CS101 - Compilers/Interpreters and Algorithms

Lecture 11

School of Computing
KAIST

Course Evaluation

Course Evaluation

- When: June 4 (Monday) – June 8 (Friday)
- Where: KAIST Portal System

Roadmap

In the first half of the semester we covered

- Why we learn programming
- Boolean values, conditionals, loops
- Objects, types, variables, methods, operators, expressions, tuples
- Functions with parameters and return values
- Local and global variables, modules, graphics
- Sequences: lists, strings, tuples

Roadmap

In the first half of the semester we covered

- Why we learn programming
- Boolean values, conditionals, loops
- Objects, types, variables, methods, operators, expressions, tuples
- Functions with parameters and return values
- Local and global variables, modules, graphics
- Sequences: lists, strings, tuples

In the second half of the semester we are covering

- String methods, set, dictionary, image processing
- File I/O
- Object creation and attributes
- Object constructors, user interface programming
- Interpreters vs. compilers and algorithms

Speed

Why is Photoshop so much faster for image processing than our cs1media programs?

Speed

Why is Photoshop so much faster for image processing than our cs1media programs?

Because computers do not speak Python.

Speed

Why is Photoshop so much faster for image processing than our cs1media programs?

Because computers do not speak Python.

A computer directly understands only one kind of language – its machine language. This language is different for different CPUs.

Speed

Why is Photoshop so much faster for image processing than our cs1media programs?

Because computers do not speak Python.

A computer directly understands only one kind of language – its machine language. This language is different for different CPUs.

Machine language is just numbers in the memory:

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

Speed

Why is Photoshop so much faster for image processing than our cs1media programs?

Because computers do not speak Python.

A computer directly understands only one kind of language – its machine language. This language is different for different CPUs.

Machine language is just numbers in the memory:

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

Each number means some instruction:

- Load value from memory to CPU register
- Add two register values
- Store register value to memory
- Compare two numbers
- Jump to a new memory address

Interpreters & Compilers

Machine language instructions are very fast. A 2 GHz processor executes 2,000,000,000 instructions per second!

Interpreters & Compilers

Machine language instructions are very fast. A 2 GHz processor executes 2,000,000,000 instructions per second!

But machine instructions are very primitive, and nobody programs in machine language anymore.

Interpreters & Compilers

Machine language instructions are very fast. A 2 GHz processor executes 2,000,000,000 instructions per second!

But machine instructions are very primitive, and nobody programs in machine language anymore.

Python uses an *interpreter*. An interpreter is a program that reads your Python code and executes its instructions one after another.

Interpreters & Compilers

Machine language instructions are very fast. A 2 GHz processor executes 2,000,000,000 instructions per second!

But machine instructions are very primitive, and nobody programs in machine language anymore.

Python uses an *interpreter*. An interpreter is a program that reads your Python code and executes its instructions one after another.

Other interpreted languages are Scheme, Matlab, or Flash.

Interpreters & Compilers

Machine language instructions are very fast. A 2 GHz processor executes 2,000,000,000 instructions per second!

But machine instructions are very primitive, and nobody programs in machine language anymore.

Python uses an *interpreter*. An interpreter is a program that reads your Python code and executes its instructions one after another.

Other interpreted languages are Scheme, Matlab, or Flash.

Languages such as C, C++, Java, or Fortran are *compiled*. The input program (source code) is converted to a numeric format that contains machine instructions.

Interpreters & Compilers

Using an interpreter is like making a dish using a cooking book in a foreign language that you cannot read well. You need to look up many words, and you execute the recipe slowly.

Interpreters & Compilers

Using an interpreter is like making a dish using a cooking book in a foreign language that you cannot read well. You need to look up many words, and you execute the recipe slowly.

When we have to cook the same dish many times, it is more efficient to first translate the recipe and to write it down in your mother tongue. This is what a compiler does.

Interpreters & Compilers

Using an interpreter is like making a dish using a cooking book in a foreign language that you cannot read well. You need to look up many words, and you execute the recipe slowly.

When we have to cook the same dish many times, it is more efficient to first translate the recipe and to write it down in your mother tongue. This is what a compiler does.

It takes time and effort to do the translation and to write it down. It is not worth doing that if we only cook the dish once. But now we can cook the dish many times quickly.

Why interpreters?

The Python shell interactively executes our instructions, so we can experiment with objects of different types and test their behavior.

Why interpreters?

The Python shell interactively executes our instructions, so we can experiment with objects of different types and test their behavior.

We can also interactively explore data, or perform mathematical analysis (try symbolic differentiation in C++!).

Why interpreters?

The Python shell interactively executes our instructions, so we can experiment with objects of different types and test their behavior.

We can also interactively explore data, or perform mathematical analysis (try symbolic differentiation in C++!).

An interpreter makes it easier to debug a program, because changing the program is fast. We can also look at the objects created in the program.

Why interpreters?

The Python shell interactively executes our instructions, so we can experiment with objects of different types and test their behavior.

We can also interactively explore data, or perform mathematical analysis (try symbolic differentiation in C++!).

An interpreter makes it easier to debug a program, because changing the program is fast. We can also look at the objects created in the program.

Memory-management is done automatically by the interpreter.

Smarter algorithms

Why is Photoshop faster than our 'posterize' function?

- Photoshop is written in C++ and compiled to machine language.
- Photoshop uses smarter algorithms.

Smarter algorithms

Why is Photoshop faster than our ’posterize’ function?

- Photoshop is written in C++ and compiled to machine language.
- Photoshop uses smarter algorithms.

A smart algorithm in an interpreted language can easily beat a simple algorithm in a compiled language.

Example: Sorting

Here is a simple algorithm to sort a list `a`:

```
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]
```

Example: Sorting

Here is a simple algorithm to sort a list `a`:

```
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]
```

If the list `a` has `n` elements, then the **`if`** statement is executed $(n-1)^2$ times.

Example: Sorting

Here is a simple algorithm to sort a list `a`:

```
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]
```

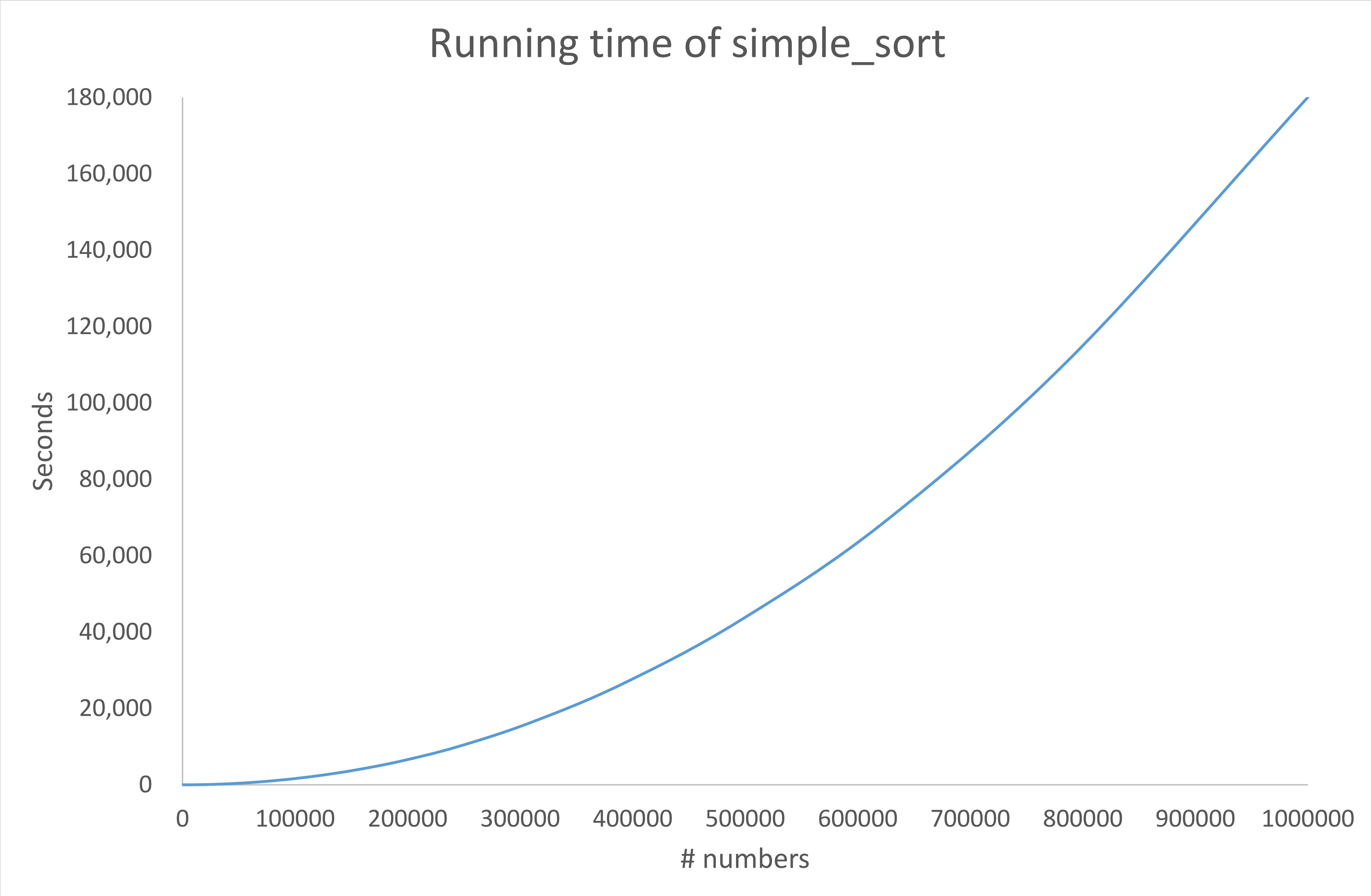
If the list `a` has `n` elements, then the **`if`** statement is executed $(n-1)^2$ times.

Test code:

```
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))
```

Example: Sorting

On a Intel i7 3.60GHz desktop, sorting 1M numbers takes 180,055 seconds.



Merge Sort: a smarter algorithm

We partition the list into small pieces of one element.

Merge Sort: a smarter algorithm

We partition the list into small pieces of one element.

Then we *merge* pieces together in pairs, until the whole list is sorted.

Merge Sort: a smarter algorithm

We partition the list into small pieces of one element.

Then we *merge* pieces together in pairs, until the whole list is sorted.

Merging two sorted lists a and b is easy, as in each step we only need to select an element from either a or b.

Merge Sort: a smarter algorithm

We partition the list into small pieces of one element.

Then we *merge* pieces together in pairs, until the whole list is sorted.

Merging two sorted lists a and b is easy, as in each step we only need to select an element from either a or b.

We can show that this algorithm compares two list elements only $n \log_2 n$ times.

Merge Sort: a smarter algorithm

We partition the list into small pieces of one element.

Then we *merge* pieces together in pairs, until the whole list is sorted.

Merging two sorted lists a and b is easy, as in each step we only need to select an element from either a or b.

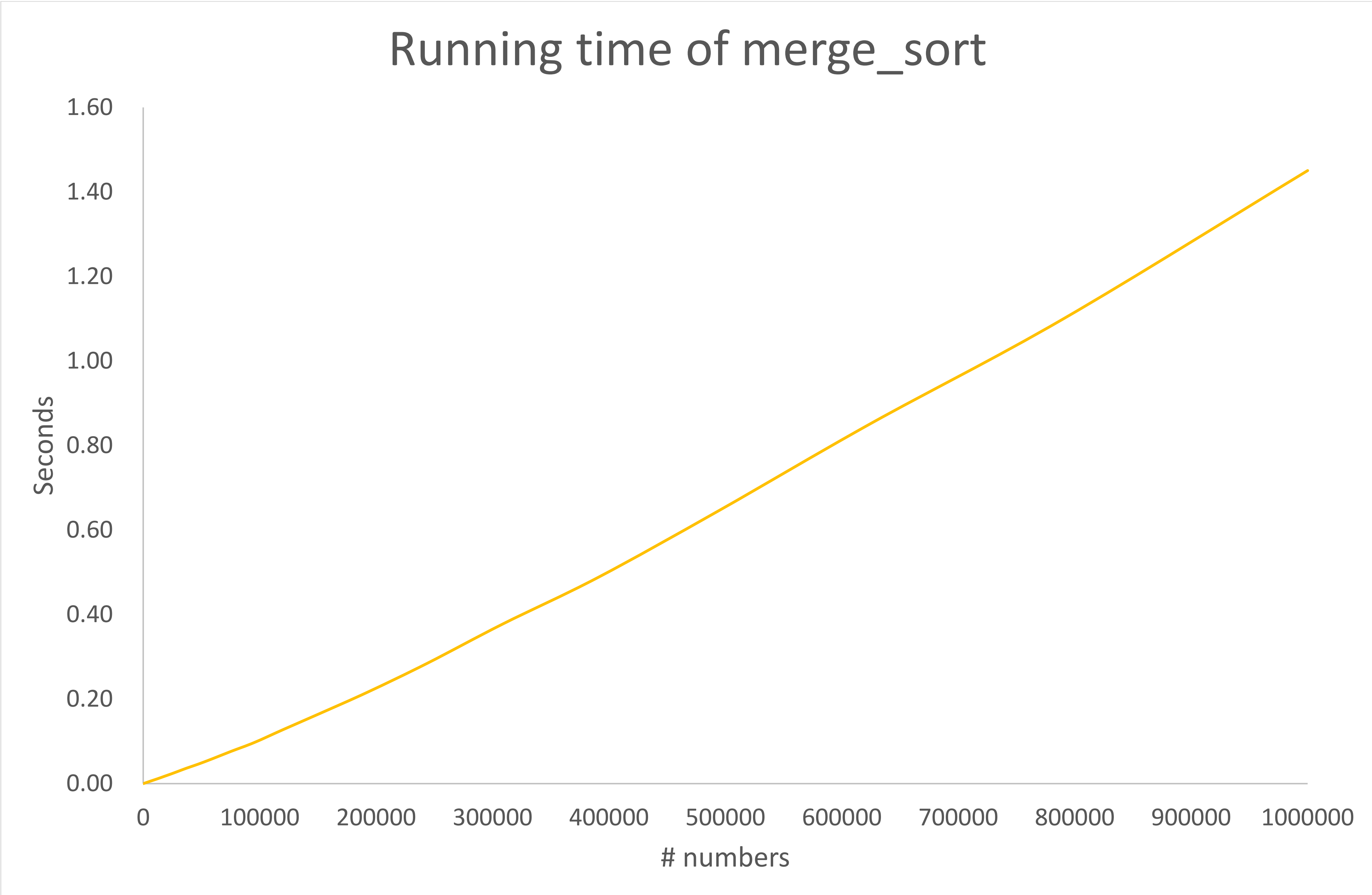
We can show that this algorithm compares two list elements only $n \log_2 n$ times.

Test code:

```
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))
```

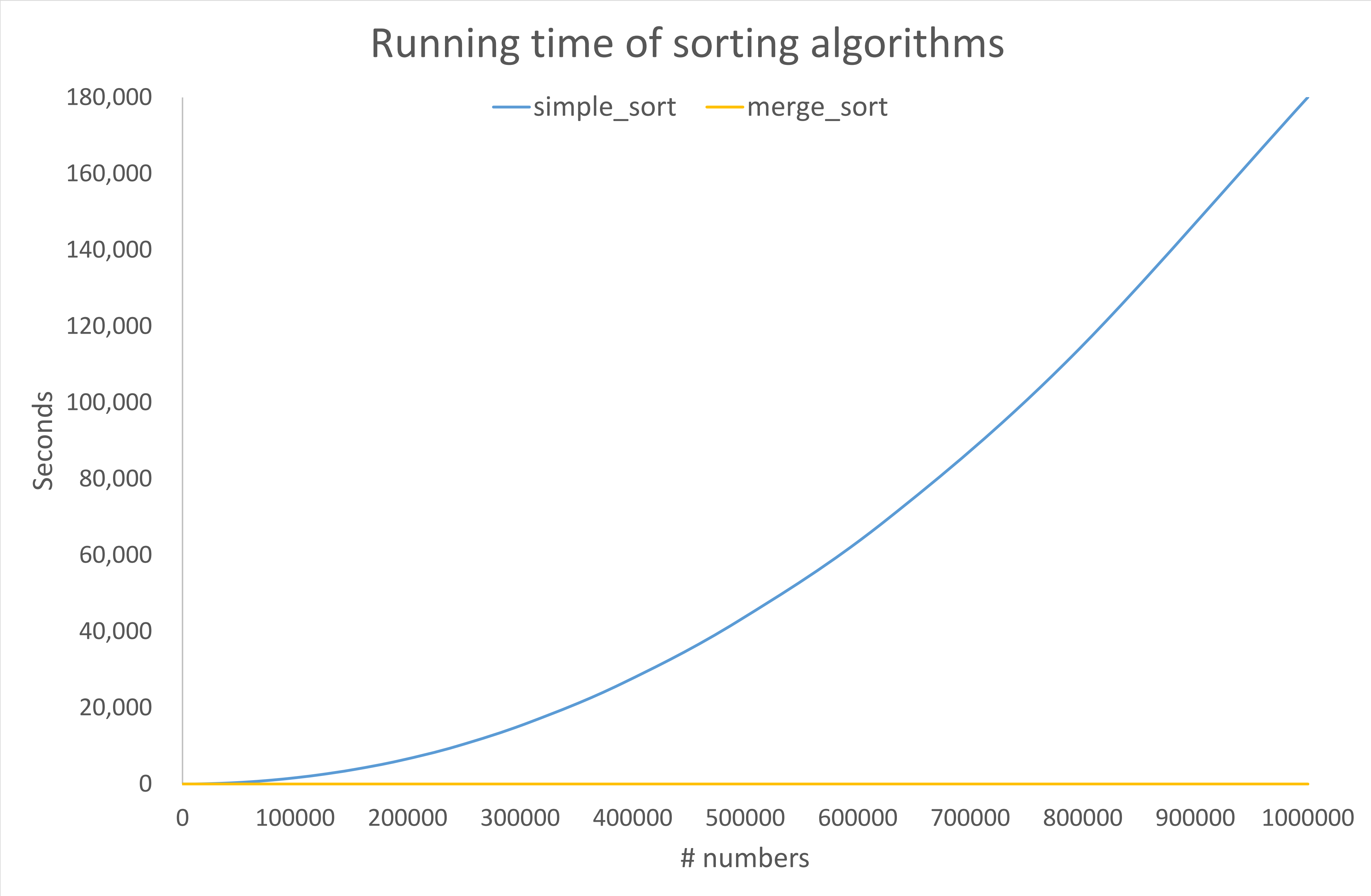
Merge Sort: a smarter algorithm

On a Intel i7 3.60GHz desktop, sorting 1M numbers takes 1.45 seconds.



Running time comparison

The comparison of `simple_sort` and `merge_sort` running time



Algorithm analysis

Designing efficient algorithms for a problem is a fundamental branch of computer science.

Algorithm analysis

Designing efficient algorithms for a problem is a fundamental branch of computer science.

Here, “efficient” means that we can prove that the number of operations made by the algorithm is bounded by some function of the problem size n.

Algorithm analysis

Designing efficient algorithms for a problem is a fundamental branch of computer science.

Here, “efficient” means that we can prove that the number of operations made by the algorithm is bounded by some function of the problem size n.

An algorithm is *optimal* if we can prove that no algorithm can possibly solve the problem with a smaller number of operations (asymptotically).

Algorithm analysis

Designing efficient algorithms for a problem is a fundamental branch of computer science.

Here, “efficient” means that we can prove that the number of operations made by the algorithm is bounded by some function of the problem size n.

An algorithm is *optimal* if we can prove that no algorithm can possibly solve the problem with a smaller number of operations (asymptotically).

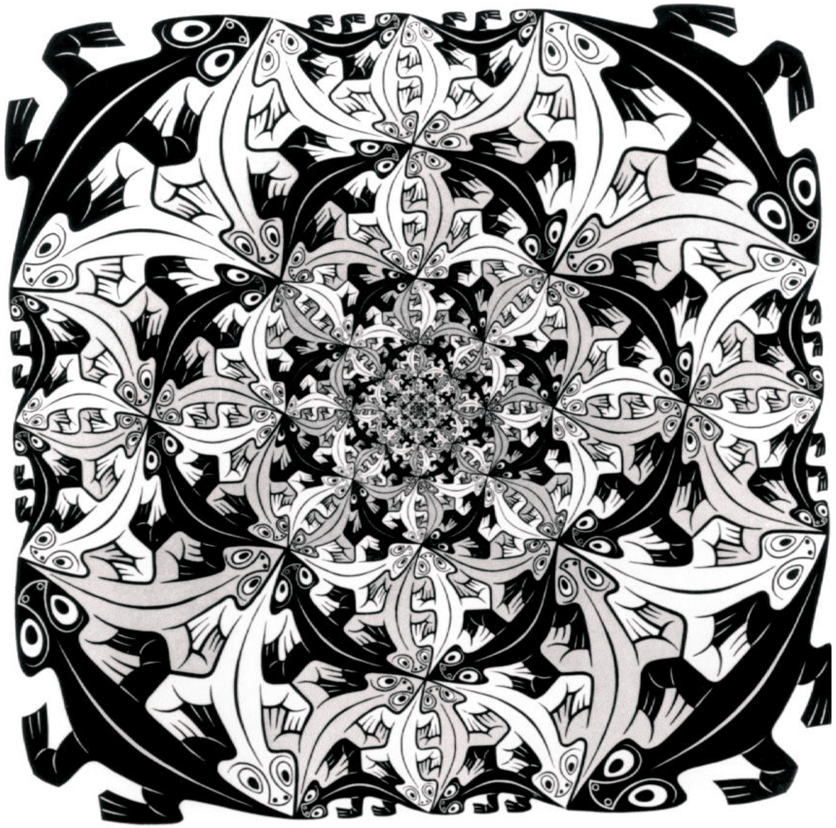
We can prove that it is impossible to sort n numbers with less than $n \log_2 n$ comparisons, and therefore Merge Sort is optimal.

Recursion

"Recursion" means to define something in terms of itself.

A folder is a collection of files and folders.

Words in dictionaries are defined in terms of other words.



A recursive function

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

A recursive function

How would you implement this function?

```
>>> downup("Hello")
Hello
Hell
Hel
He
H
He
Hel
Hell
Hello
```

A recursive function

How would you implement this function?

```
>>> downup("Hello")
Hello
Hell
Hel
He
H
He
Hel
Hell
Hello
```

One solution

```
def downup(w):
  print(w)
  if len(w) <= 1:
    return
  downup(w[:-1])   # Recursive call
  print(w)
```

Printing in any base

How do you print a number in binary? Or in base 8?

Printing in any base

How do you print a number in binary? Or in base 8?

The last digit of number n in base b is easy: `n % b`

The remaining digits are the representation of `n // b` in base b:

Printing in any base

How do you print a number in binary? Or in base 8?

The last digit of number n in base b is easy: `n % b`

The remaining digits are the representation of `n // b` in base b:

```
def to_radix(n, b):
  if n < b:
    return str(n)
  s = to_radix(n // b, b)
  return s + str(n % b)
```

Merge Sort is recursive

Merge Sort is an example of a more interesting recursive algorithm. It uses two recursive calls:

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

Merge Sort is recursive

Merge Sort is an example of a more interesting recursive algorithm. It uses two recursive calls:

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

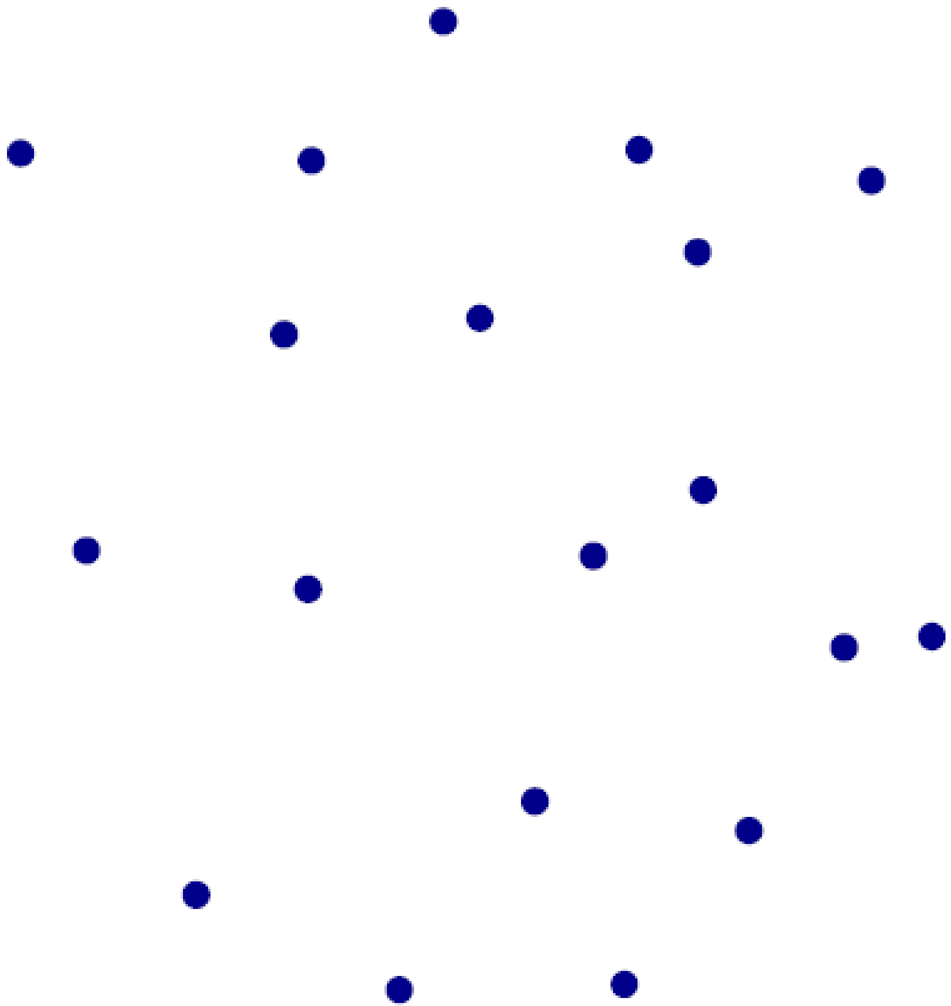
Divide & Conquer: Divide a problem into two smaller problems. Solve the smaller problems, and combine the solutions.

Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.

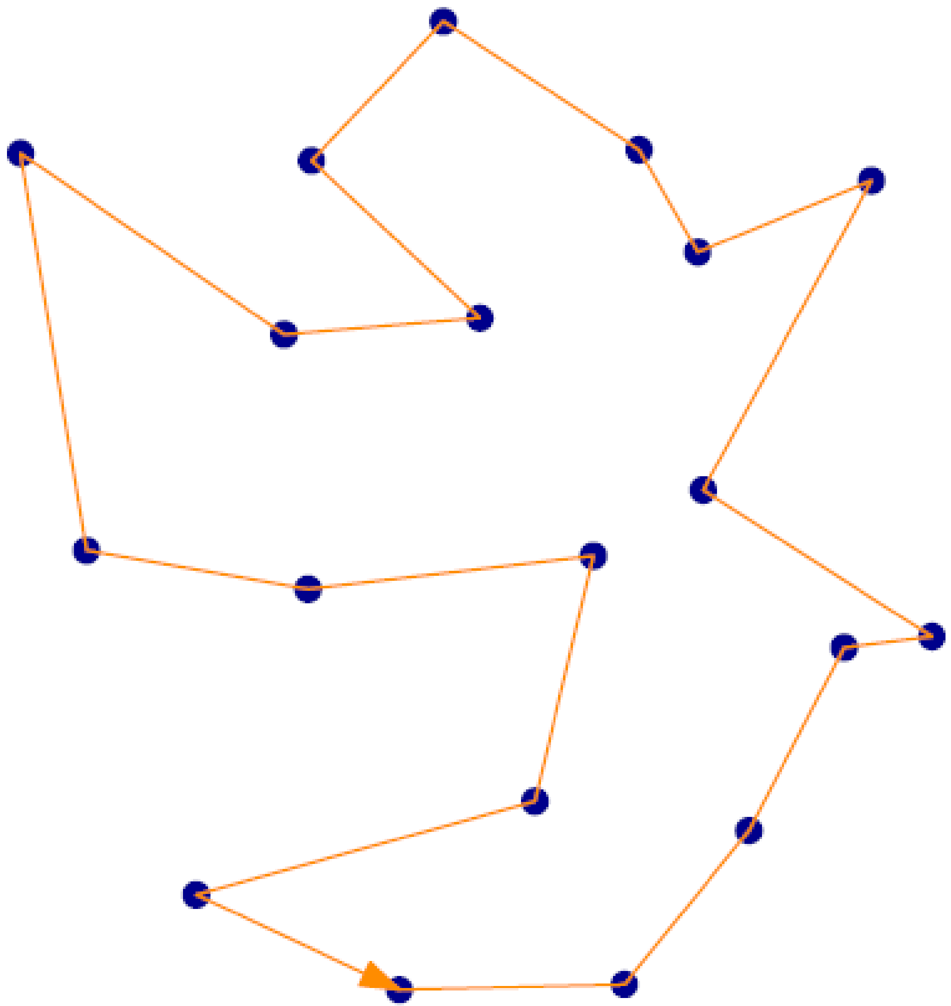
Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.



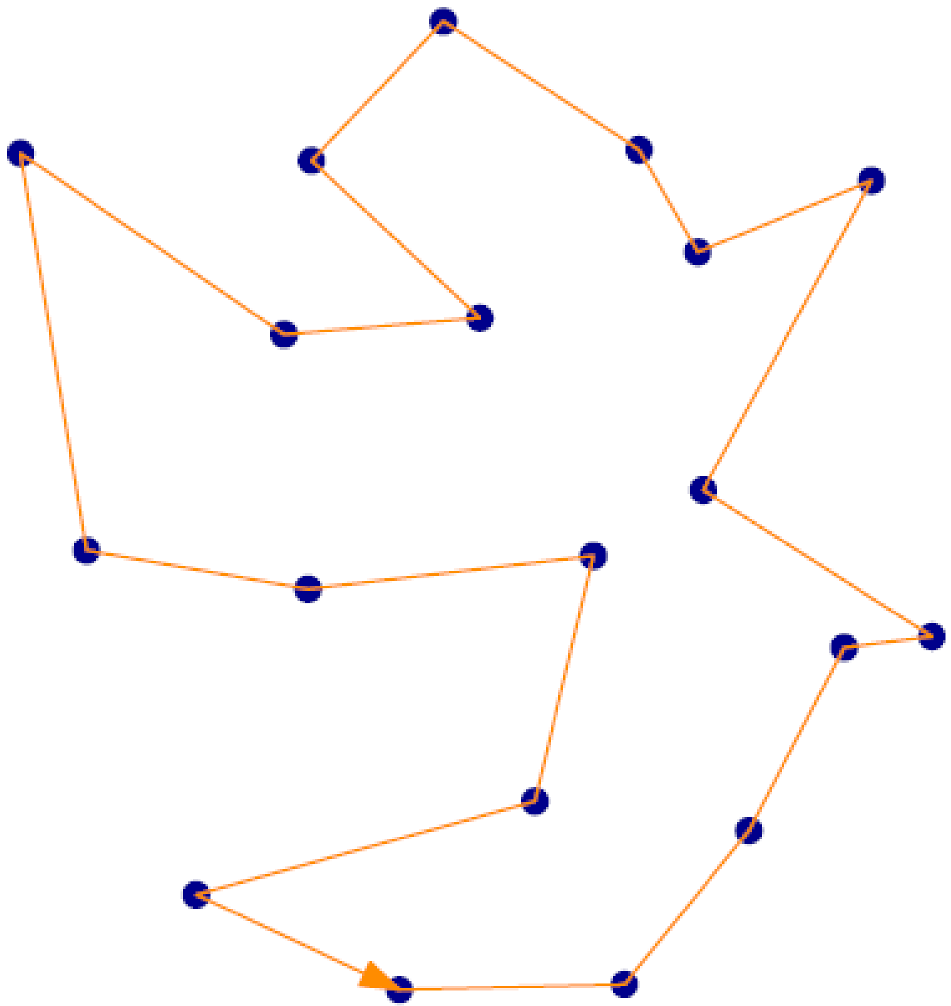
Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.



Efficient algorithms for everything?

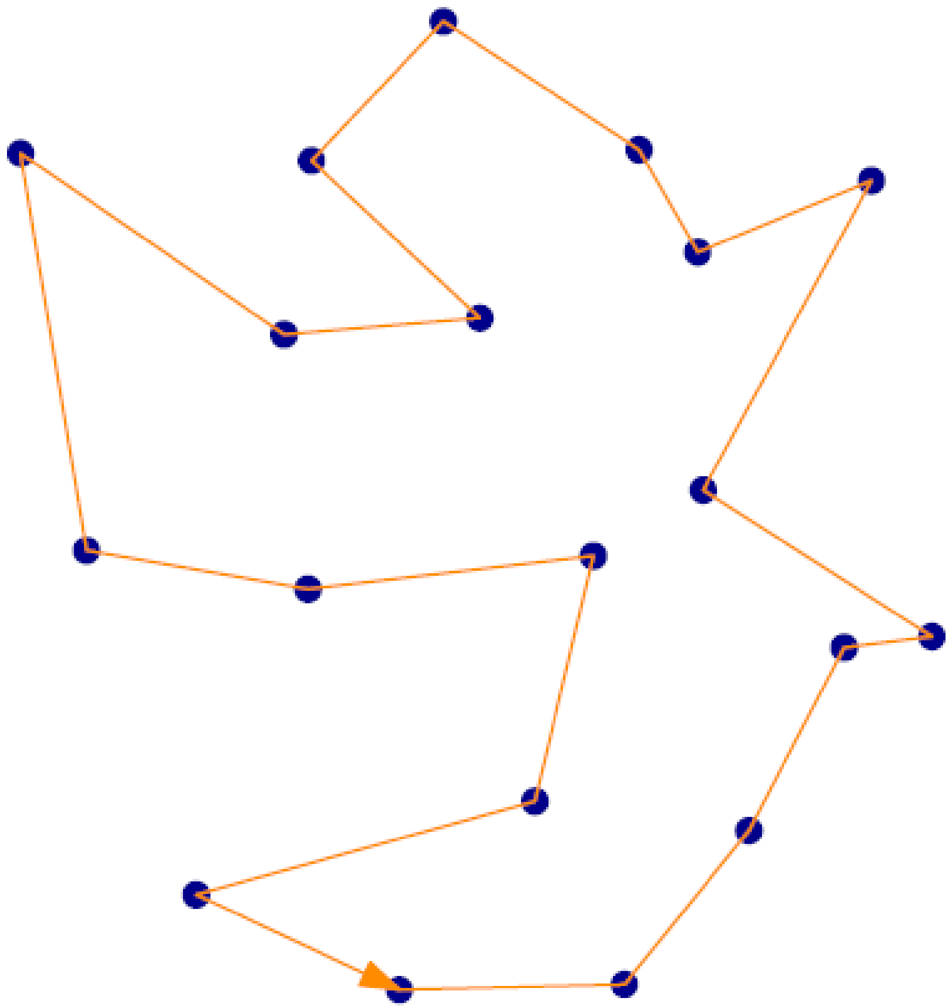
Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.



Best known algorithm needs roughly 2^n operations.

Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.

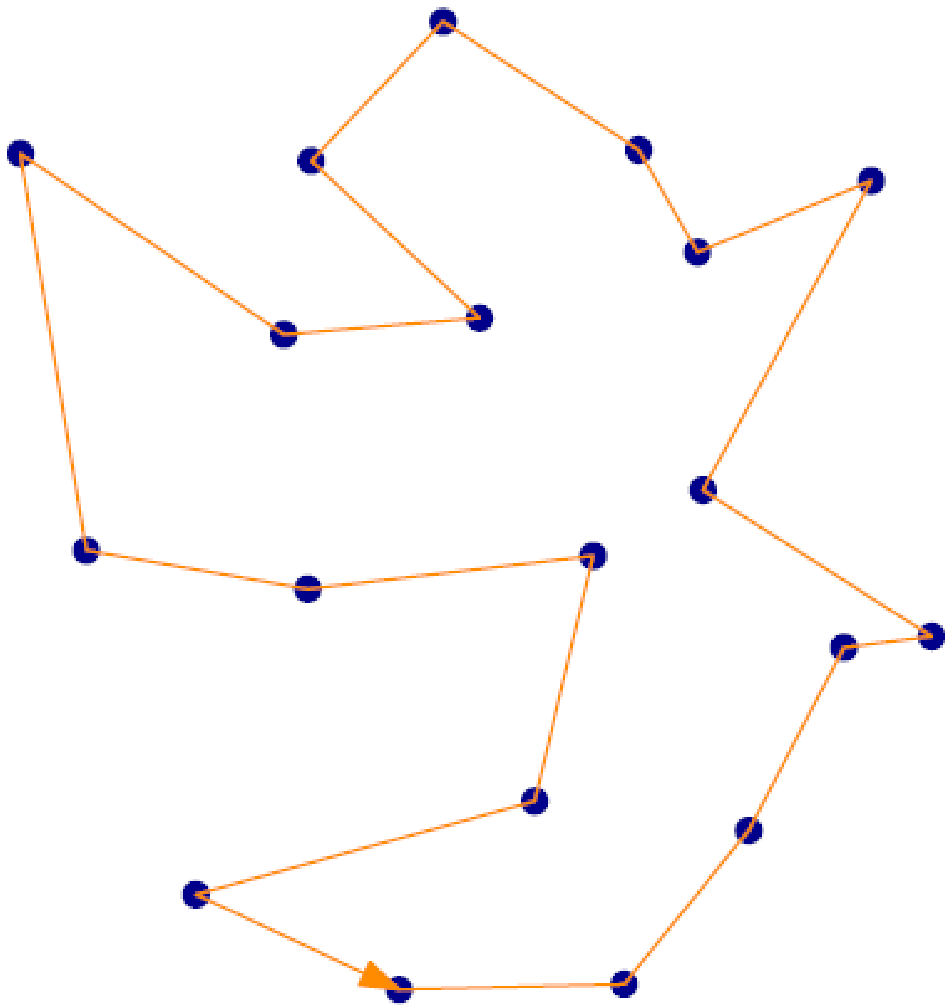


Best known algorithm needs roughly 2^n operations.

Nobody can prove that n^2 operations is impossible.

Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.



Best known algorithm needs roughly 2^n operations.

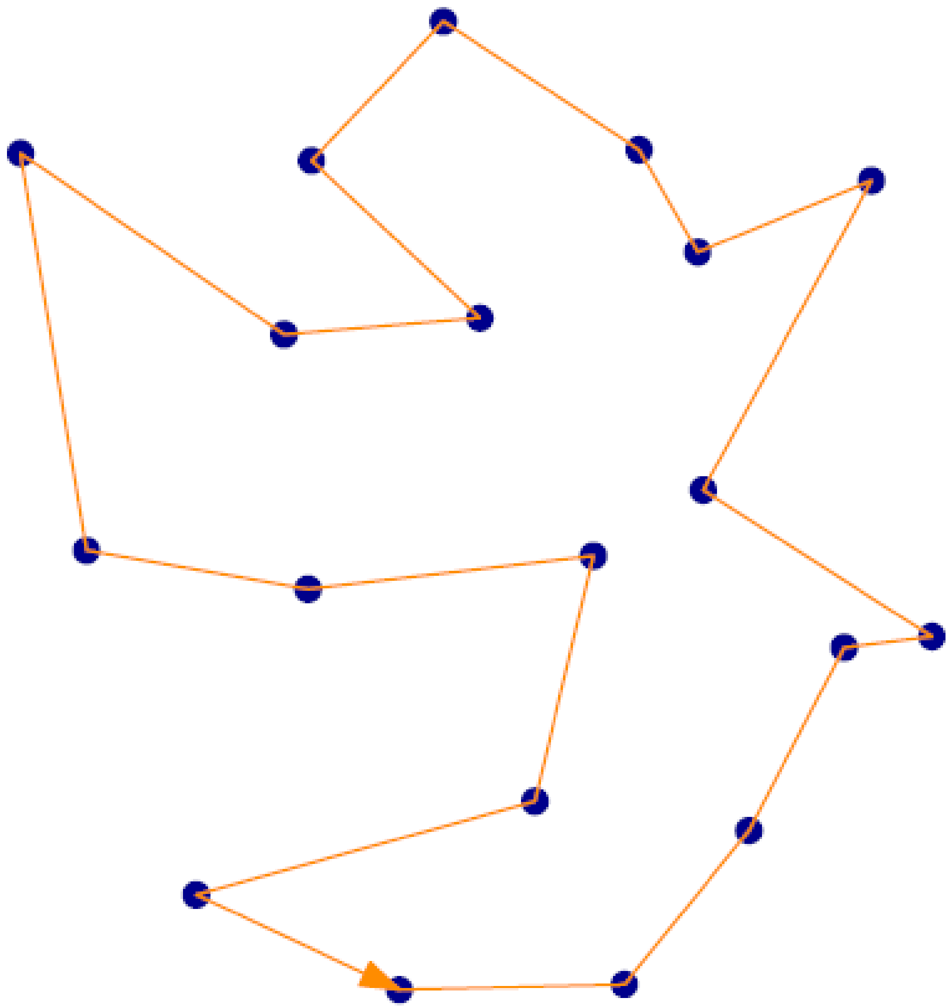
Nobody can prove that n^2 operations is impossible.

Million-dollar question:

$$P = NP?$$

Efficient algorithms for everything?

Travelling Salesman: Given n points in the plane, find the shortest tour that visits all the points.



Best known algorithm needs roughly 2^n operations.

Nobody can prove that n^2 operations is impossible.

Million-dollar question:

$$P = NP?$$

There are problems for which we can prove that no algorithm exists.

What have we learnt?

We learnt a language for expressing computations (Python).

What have we learnt?

We learnt a language for expressing computations (Python).

We learnt about the process of writing and debugging a program.

What have we learnt?

We learnt a language for expressing computations (Python).

We learnt about the process of writing and debugging a program.

We learnt about abstractions (data and functions).

What have we learnt?

We learnt a language for expressing computations (Python).

We learnt about the process of writing and debugging a program.

We learnt about abstractions (data and functions).

We learnt about breaking problems into smaller pieces, and testing parts of a program one-by-one.

What have we learnt?

We learnt a language for expressing computations (Python).

We learnt about the process of writing and debugging a program.

We learnt about abstractions (data and functions).

We learnt about breaking problems into smaller pieces, and testing parts of a program one-by-one.

Remember that you can program to find answers to questions.

Other CS Courses

CS109: Programming Practice Gain experience programming with some fun projects

CS202 Problem Solving Learn about methods for problem solving and algorithm development.

CS204 Discrete Mathematics Cover mathematical concepts frequently employed in computer science: sets, relations, propositional logic, etc.

CS206: Data Structures (Separate sections for CS / non-CS in Spring) Improve programming skills; learn to design, use, and implement abstract data types; and learn about a number of fundamental standard data structures and algorithms.