Introduction to Algorithms

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Representing Solutions

A primary aim of computer science is to solve problems using computers.

It is necessary to express those solutions both mathematically and in a way that can be executed by a computer.

We should also be able to analyze the performance of our solutions.

Representing Solutions

A primary aim of computer science is to solve problems using computers.

It is necessary to express those solutions both mathematically and in a way that can be executed by a computer.

We should also be able to analyze the performance of our solutions.

A solution may exist yet be infeasible to execute.

Representing Solutions

Analyzing the performance of an algorithm can mean many things.

- How much memory does it use?
- How much time does it take to execute?
- How much energy does it use?
- How much heat does it generate?
- How much bandwidth does it consume?
- How much does it cost to execute?

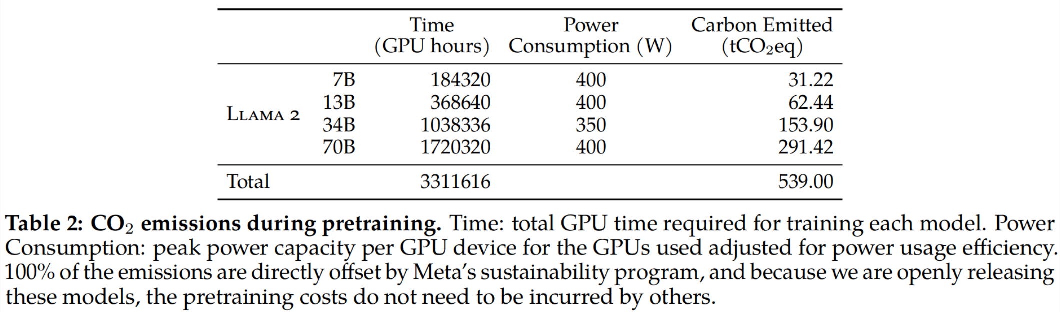
Representing Solutions

Meta released the second iteration of their Llama large language model in 2023.

The training time was such that they measured its impact in GPU hours.

They also report the tons of CO2 equivalent emitted during training.

Representing Solutions



		Time (GPU hours)	Power Consumption (W)	Carbon Emitted (tCO_2eq)
Llama 2	7B	184320	400	31.22
	13B	368640	400	62.44
	34B	1038336	350	153.90
	70B	1720320	400	291.42
Total		3311616		539.00

Table 2: CO_2 emissions during pretraining. Time: total GPU time required for training each model. Power Consumption: peak power capacity per GPU device for the GPUs used adjusted for power usage efficiency. 100% of the emissions are directly offset by Meta's sustainability program, and because we are openly releasing these models, the pretraining costs do not need to be incurred by others.

Source: `https://ai.meta.com/llama/`

Representing Solutions

An **algorithm** is a step-by-step procedure for solving a problem.

It can also be viewed as a way to transform a given input into a desired output.

Representing Solutions

How we represent the input and output of an algorithm is important.

Different representations, or **data structures**, call for different algorithms.

Insertion Sort

Sorting Algorithms

Sorting and searching are two fundamental and practical problems studied in computer science.

We will start our study of evaluating algorithms by looking at a simple sorting algorithm.

First, we must define what it means to sort a list of numbers.

Sorting Algorithms

Given a sequence of n objects

$$A = \langle a_1, a_2, \ldots, a_n \rangle,$$

a sorting algorithm rearranges the elements such that

$$A' = \langle a'_1, a'_2, \ldots, a'_n \rangle \text{ and } a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Sorting Algorithms

Under this definition, we assume that the elements of the set are comparable. It is enough that we can determine if $a_i \leq a_j$ for any i and j.

Some sets, such as the set of all real numbers $\mathbb{R}$, have a natural ordering.

Others, such as the set of all strings Σ^*, do not. In this case, we can define an ordering based on the lexicographic ordering of the strings.

Insertion Sort: Definition

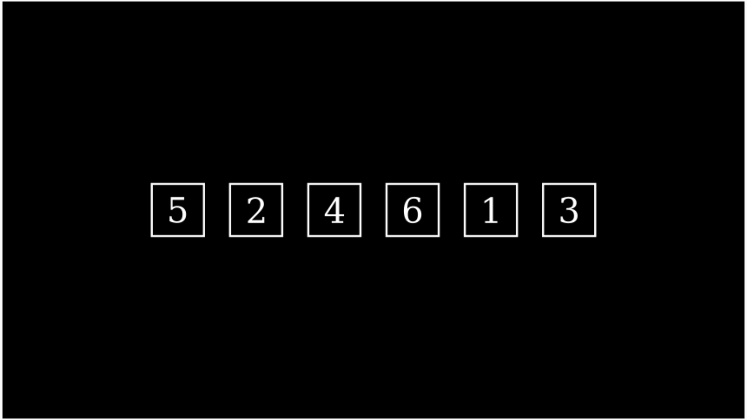
Insertion sort is defined in Python as

```python
def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Example: Sorting Numbers

Sorting Numbers

Let’s walk through the algorithm on the following list of numbers.



5 2 4 6 1 3

Figure 1: Original list of numbers.

Sorting Numbers

The algorithm starts by selecting the second element of the list as the key.

Sorting Numbers

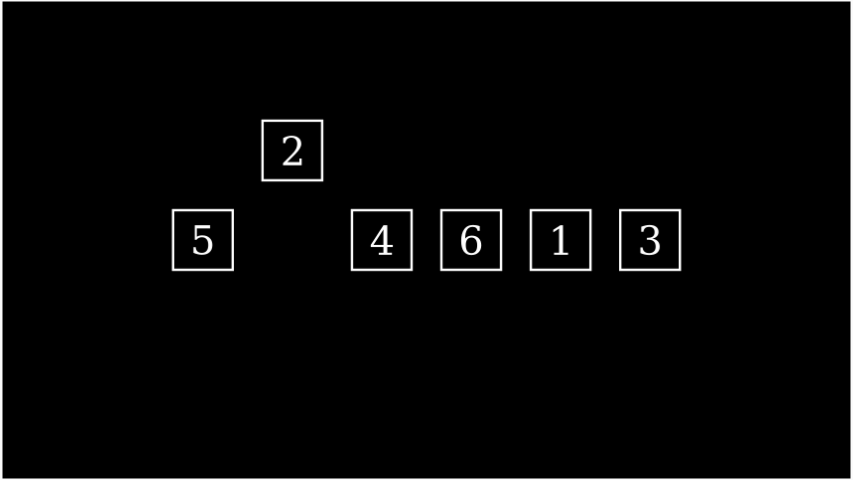


Figure 2: Element at index 1 is selected.

<title>

All values to the left of the key are checked to see if they are greater than the key.

If they are, they are moved to the right.

Only 1 element is to the left, and it is greater, so we will move it to the right.

Sorting Numbers

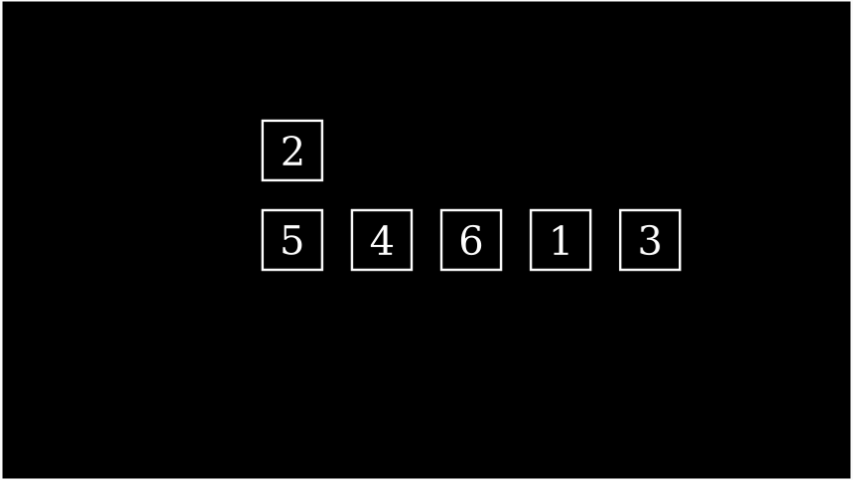


Figure 3: Element at index 0 is moved to index 1.

Sorting Numbers

There are no more elements to consider, so we insert the key at the current position.

Sorting Numbers

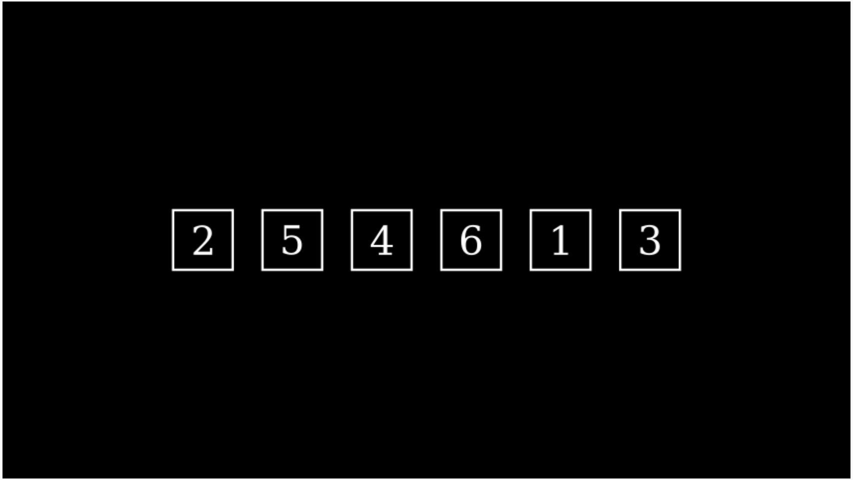


Figure 4: The key is moved to index 0.

Sorting Numbers

We return to the top of the outer loop and select the next element as they key.

In this case, *i* is 2 and the key is 4.

Sorting Numbers

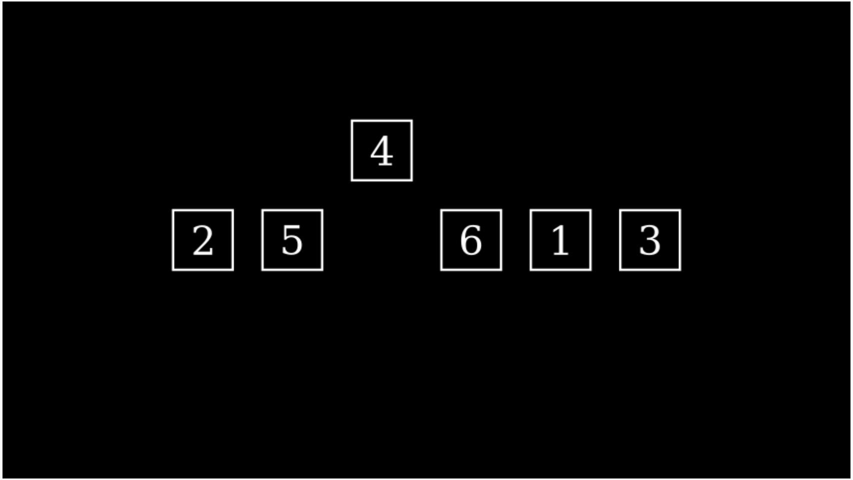


Figure 5: The key at index 2 is selected.

Sorting Numbers

In the inner loop, we compare the key to the element at index 1.

5 is greater than 4, so we move 5 to the right.

Sorting Numbers

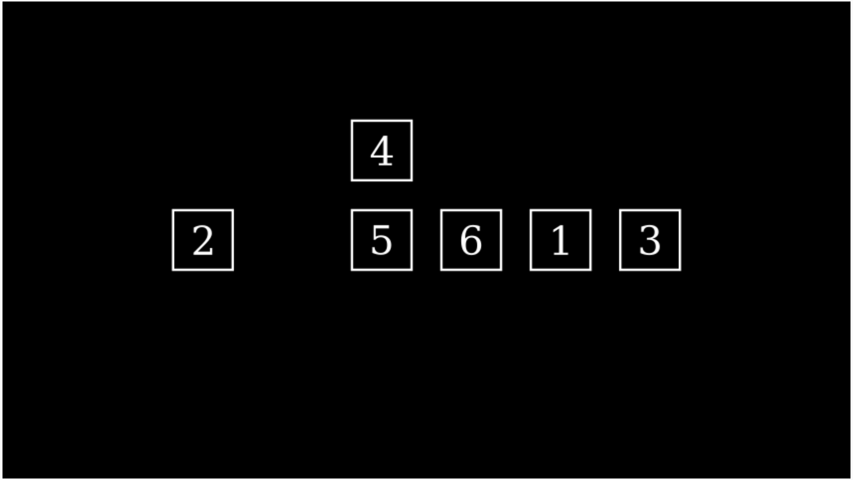


Figure 6: The element at index 1 is moved to the right.

Sorting Numbers

Next, the value 2 at index 0 is evaluated, but it is less than the key, so we stop and insert the key at index 1.

Sorting Numbers

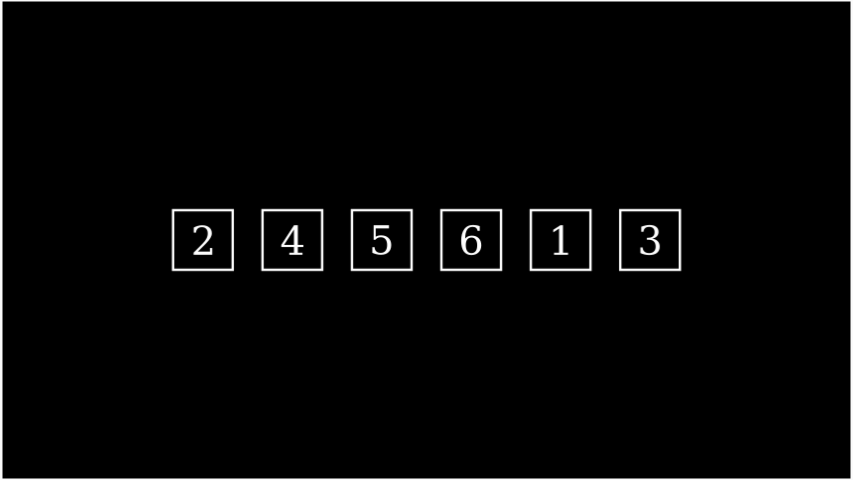


Figure 7: The key is inserted at index 1.

Sorting Numbers

The key is updated to be the value at index 3, which is 6.

We can see that it is already larger than the item to the left, so the statements in the inner loop are never executed.

Our list stays in the same relative order.

Sorting Numbers

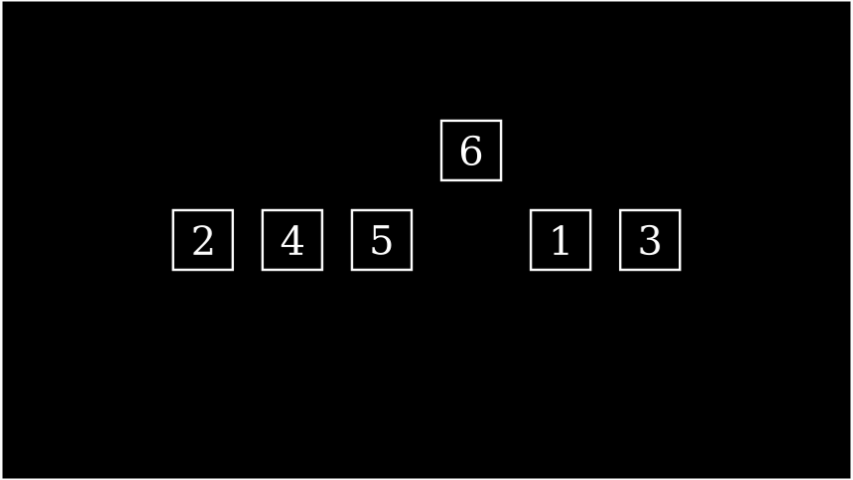


Figure 8: The key at index 3 is not moved.

Sorting Numbers

The outer loop updates the key to index 5, which is 1.

We can see that it is less than all values to the left, so the inner loop will shift each value to the right.

Sorting Numbers

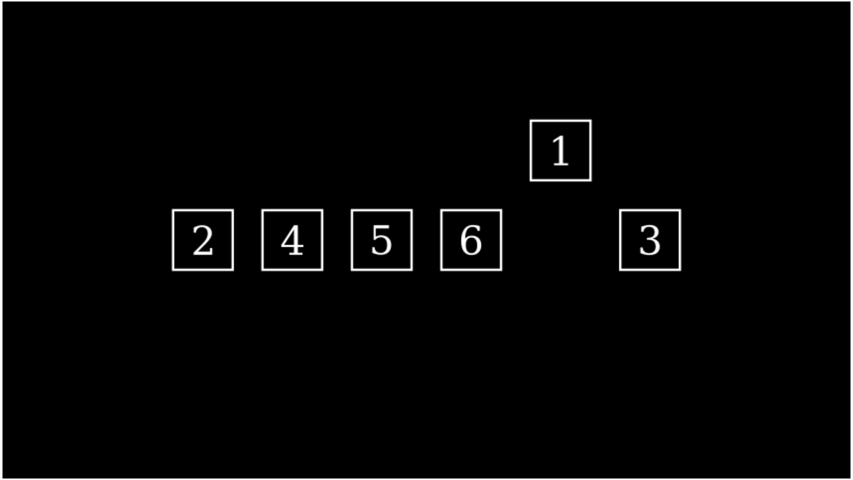


Figure 9: The key at index 4 is selected.

Sorting Numbers

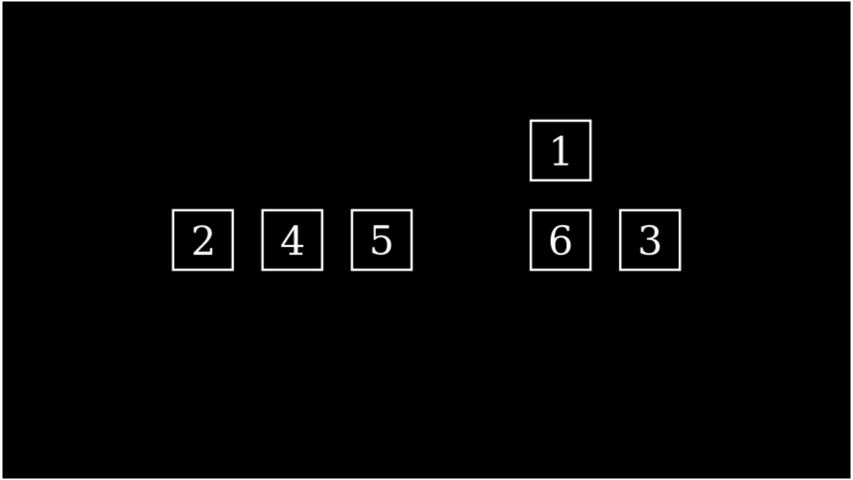


Figure 10: $6 > 1$, so 6 is moved to the right.

Sorting Numbers

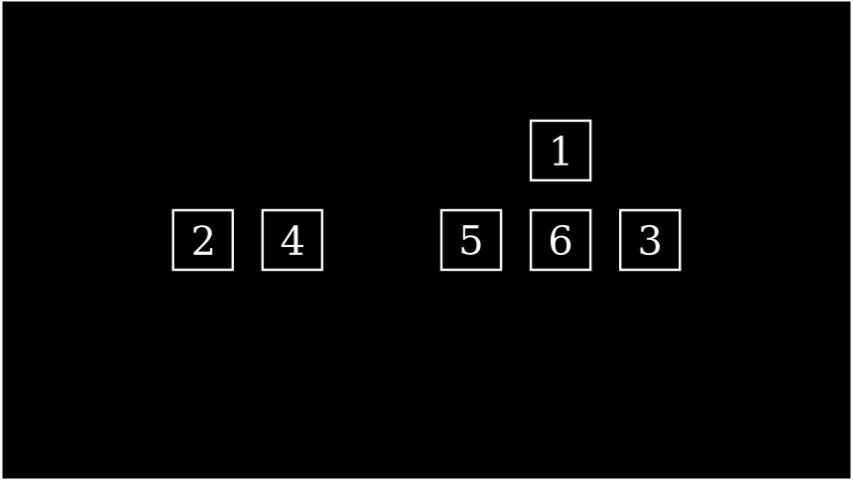


Figure 11: 5 $>$ 1, so 5 is moved to the right.

Sorting Numbers

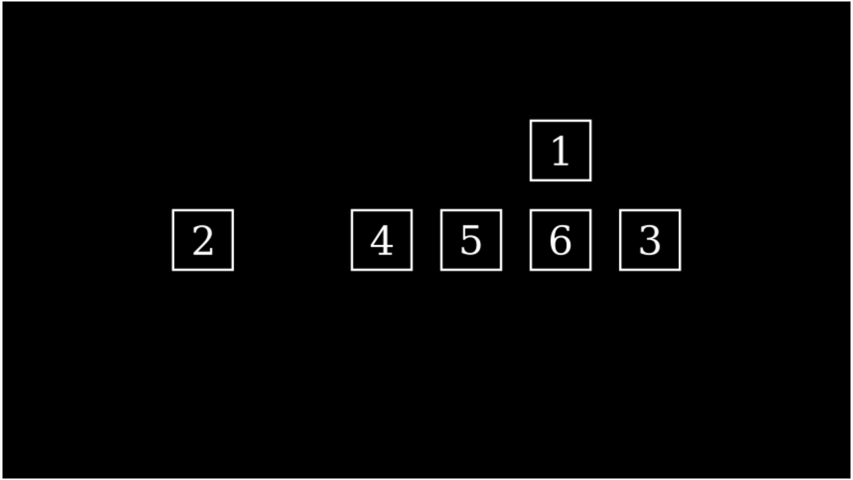


Figure 12: $4 > 1$, so 4 is moved to the right.

Sorting Numbers

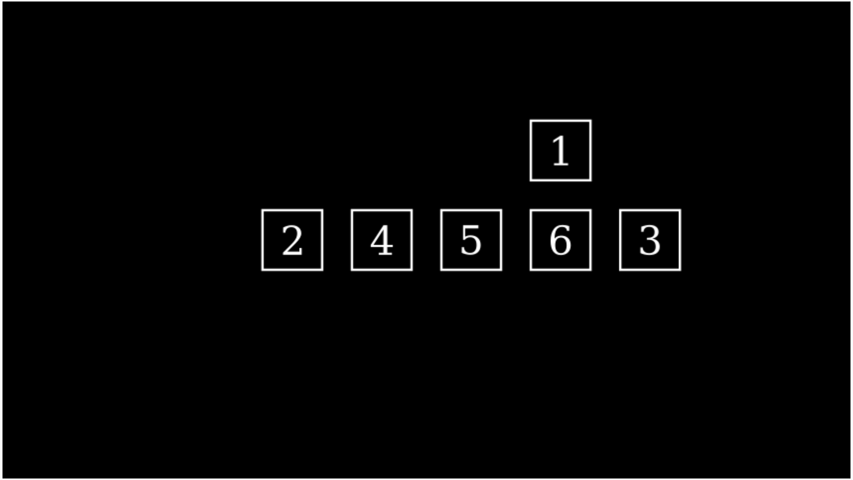


Figure 13: $2 > 1$, so 2 is moved to the right.

Sorting Numbers

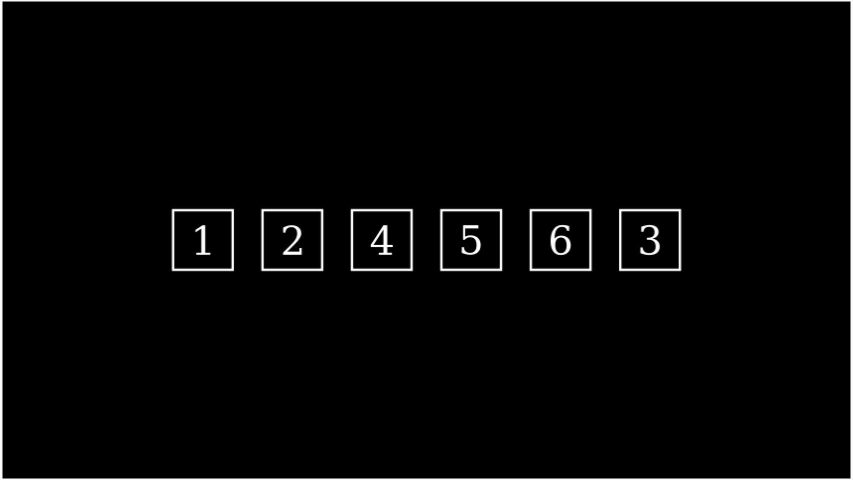


Figure 14: The value 1 is placed all the way at index 0.

Sorting Numbers

The last item in the list is selected as the final key.

The values 4, 5, and 6 are shift to the right and the key is inserted at index 2.

Sorting Numbers

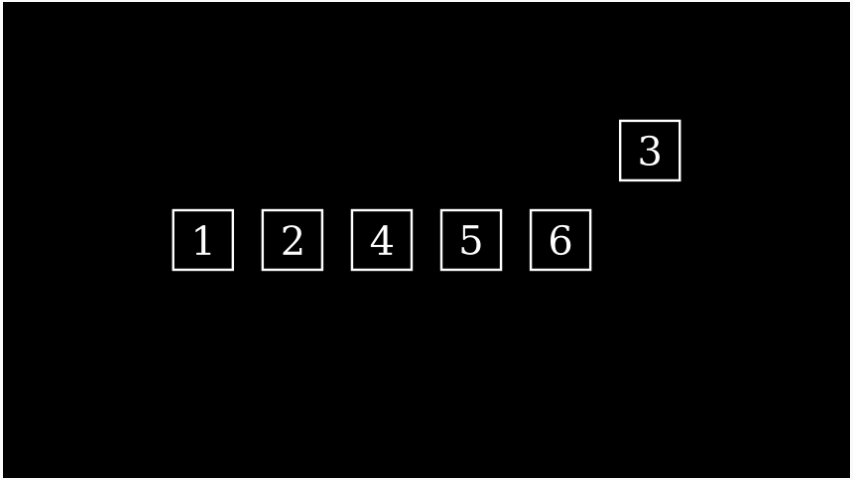


Figure 15: The last value is selected as the key.

Sorting Numbers

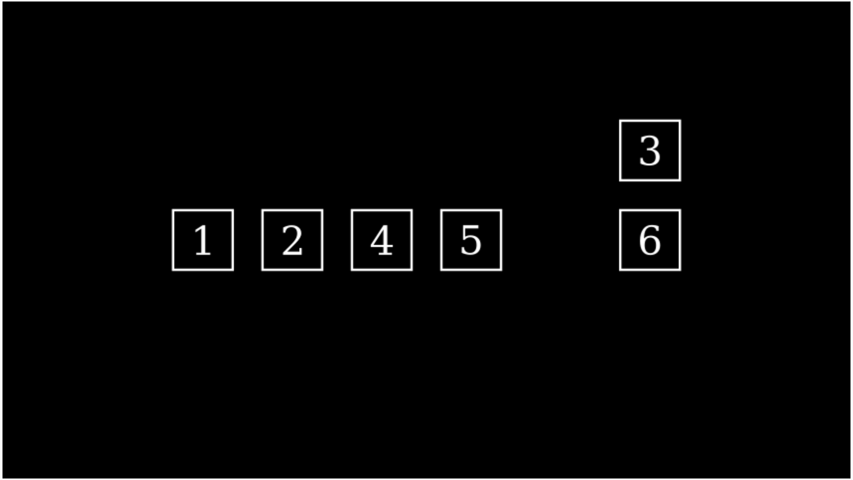


Figure 16: $6 > 3$, so 6 is moved to the right.

Sorting Numbers

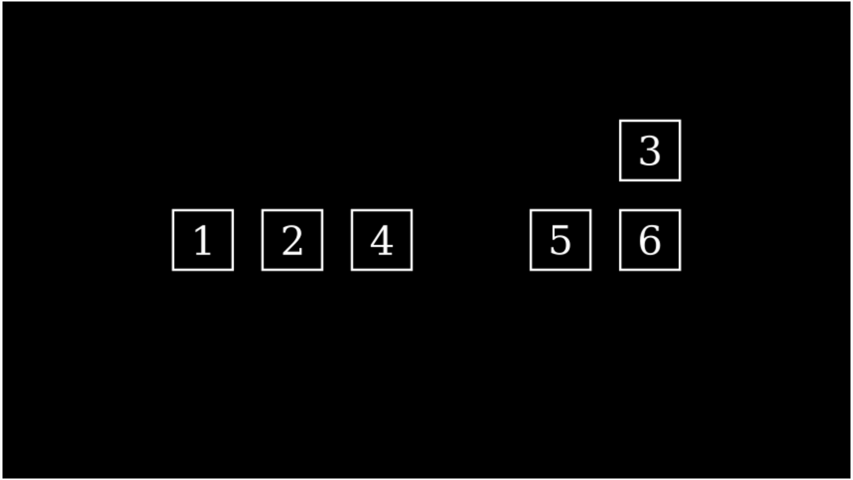


Figure 17: $5 > 3$, so 5 is moved to the right.

Sorting Numbers

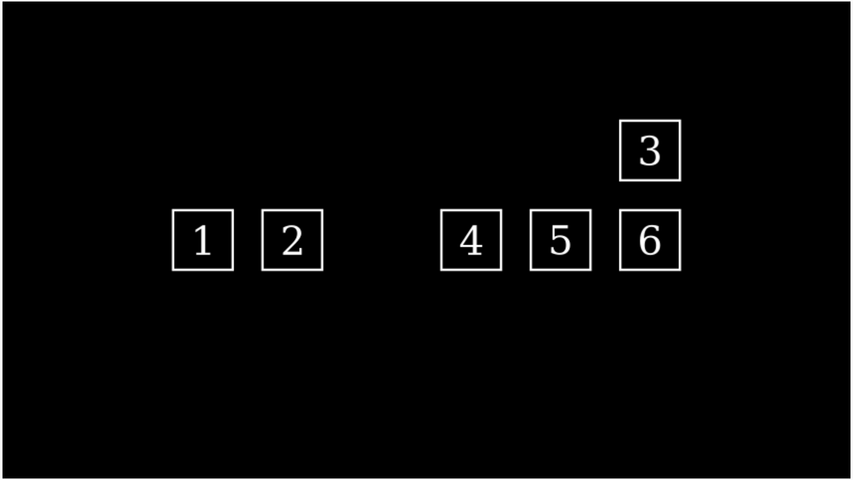


Figure 18: $4 > 3$, so 4 is moved to the right.

Sorting Numbers

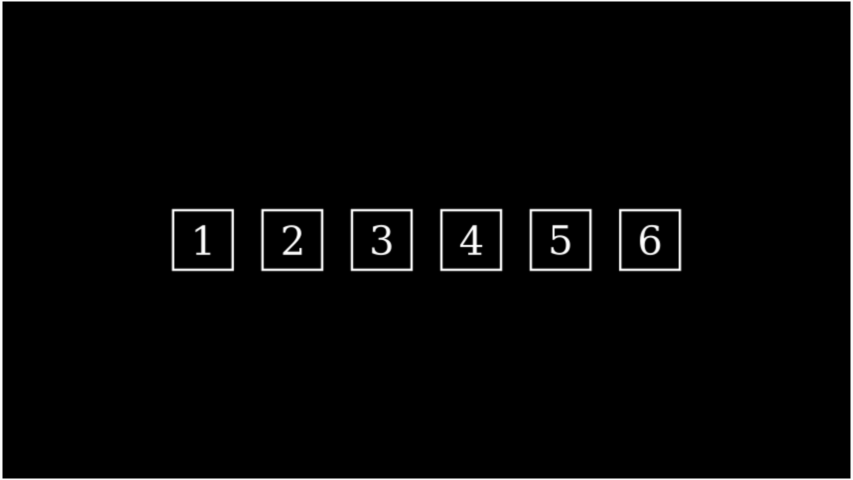


Figure 19: The last value is inserted at index 2, and the list is sorted.

Verifying Correctness

Correctness

How can we be sure that an algorithm works for all input sequences?

Correctness

How can we be sure that an algorithm works for all input sequences?

By verifying its correctness.

Correctness

Proving the correctness of an algorithm may be tricky in some cases, but we can utilize a few techniques to make it easier.

The first such technique is called a **loop invariant**: a statement that is true before and after each iteration of a loop.

Correctness

The loop invariant for insertion sort is that the subarray $A[0 : i - 1]$ is sorted.

In some cases, the loop invariant is clear from the problem statement. In others, it may require some thought.

It is also possible that more than one loop invariant exists for a given algorithm.

Loop Invariants

To determine if a loop invariant is correct, we must verify three things:

1. **Initialization**: The loop invariant is true before the first iteration of the loop.
2. **Maintenance**: If the loop invariant is true before an iteration of the loop, it remains true after the iteration.
3. **Termination**: When the loop terminates, the loop invariant is true.

If these properties hold, the algorithm is correct.

Verifying Insertion Sort

Let's verify the correctness of insertion sort.

Initialization

- Before the first iteration of the loop, $i = 1$.
- The subarray $A[0 : i - 1] = A[0 : 0] = \langle A[0] \rangle$.
- Since a single element is always sorted, the loop invariant is true before the first iteration.

Verifying Insertion Sort

Maintenance

Assume that the loop invariant is true before the ith iteration of the loop: $A[0:i-1]$ is sorted.

We need to show that the loop invariant remains true after the $i+1$th iteration.

- The $i+1$th iteration of the loop will swap elements in the subarray $A[0:i]$.
- The loop invariant is maintained if the subarray $A[0:i]$ is sorted after the $i+1$th iteration.
- This is true because the $i+1$th iteration will swap elements in the subarray $A[0:i]$ until the element at index i is in the correct position.

Characterizing the Running Time

Characterizing the Running Time

We can characterize the running time of an algorithm by counting the number of operations it performs.

Since the exact execution time of each atomic statement will be different depending on the hardware, we will assign each one a constant value.

We will see later on that the exact value of the constant does not matter since the algorithms are compared based on their rate of growth.

Characterizing the Running Time

```
1 def insertion_sort(A):
2     for i in range(1, len(A)): # ?
3         key = A[i]
4         j = i - 1
5         while j >= 0 and A[j] > key:
6             A[j + 1] = A[j]
7             j = j - 1
8         A[j + 1] = key
```

How many times is line 2 evaluated for an input of size *n*?

Characterizing the Running Time

```
1 def insertion_sort(A):
2     for i in range(1, len(A)): # $c_1n$
3         key = A[i]
4         j = i - 1
5         while j >= 0 and A[j] > key:
6             A[j + 1] = A[j]
7             j = j - 1
8         A[j + 1] = key
```

The first statement on line 2 is evaluated n times at a cost of c_1 each time.

Characterizing the Running Time

```
1 def insertion_sort(A):
2     for i in range(1, len(A)): # $c_1 n$
3         key = A[i] # ?
4         j = i - 1 # ?
5         while j >= 0 and A[j] > key:
6             A[j + 1] = A[j]
7             j = j - 1
8         A[j + 1] = key # ?
```

How many times are lines 3, 4, and 8 evaluated for an input of size n?

Characterizing the Running Time

```
def insertion_sort(A):
    for i in range(1, len(A)): # $c_1 n$
        key = A[i] # $c_2(n-1)$
        j = i - 1 # $c_3(n-1)$
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key # $c_7(n-1)$
```

Before evaluating the inner loop, we can assign constant values to the other statements. These are only executed $n-1$ times.

Characterizing the Running Time

The inner `while` loop requires a bit of thought to analyze.

The number of times it is executed depends on the value of j as well as the relative order of the elements to the left of the key.

If we acknowledge this value to be variable, we can represent it as t_i, where i represents the current step of the outer loop.

Characterizing the Running Time

```
1 def insertion_sort(A):
2     for i in range(1, len(A)): # $c_1 n$
3         key = A[i] # $c_2(n-1)$
4         j = i - 1 # $c_3(n-1)$
5         while j >= 0 and A[j] > key: # $c_4 \sum_{i=2}^{n} t_i$
6             A[j + 1] = A[j]
7             j = j - 1
8         A[j + 1] = key # $c_7(n-1)$
```

If each evaluation of line 5 costs c_4, then the cost of the inner loop is $c_4 t_i$, where t_i is the number of times the loop expression is evaluated.

This is evaluated $n-1$ times, so the total cost of the inner loop is $c_4 \sum_{i=2}^{n} t_i$.

Characterizing the Running Time

IMPORTANT: Convert between 0-based and 1-based indexing.

The loop expressed in mathematical notation is $\sum_{i=2}^{n} t_i$, but the loop in Python is `for i in range(1, len(A))`.

Notation in mathematics usually follows a 1-based indexing scheme, while most programming languages use 0-based indexing.

We should practice converting between the two.

Characterizing the Running Time

```
def insertion_sort(A):
    for i in range(1, len(A)):          # $c_1 n$
        key = A[i]                      # $c_2(n-1)$
        j = i - 1                       # $c_3(n-1)$
        while j >= 0 and A[j] > key:    # $c_4 \sum_{i=2}^{n} t_i$
            A[j + 1] = A[j]             # $c_5 \sum_{i=2}^{n}(t_i - 1)$
            j = j - 1                   # $c_6 \sum_{i=2}^{n}(t_i - 1)$
        A[j + 1] = key                  # $c_7(n-1)$
```

The statements inside the inner loop are executed $t_i - 1$ times, so the total cost of the inner loop is $\sum_{i=2}^{n} t_i - 1$ multiplied by their individual execution costs.

Characterizing the Running Time

The total cost of the algorithm is the sum of the costs of each statement.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n}(t_i - 1) + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7(n-1)$$

Worst-case Analysis

This analysis is a good start, but it doesn’t paint the whole picture.

The number of actual executions will depend on the input that is given.

For example, what if the input is already sorted, or given in reverse order?

It is common to express the worst-case runtime for a particular algorithm.

Worst-case Analysis

For insertion sort, that is when the input is in reverse order.

In this case, each element $A[i]$ is compared to every other element in the sorted subarray.

This means that $t_i = i$ for every iteration of the `for` loop.

The worst-case runtime is given as

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} i + c_5 \sum_{i=2}^{n}(i-1) + c_6 \sum_{i=2}^{n}(i-1) + c_7(n-1)$$

Worst-case Analysis

To express this runtime solely in terms of n, we can use the fact that

$$\sum_{i=2}^{n} i = (\sum_{i=1}^{n} i) - 1 = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Worst-case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\
&= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)
\end{aligned}
$$

Worst-case Analysis

The terms in parentheses are constants, so we can simplify the expression to

$$T(n) = an^2 + bn + c$$

where a, b, and c are constants.

Best-case Analysis

Best-case Analysis

The best-case runtime for insertion sort is when the input is already sorted.

In this case, the `while` check is executed only once per iteration of the `for` loop.

That is, $t_i = 1$ for every iteration of the `for` loop.

Best-case Analysis

The best-case runtime is given as

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

Best-case Analysis

Let $a = c_1 + c_2 + c_3 + c_4 + c_7$ and $b = -(c_2 + c_3 + c_4 + c_7)$

Then the best-case runtime is given as $an + b$, a linear function of n.

Rate of Growth

Rate of Growth

We can simplify how we express the runtime of both these cases by considering only the highest-order term.

Consider the worst-case, $T(n) = an^2 + bn + c$.

As n grows, the term an^2 will dominate the runtime, rendering the others insignificant by comparison.

Rate of Growth

This simplification is typically expressed using Θ notation.

For the worst-case, we say that $T(n) = \Theta(n^2)$.

It is a compact way of stating that the runtime is proportional to n^2 for large values of n.

Class Exercise: Analysis of Selection Sort

Analysis of Selection Sort

Based on the analysis above, let's check our understanding and see if we can characterize the runtime of another sorting algorithm, selection sort.

```python
def selection_sort(A):
    for i in range(0, len(A) - 1):
        min_index = i
        for j in range(i + 1, len(A)):
            if A[j] < A[min_index]:
                min_index = j
        A[i], A[min_index] = A[min_index], A[i]
```

Analysis of Selection Sort

```
1 def selection_sort(A):
2     for i in range(0, len(A) - 1):
3         min_index = i
4         for j in range(i + 1, len(A)):
5             if A[j] < A[min_index]:
6                 min_index = j
7         A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 2 evaluated for an input of size n?

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                  # $c_1 n$
3          min_index = i
4          for j in range(i + 1, len(A)):
5              if A[j] < A[min_index]:
6                  min_index = j
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 2 evaluated for an input of size *n*?

n times, since it must be evaluated a final time to check if the loop limit is reached.

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                    # $c_1n$
3          min_index = i
4          for j in range(i + 1, len(A)):
5              if A[j] < A[min_index]:
6                  min_index = j
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 3 evaluated for an input of size n?

Analysis of Selection Sort

```
1 def selection_sort(A):
2     for i in range(0, len(A) - 1):                    # $c_1 n$
3         min_index = i                                  # $c_2(n-1)$
4         for j in range(i + 1, len(A)):
5             if A[j] < A[min_index]:
6                 min_index = j
7         A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 3 evaluated for an input of size n?

$n - 1$ times, since that is the number of times the outer loop is traversed.

Analysis of Selection Sort

```
1 def selection_sort(A):
2     for i in range(0, len(A) - 1):                  # $c_1 n$
3         min_index = i                               # $c_2(n-1)$
4         for j in range(i + 1, len(A)):
5             if A[j] < A[min_index]:
6                 min_index = j
7         A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 4 evaluated for an input of size n?

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):               # $c_1 n$
3          min_index = i                            # $c_2 (n-1)$
4          for j in range(i + 1, len(A)):           # $c_3 \sum_{i=1}^{n-1} (n-i+1)$
5              if A[j] < A[min_index]:
6                  min_index = j
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 4 evaluated for an input of size n?

$n - i$ times. Unlike insertion sort, the inner loop is not dependent on an additional condition.

Since this is executed for each iteration of the other loop, the total is $\sum_{i=1}^{n-1} (n-i+1)$.

Analysis of Selection Sort

```
1 def selection_sort(A):
2     for i in range(0, len(A) - 1):                  # $c_1 n$
3         min_index = i                               # $c_2 (n - 1)$
4         for j in range(i + 1, len(A)):              # $c_3 \sum_{i=1}^{n-1} (n - i + 1)$
5             if A[j] < A[min_index]:
6                 min_index = j
7         A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 5 evaluated for an input of size n?

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                   # $c_1 n$
3          min_index = i                                 # $c_2(n-1)$
4          for j in range(i + 1, len(A)):                # $c_3 \sum_{i=1}^{n-1}(n-i+1)$
5              if A[j] < A[min_index]:                   # $c_4 \sum_{i=1}^{n-1}(n-i)$
6                  min_index = j
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 5 evaluated for an input of size n?

It is executed 1 less than the number of times line 4 is executed, so the total is $\sum_{i=1}^{n-1}(n-i)$.

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                    # $c_1 n$
3          min_index = i                                 # $c_2(n-1)$
4          for j in range(i + 1, len(A)):                # $c_3 \sum_{i=1}^{n-1}(n-i+1)$
5              if A[j] < A[min_index]:                   # $c_4 \sum_{i=1}^{n-1}(n-i)$
6                  min_index = j
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 6 evaluated for an input of size *n*?

This one is conditioned on line 5 being true!

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                        # $c_1 n$
3          min_index = i                                     # $c_2(n-1)$
4          for j in range(i + 1, len(A)):                    # $c_3 \sum_{i=1}^{n-1}(n-i+1)$
5              if A[j] < A[min_index]:                       # $c_4 \sum_{i=1}^{n-1}(n-i)$
6                  min_index = j                             # $c_5 \sum_{i=1}^{n-1} t_i$
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 6 evaluated for an input of size n?

Since it is conditional, the number of times is runs is dependent on the exact input. In this case we can represent it with a veriable t_i.

The total is then $\sum_{i=1}^{n-1} t_i$.

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                # $c_1 n$
3          min_index = i                             # $c_2 (n - 1)$
4          for j in range(i + 1, len(A)):            # $c_3 \sum_{i=1}^{n-1} (n - i + 1)$
5              if A[j] < A[min_index]:               # $c_4 \sum_{i=1}^{n-1} (n - i)$
6                  min_index = j                     # $c_5 \sum_{i=1}^{n-1} t_i$
7          A[i], A[min_index] = A[min_index], A[i]
```

How many times is line 7 evaluated for an input of size n?

Analysis of Selection Sort

```
1  def selection_sort(A):
2      for i in range(0, len(A) - 1):                    # $c_1 n$
3          min_index = i                                 # $c_2 (n-1)$
4          for j in range(i + 1, len(A)):                # $c_3 \sum_{i=1}^{n-1} (n-i+1)$
5              if A[j] < A[min_index]:                   # $c_4 \sum_{i=1}^{n-1} (n-i)$
6                  min_index = j                         # $c_5 \sum_{i=1}^{n-1} t_i$
7          A[i], A[min_index] = A[min_index], A[i]       # $c_6 (n-1)$
```

How many times is line 7 evaluated for an input of size n?

It is evaluated the same number of times as line 3, so the total is $c_6(n-1)$.

Analysis of Selection Sort

Combining these results, we can express the runtime of selection sort as

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + c_4 \sum_{i=1}^{n-1}(n-i) + c_5 \sum_{i=1}^{n-1} t_i + c_6(n-1)$$

Analysis of Selection Sort

Now that we have a general analysis of the running time, let's look at the worst-case scenario.

Just like insertion sort, the worst-case scenario for selection sort is when the input is given in reverse order.

In this case, line 6 is executed as many times as line 5.

This yields $t_i = n - i$ for every iteration of the outer loop.

Analysis of Selection Sort

The runtime for the worst-case scenario is

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + c_4 \sum_{i=1}^{n-1}(n-i) + c_5 \sum_{i=1}^{n-1}(n-i) + c_6(n-1)$$

Analysis of Selection Sort

We can again use the arithmetic series to simplify the expression, where

$$\sum_{i=1}^{n-1}(n-i+1) = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Analysis of Selection Sort

Our running time for the worst-case is then simplified to

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3\left(\frac{n(n+1)}{2} - 1\right) + c_4\left(\frac{n(n-1)}{2}\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6(n-1) \\ &= \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6\right) n - (c_2 + c_3 + c_6) \end{aligned}$$

Analysis of Selection Sort

We can simplify this further by letting $a = \frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}$ and $b = c_1 + c_2 + c_3 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6$ and $c = -(c_2 + c_3 + c_6)$.

$$T(n) = an^2 + bn + c$$

Using theta-notation, this can be expressed as

$$T(n) = \Theta(n^2)$$