CS101 – 프로그램 실행 속도 향상 방법

Lecture 24

School of Computing
KAIST

학습 목표:

- 프로그램 실행 속도에 영향을 주는 해석기와 (interpreter) 컴파일러의 차이를 이해할 수 있다.
- 프로그램 실행 속도에 보다 큰 영향을 미치는 좋은 알고리즘의 중요성을 이해할 수 있다.

속도

왜 포토샵이 우리가 cs1media 모듈로 만든 그래픽 프로그램보다 빠를까요?

컴퓨터는 Python 언어를 직접적으로 이해할 수 없습니다.

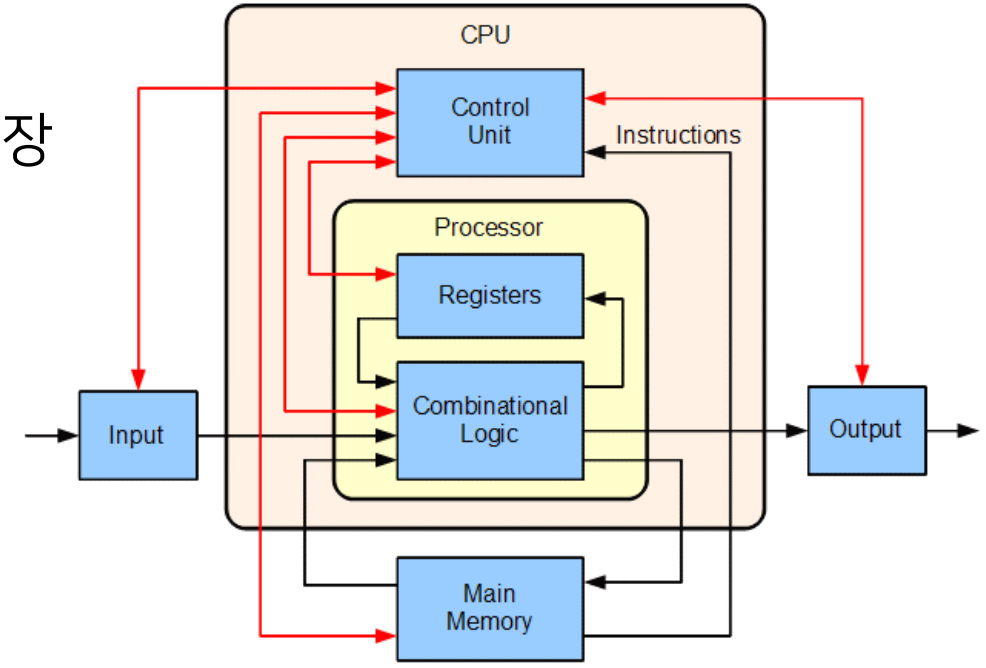
컴퓨터는 기계어라는 단 한 가지 언어만을 직접적으로 이해할 수 있습니다.
이 기계어는 CPU 종류마다 다르게 사용됩니다.

기계어는 숫자들의 나열로 이루어집니다.

```
21 37 158 228 255 10 49 26 88 250 12 ...
```

각 숫자는 의미를 가지고 있습니다.

- 저장소의 값을 읽어 CPU 레지스터에 저장
- 두 레지스터의 값을 더함
- 레지스터의 값을 저장소에 저장
- 두 숫자를 비교
- 다른 저장소 주소로 이동



해석기, 컴파일러

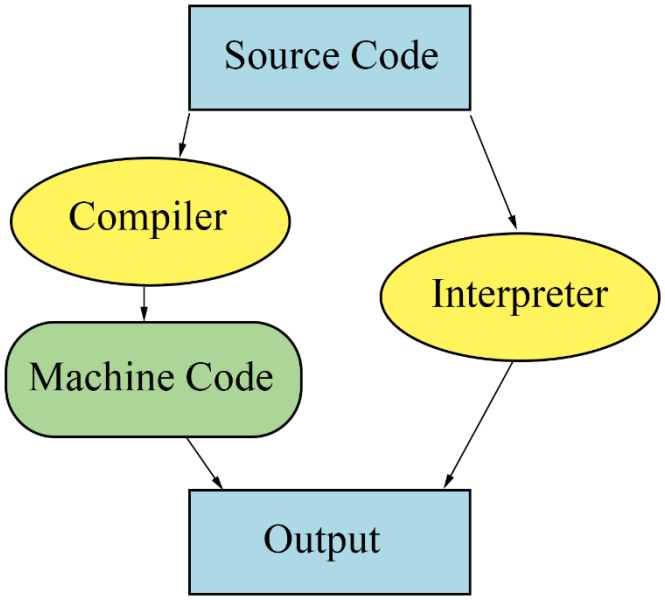
기계어로 된 명령들은 굉장히 빠르게 실행됩니다.
2GHz 프로세서는 1초에 무려 2,000,000,000개의 명령을 실행할 수 있습니다.

하지만, 기계어 명령들은 굉장히 알아보기 힘들게 되어 있어 기계어를 직접 다뤄서 프로그래밍하는 경우는 거의 없습니다.

Python은 **해석기(Interpreter)**를 사용합니다.
해석기는 Python 코드를 읽어서 명령을 하나씩 기계어로 바꿔 실행시켜주는 프로그램입니다.

해석기를 사용하는 프로그래밍 언어는
Scheme, Matlab, Flash 등이 있습니다.

C, C++, Java, Fortran은 **컴파일러(Compiler)**를 사용합니다. 컴파일러는 프로그램 코드를 읽어서 기계어 명령들로 된 프로그램으로 바꿔주는 프로그램입니다.

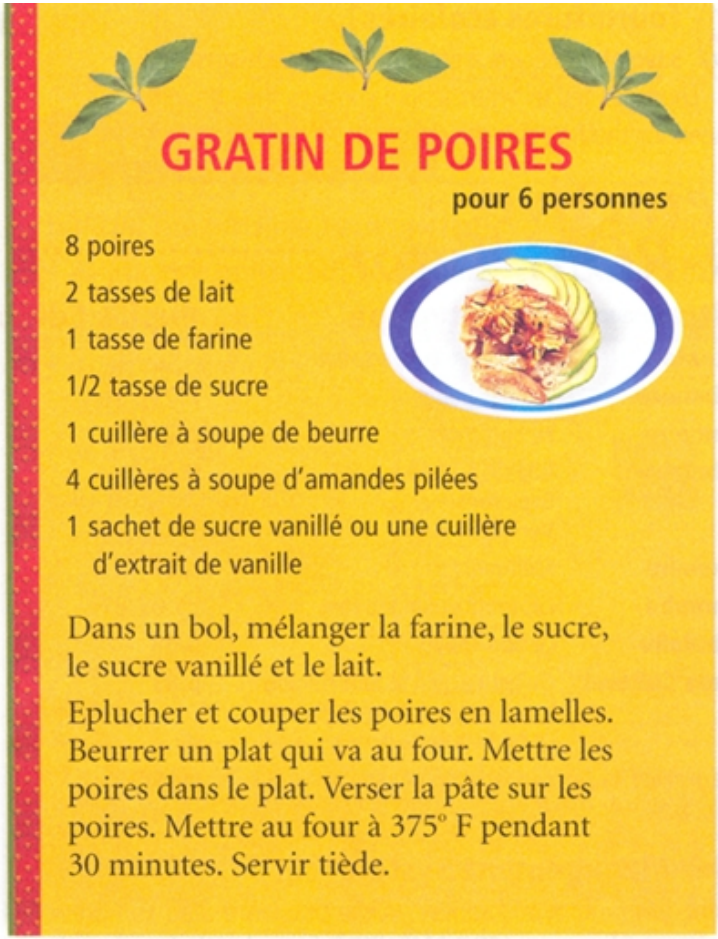


해석기, 컴파일러

해석기를 사용하는 것은, 외국어로 된 요리 책을 보면서 요리를 한 단계씩 만드는 것과 비슷합니다. 책을 조금씩 번역하며 요리를 만들기 때문에 요리를 천천히 만들게 됩니다.

비슷한 요리를 여러 차례 만들어야 할 때는, 요리 레시피를 모국어로 번역해서 사용하는 것이 더 효율적입니다. 컴파일러가 이런 일을 합니다.

레시피를 한 번에 모두 번역하는 데는 시간이 필요합니다. 한 번만 만들어야 하는 요리는 이렇게 하는게 더 비효율적일 수 있습니다. 하지만, 번역이 끝나면 같은 요리를 빠르게 여러 번 만들 수 있습니다.



해석기를 사용하는 이유

Python 해석기는 사용자의 명령과 상호작용할 수 있습니다.
사용자는 명령을 하나씩 실행하며 쉽게 테스트를 해볼 수 있습니다.

사용자는 현재 사용되는 객체에 어떤 값이 들어있는지 쉽게 알 수 있습니다.
객체가 가진 값을 수학적으로 분석할 수도 있습니다.
(C++에서 비슷한 일을 하려면 더 어려운 방법을 써야 합니다)

해석기를 사용한 프로그램은 프로그램을 쉽게 바꾸고 실행할 수 있어 디버깅을 더 쉽게 할 수 있습니다.

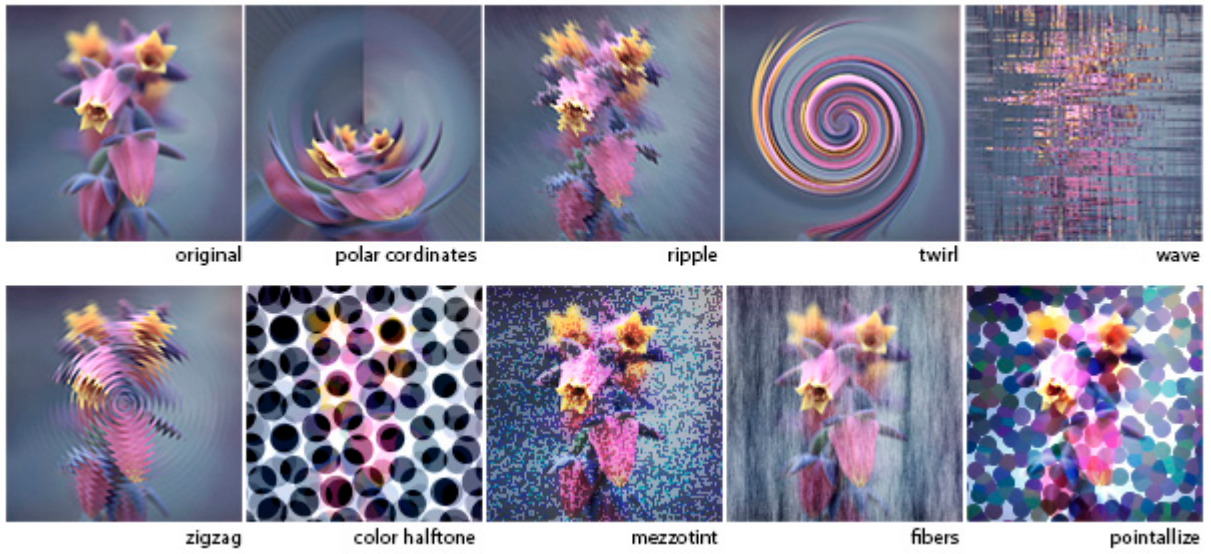
해석기는 프로그램의 메모리도 자동으로 관리합니다.

좋은 알고리즘

왜 포토샵 프로그램이 우리가 cs1media 모듈로 만든 그래픽 프로그램보다 빠를까요?

- 포토샵은 C++로 작성되어 기계어로 컴파일된 프로그램입니다.
- 포토샵은 더 좋은 알고리즘을 사용합니다.

해석기를 사용하는 프로그램도 좋은 알고리즘을 사용하면 컴파일러를 사용하는 프로그램보다 속도가 더 빠를 수 있습니다.



예제: 정렬

다음은 리스트 a를 정렬하는 간단한 알고리즘입니다.

```
def simple_sort(a):
  for i in range(len(a) - 1):
    for j in range(len(a) - 1):
      if a[j] > a[j+1]:
        a[j], a[j+1] = a[j+1], a[j]
```

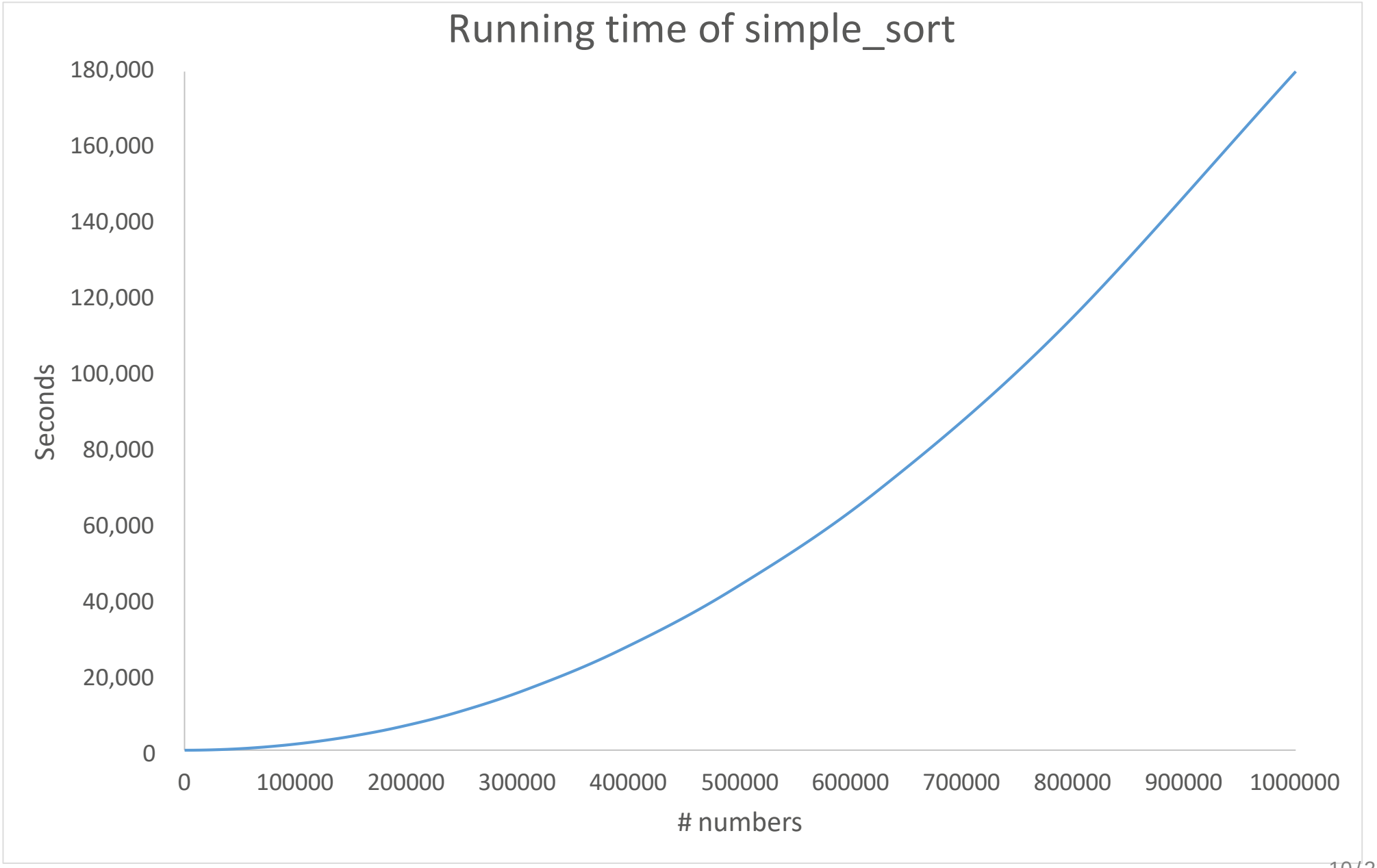
리스트 a가 n개의 원소를 가지고 있으면 **if** 문은 **총** $(n-1)^2$**번** 호출됩니다.

프로그램의 실행 시간을 측정해봅시다.

```
import random
import time
large_list = list(range(200000))
random.shuffle(large_list)
st = time.time()
simple_sort(large_list)
print("Running time: %f sec" % (time.time() - st))
```

예제: 정렬

아래 그래프는 위 프로그램을 Intel i7 3.60GHz 데스크탑에서 실행한 그래프입니다.
1백만 개의 원소를 가진 리스트를 정렬하는 데 **50시간** (180,055초) 걸렸습니다.



병합 정렬: 더 좋은 알고리즘

하나의 리스트는 한 개의 원소를 가지는 조각들로 나눌 수 있습니다.

그 후, 리스트 전체가 정렬될 때까지 두 조각씩 합쳐서 정렬된 한 조각으로 만듭니다.

정렬된 두 리스트 a, b를 합치는 것은 쉽습니다
(a, b 의 원소를 하나씩 순차적으로 비교하면 됩니다)

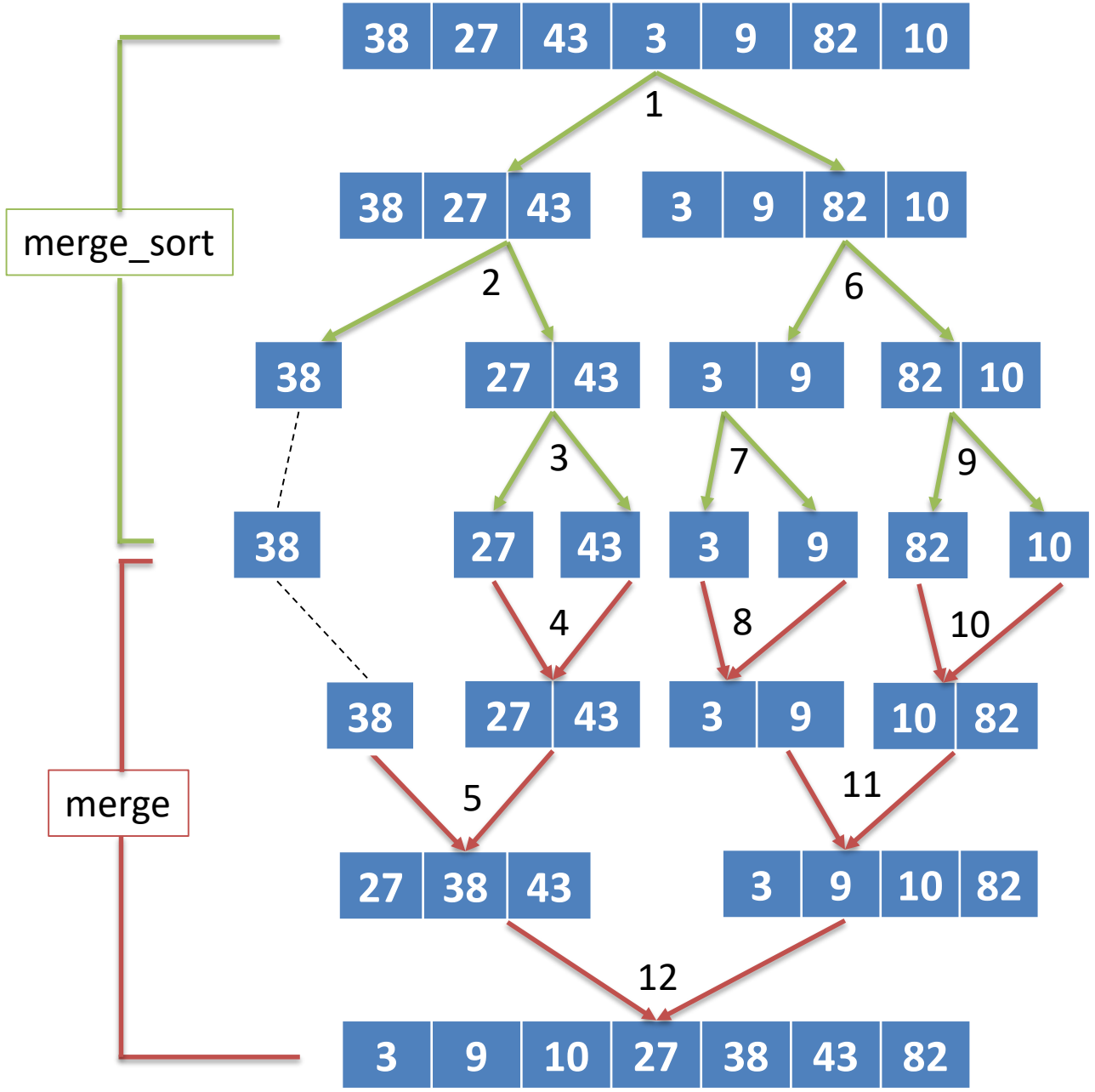
이 알고리즘은 두 원소의 값을 **최대 $n\,log_2 n$ 번** 비교합니다.

프로그램의 실행 시간을 측정해봅시다.

```
import random
import time
large_list = list(range(200000))
random.shuffle(large_list)
st = time.time()
merge_sort(large_list)
print("Running time: %f sec" % (time.time() - st))
```

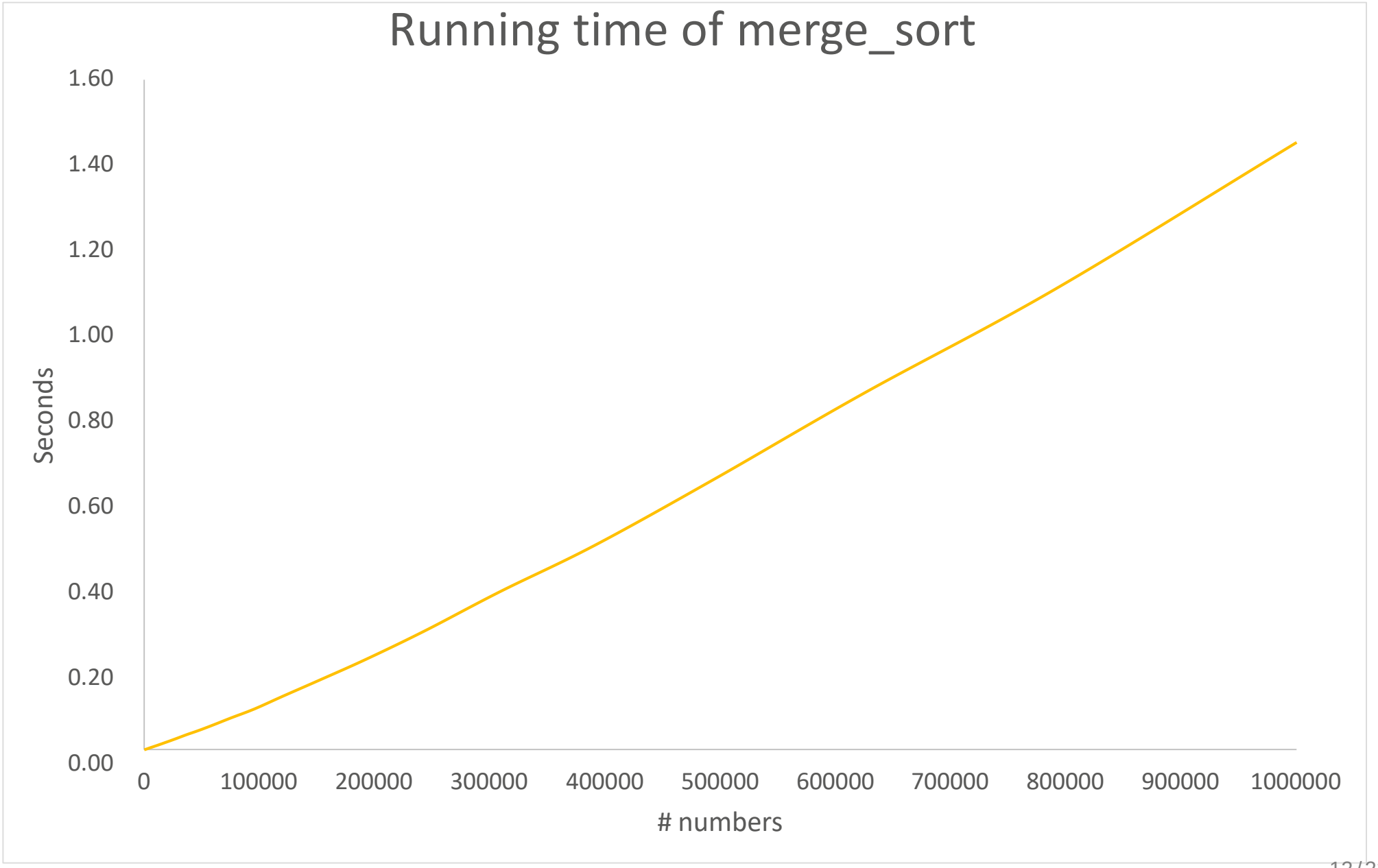
병합 정렬은 재귀적이다

```
def merge_sort(a):
    if len(a) <= 1:
        return a
    m = len(a)//2
    a1 = a[:m]
    a2 = a[m:]
    l = merge_sort(a1)
    r = merge_sort(a2)
    lst= merge(l,r)
    return lst
```



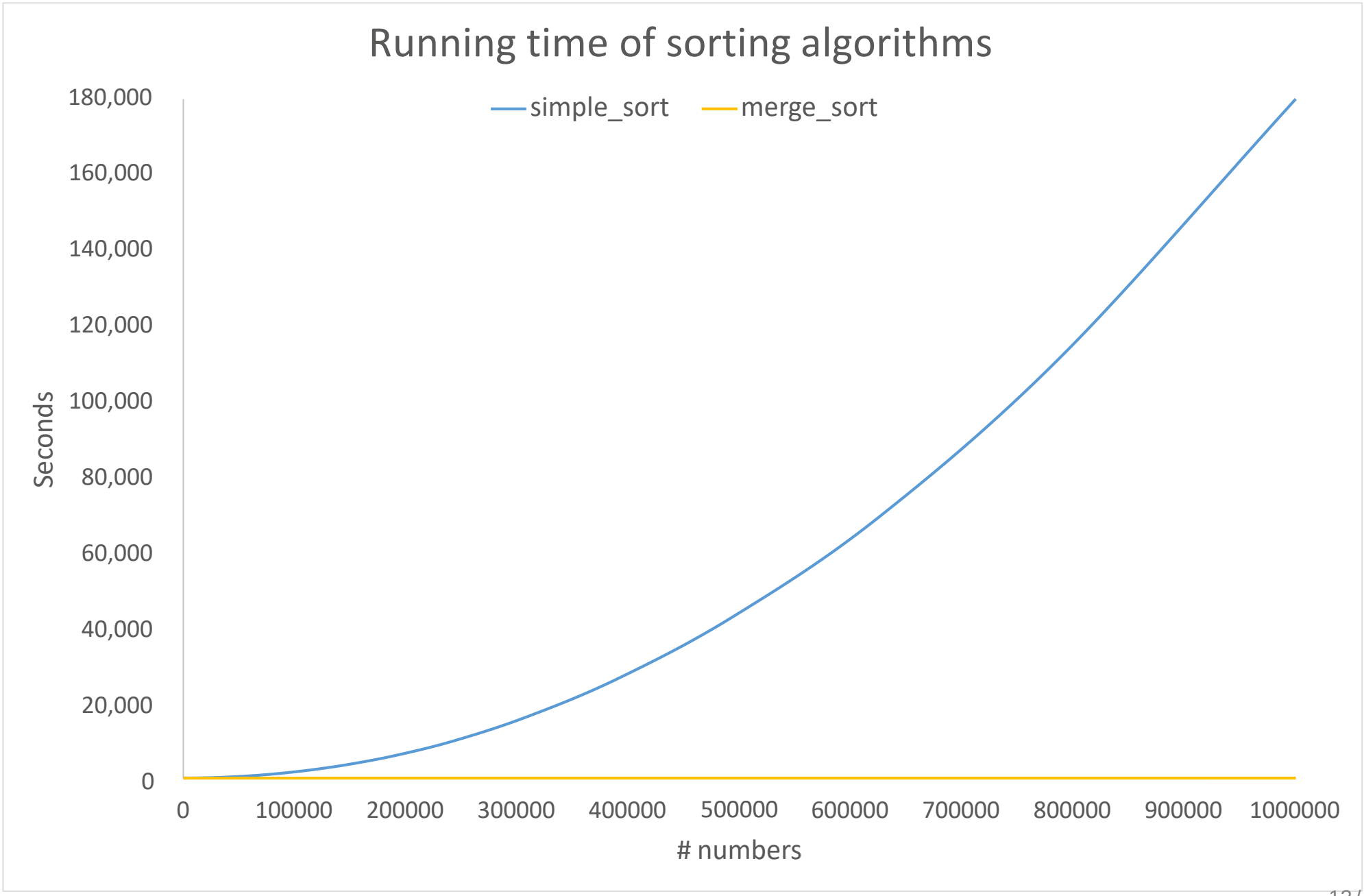
병합 정렬: 더 좋은 알고리즘

아래 그래프는 위 프로그램을 Intel i7 3.60GHz 데스크탑에서 실험한 그래프입니다.
1백만 개의 원소를 가진 리스트를 정렬하는 데 **1.45초**가 걸렸습니다.



실행 시간 비교

simple_sort와 merge_sort의 실행 시간을 비교해봅시다.



알고리즘 분석

문제를 해결하기 위한 더 효율적인 알고리즘을 만드는 것은 전산학의 중요한 분야입니다.

알고리즘이 사용하는 연산의 최대 횟수는 입력 값에 대한 사이즈 n의 함수 형태로 나타낼 수 있습니다. 이 함수가 더 작을 때, 알고리즘이 더 **효율적(Efficient)**이라고 말합니다.

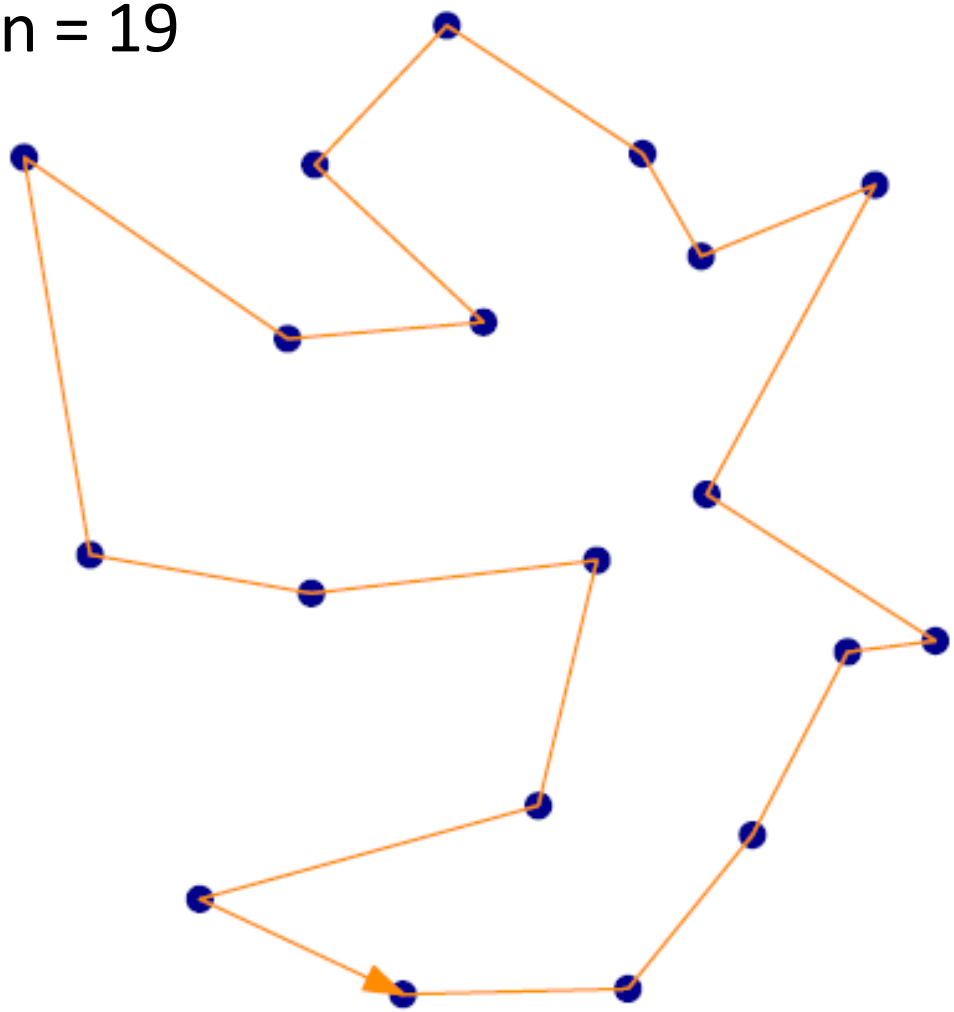
어떤 알고리즘보다 더 적은 연산으로 문제를 해결할 수 없다는 것을 증명할 수 있으면, 그 알고리즘을 **최적(Optimal)**의 알고리즘이라 부릅니다.

n개의 숫자는 $n\,log_2 n$ 보다 적은 비교 횟수로 정렬할 수 없다는 것을 증명할 수 있습니다. 따라서 병합 정렬은 최적의 알고리즘입니다.

효율적인 알고리즘

외판원 문제 :
평면상에 n개의 점이 있을 때, 모든 점을 방문하는 가장 짧은 경로를 구하시오.

n = 19



이 문제를 해결하는 지금까지 알려진 가장 효율적인 알고리즘은 2^n개의 연산을 사용합니다.

하지만, 더 빠르게 n^2개의 연산으로 문제를 해결할 수 없다는 것은 증명하지 못했습니다.

밀레니엄 문제:

$$P = NP?$$

문제를 해결하는 알고리즘이 없다는 것을 증명할 수 없는 문제들도 있습니다.

정리 및 예습

본 강의 학습목표:

- 프로그램 실행 속도에 영향을 주는 해석기와 (interpreter) 컴파일러의 차이를 이해할 수 있다.
- 프로그램 실행 속도에 보다 큰 영향을 미치는 좋은 알고리즘의 중요성을 이해할 수 있다.

다음 강의 학습 목표:

- 반복문과 유사한 재귀 (recursive) 함수를 이해하고 활용할 수 있다.
- CS101 강의에서 배운 모든 내용을 정리할 수 있다.