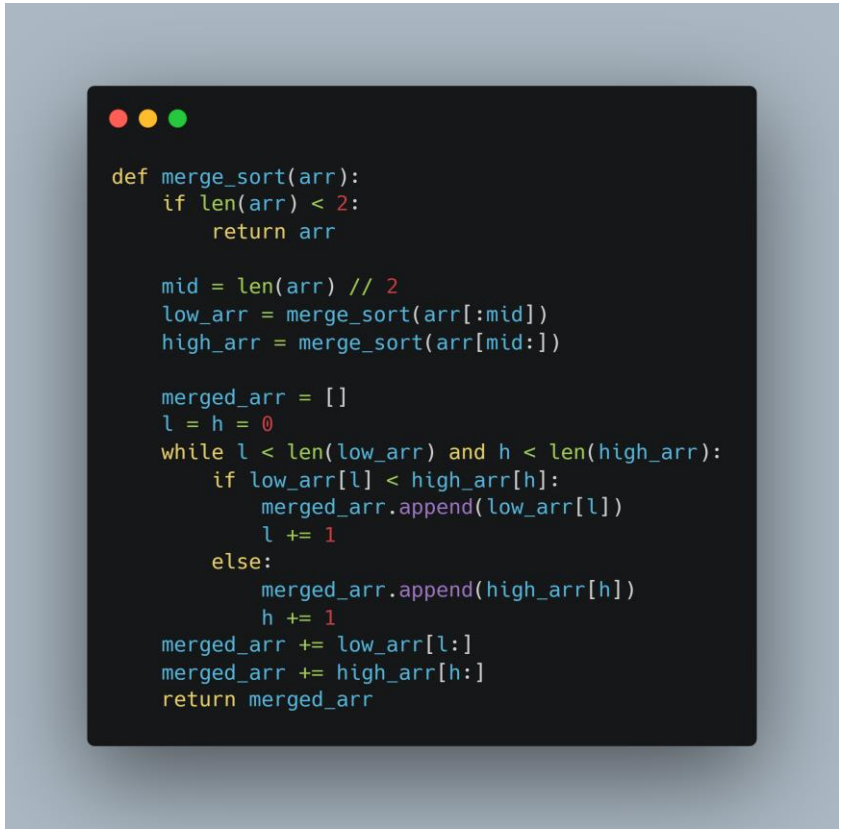
과정

- 리스트 길이가 0 또는 1이면 이미 정렬된 것으로 판단
- 정렬 되지 않는 리스트를 절반으로 나눈다
- 각 부분 리스트를 재귀적으로 합병 정렬
  - 정렬된 리스트가 되면 합친다.

정렬 알고리즘 – 병합 정렬

과정 및 알고리즘

- 리스트 길이가 0 또는 1이면 이미 정렬된 것으로 판단
- 정렬 되지 않는 리스트를 절반으로 나눈다
- 각 부분 리스트를 재귀적으로 병합 정렬
  - 정렬된 리스트가 되면 합병한다



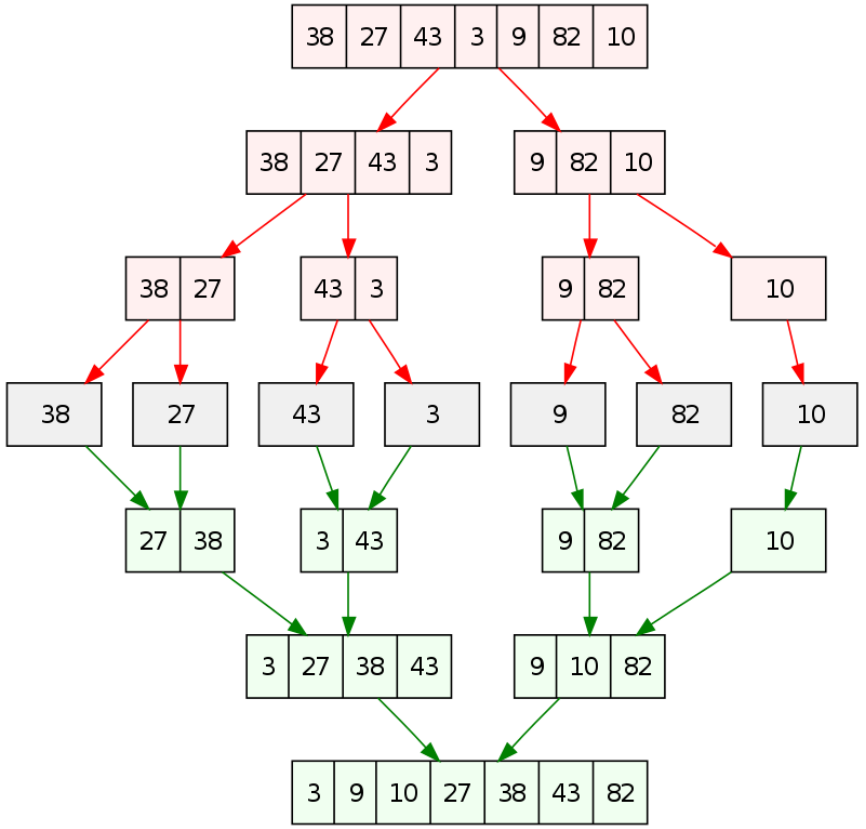
```python
def merge_sort(arr):
    if len(arr) < 2:
        return arr

    mid = len(arr) // 2
    low_arr = merge_sort(arr[:mid])
    high_arr = merge_sort(arr[mid:])

    merged_arr = []
    l = h = 0
    while l < len(low_arr) and h < len(high_arr):
        if low_arr[l] < high_arr[h]:
            merged_arr.append(low_arr[l])
            l += 1
        else:
            merged_arr.append(high_arr[h])
            h += 1
    merged_arr += low_arr[l:]
    merged_arr += high_arr[h:]
    return merged_arr
```

정렬 알고리즘 – 병합 정렬

시간 복잡도

- 트리 깊이 : log (n)
- 합치는 연산 : n

- 결론 : $O(n \log n)$



정렬 알고리즘 – 병합 정렬

최적화

- 메모리 사용 최적화
  - Before : 리스트 새로 생성
  - After : In-place하여 합침.

```python
def merge_sort(arr):
    def _sort(low, high):
        if high - low < 2:
            return
        mid = (low + high) // 2
        _sort(low, mid)
        _sort(mid, high)
        _merge(low, mid, high)
    def _merge(low, mid, high):
        temp = []
        l, h = low, mid

        while l < mid and h < high:
            if arr[l] < arr[h]:
                temp.append(arr[l])
                l += 1
            else:
                temp.append(arr[h])
                h += 1
        while l < mid:
            temp.append(arr[l])
            l += 1
        while h < high:
            temp.append(arr[h])
            h += 1
        for i in range(low, high):
            arr[i] = temp[i - low]
    return _sort(0, len(arr))
```

정렬 알고리즘 – 밴치마크

선택정렬

- Pycharm 실행
- $O(n^2)$

- N = 1000, 약 0.03초
- N = 10000, 약 3.3초

병합 정렬

- Pycharm 실행
- $O(n \log n)$

- N = 1000, 약 0.008초
- N = 10000, 약 0.10초

병합 정렬 - 최적화

- Pycharm 실행
- $O(n \log n)$

- N = 1000, 약 0.006초
- N = 10000, 약 0.07초

정렬 알고리즘 – 밴치마크

병합 정렬 – 최적화

- Pycharm 실행
- $O(n \log n)$

- N = 10000, 약 0.07초
- N = 100000, 약 0.862초
- N = 1000000, 약 10.622초

Built-in

- Pycharm 실행
- $O(n \log n)$

- N = 10000, 약 0.001초
- N = 100000, 약 0.014초
- N = 1000000, 약 0.220초

정렬 알고리즘 – 밴치마크

같은 $O(n \log n)$인데 더 느린 이유

- Python sorted : 실제로는 C언어로 작성
- Implement Merge Sort : Python

- Python
  - PyObject 연산 차이
  - 데이터 지역성 (메모리 캐시)

Built-in 실제 코드

- https://github.com/python/cpython/blob/ba18c0b13ba3c08077ea3db6658328523823a33f/Objects/listobject.c#L1051

외판원 문제

모든 경우의 수를 확인

- 모든 경우의 수 = $n!$
- 시간 복잡도 : $O(n!)$

- N = 10 : 3628800
- N = 12 : 479001600

- 실제로 다룰 수 없다.

동적계획법(DP)

- 동적계획법을 사용
- 시간 복잡도 : $O(2^n n^2)$

- N = 10 : 102400
- N = 12 : 589824

- 공간 복잡도 : $O(n 2^n)$

근사 알고리즘

- Christofides Algorithm

- Genetic Algorithm

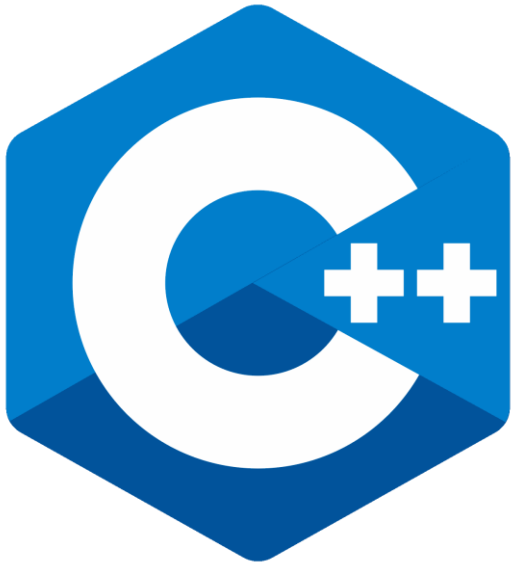
결론

알고리즘 설계 측면

- 문제의 니즈 정확하게 파악
  - 입력, 출력
- 사용하는 분야에서 필요한 속도 확인.
  - 너무 느린 알고리즘은 사용 할 수 없다.
- 근사를 사용하는 경우 오차 확인

최적화 관점

- 시간 복잡도를 최대한 줄이자.
  - 알고리즘 개선
- 연산량을 줄이자.
- 메모리 관점에서 생각하자.
- 하드웨어 가속을 사용



감사합니다.

김영현

kyh0581@naver.com

https://github.com/YoungHyuenKim