CSC 225 - Summer 2019

Algorithm Analysis I

Bill Bird

Department of Computer Science
University of Victoria

May 8, 2019

Algorithms (1)

An **algorithm** is a precise and unambiguous sequence of instructions to perform a clearly defined task with finite resources. The instructions must be specified in enough detail that no creativity or cleverness is required to understand them or carry them out.

Algorithms (2)

An **algorithm** is a **precise and unambiguous** sequence of instructions to perform a clearly defined task with finite resources. The instructions must be specified in enough detail that no creativity or cleverness is required to understand them or carry them out.

- The definition above captures the meaning of the word 'algorithm' as it applies to this course.
- If any steps are ambiguous or lack detail, it is impossible to guarantee a correct result.

Algorithms (3)

An **algorithm** is a precise and unambiguous sequence of instructions to perform a **clearly defined task** with finite resources. The instructions must be specified in enough detail that no creativity or cleverness is required to understand them or carry them out.

- Clearly defining the problem can be the most difficult part of algorithm design.
- Working with crisp, formal problem definitions is a luxury that does not often occur in practice.

Algorithms (4)

An **algorithm** is a precise and unambiguous sequence of instructions to perform a clearly defined task with **finite resources**. The instructions must be specified in enough detail that no creativity or cleverness is required to understand them or carry them out.

- Algorithms must complete within a finite amount of time, using a finite amount of space.
- It is acceptable for algorithms to require exorbitant amounts of resources (e.g. 1 million years of running time or 10^{100} bytes of space), as long as the requirements are finite.

Algorithms (5)

An **algorithm** is a precise and unambiguous sequence of instructions to perform a clearly defined task with **finite resources**. The instructions must be specified in enough detail that no creativity or cleverness is required to understand them or carry them out.

- Flippant Example: Consider the following "algorithm" to sort a deck of cards.

  while the deck is not sorted **do**
  Throw the cards on the floor.
  Pick them up in a random order.
  end while
- This process may never finish, so it is not an algorithm.

Algorithms (6)

An **algorithm** is a precise and unambiguous sequence of instructions to perform a clearly defined task with finite resources. The instructions must be specified in enough detail that **no creativity or cleverness is required** to understand them or carry them out.

- Creativity and cleverness are crucial parts of the algorithm *design* process.
- However, the algorithm itself should require no intellect to carry out, besides the ability to follow instructions.

Algorithm Design Problems

Task: Develop algorithms to solve the problems below.

PairSum225

Input: An array A of n positive integers.

Output: `true` if there exist indices i and j such that

$$A[i] + A[j] = 225$$

`false` otherwise.

ContainsDuplicate

Input: An array A of n integers.

Output: `true` if there exist indices i and j such that $i \neq j$ and

$$A[i] = A[j]$$

`false` otherwise.

PairSum225 (1)

Input Array	Result	Pair
50, 150, 75, 50	`true`	$150 + 75 = 225$
150, 125, 100, 175	`true`	$150 + 75 = 225$
1, 200, 100, 225	`false`	(none)
50, 25, 75, 25, 75	`false`	(none)
225, 225, 500, 1000	`false`	(none)
225, 0, 1, 5000	`true`	$0 + 225 = 225$
224, 0, 2, 2, 5, 2	`false`	(none)
224, 1, 0, 2, 2, 5	`true`	$1 + 224 = 225$

PairSum225 is an example of a **decision problem**, since the problem is to decide whether a pair exists, not necessarily find a specific example of a pair. However, explicitly finding a pair is an easy way to prove that one exists.

PairSum225 (2)

```
public static boolean PairSum225(int[] A){
    int n = A.length;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (A[i] + A[j] == 225)
                return true;
    return false;
}
```

A simple algorithm uses nested loops to check every pair.

The algorithm can be represented concisely in Java...

PairSum225 (3)

```
int PairSum225(int* A, int n){
    int i,j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (A[i] + A[j] == 225)
                return 1;
    return 0;
}
```

...or C

PairSum225 (4)

```
def PairSum225(A):
    n = len(A)
    for x in A:
        for y in A:
            if x + y == 225:
                return True
    return False
```

...or Python

The algorithm itself is not language-specific, but using a particular language to describe it might introduce a dependency on the peculiar characteristics of that language. For example, how does a Python list work internally?

PairSum225 (5)

```
procedure PairSum225(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if A[i] + A[j] = 225 then
                return True
            end if
        end for
    end for
    return False
end procedure
```

For the purposes of designing and studying algorithms, it is often better to write algorithms in pseudocode rather than a programming language.

PairSum225 (6)

```
procedure PairSum225(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if A[i] + A[j] = 225 then
                return True
            end if
        end for
    end for
    return False
end procedure
```

By using pseudocode, we can focus on the parts of the algorithm that are important, and ensure that the algorithm is clearly presented.

ContainsDuplicate (1)

Input Array	Result	Duplicate Elements
50, 150, 75, 50	`true`	50
150, 125, 100, 175	`false`	(none)
1, 200, 100, 225	`false`	(none)
50, 25, 75, 25, 75	`true`	25 and 75
225, 225, 500, 1000	`true`	225
225, 0, 1, 5000	`false`	(none)
224, 0, 2, 2, 5, 2	`true`	2
224, 1, 0, 2, 2, 5	`true`	2

The ContainsDuplicate problem also searches the array for particular pairs of values.

ContainsDuplicate (2)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if i ≠ j and A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

The simplest algorithm uses two nested loops, similar to the solution to PairSum225.

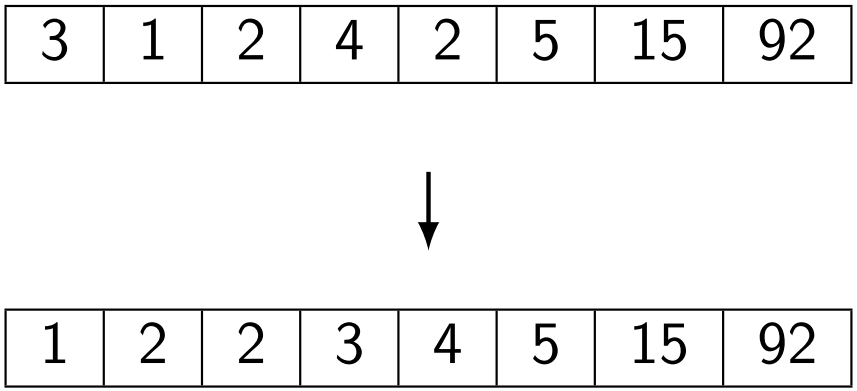
ContainsDuplicate (3)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = i + 1, i + 2, ..., n − 1 do
            if A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

Observation: We can start the nested loop at $i + 1$, instead of 0, since it is not necessary to check both the pair $(A[i], A[j])$ and the pair $(A[j], A[i])$.

Applying this refinement gives the improved algorithm above.

ContainsDuplicate (4)



Observation: If the input array is sorted, duplicate elements will be grouped together.

Duplicates in a sorted array can be detected easily with a single loop.

ContainsDuplicate (5)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    Sort A
    for i = 0, 1, 2, ..., n − 2 do
        if A[i] = A[i + 1] then
            return True
        end if
    end for
    return False
end procedure
```

We can use the observation on the previous slide to create another algorithm for ContainsDuplicate.

Question: How can we compare the three algorithms to determine which one is the fastest?

Algorithms for ContainsDuplicate (1)

Nested Loops (Original)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if i ≠ j and A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

Nested Loops (Improved)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = i + 1, i + 2, ..., n − 1 do
            if A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

Sort and Scan

```
procedure ContainsDuplicate(A)
    n ← Length of A
    Sort A
    for i = 0, 1, 2, ..., n − 2 do
        if A[i] = A[i + 1] then
            return True
        end if
    end for
    return False
end procedure
```

We want to compare the performance of the three algorithms and choose the 'best' one.

Algorithms for ContainsDuplicate (2)

Nested Loops (Original)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if i ≠ j and A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

Nested Loops (Improved)

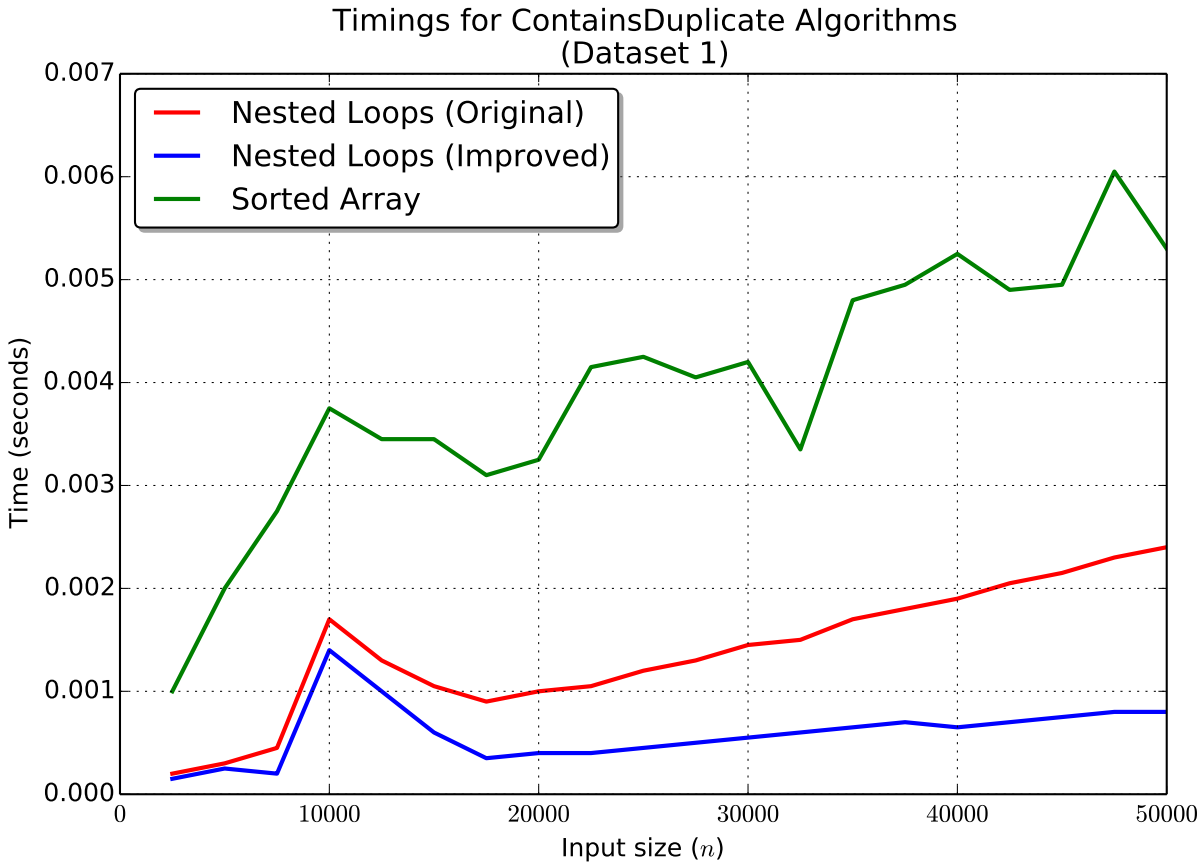
```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = i + 1, i + 2, ..., n − 1 do
            if A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

Sort and Scan

```
procedure ContainsDuplicate(A)
    n ← Length of A
    Sort A
    for i = 0, 1, 2, ..., n − 2 do
        if A[i] = A[i + 1] then
            return True
        end if
    end for
    return False
end procedure
```

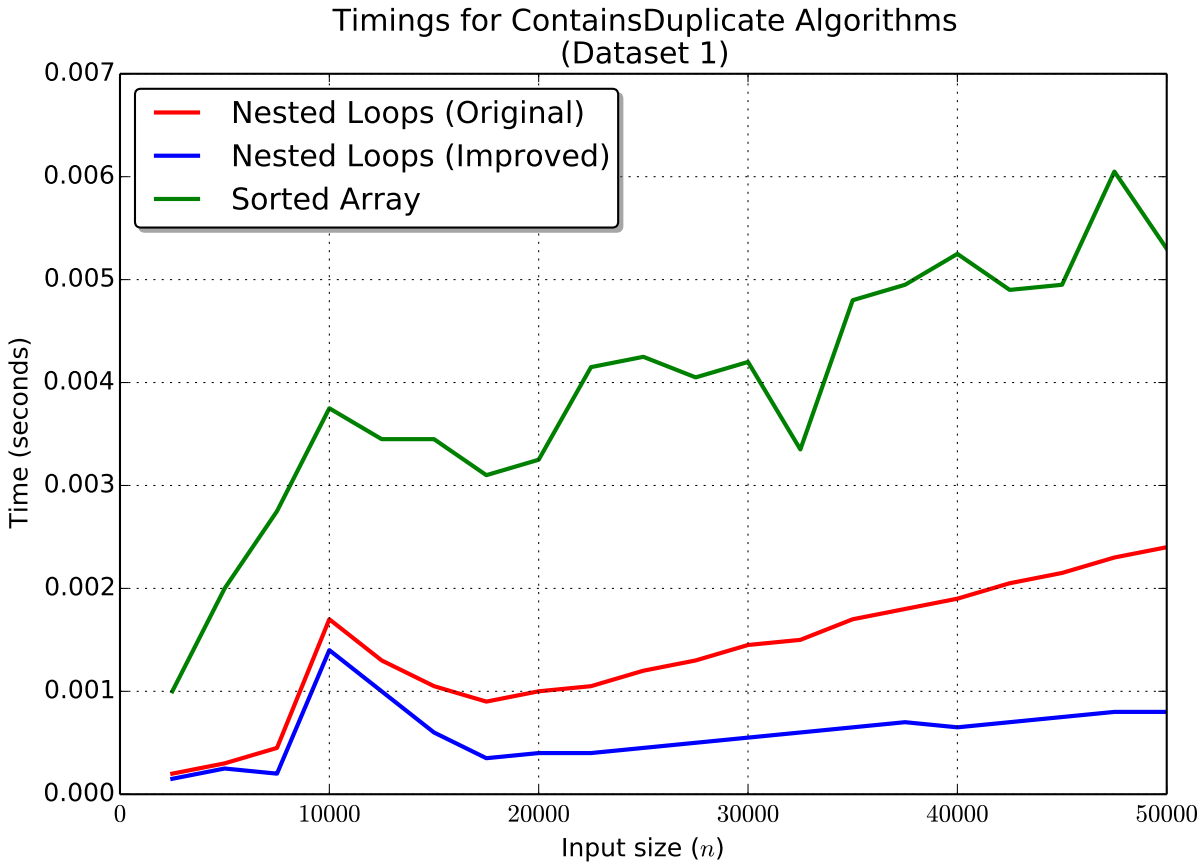
To do this, we will need to decide what 'best' means, what factors to use when comparing algorithms, and how to measure those factors.

Algorithms for ContainsDuplicate (3)



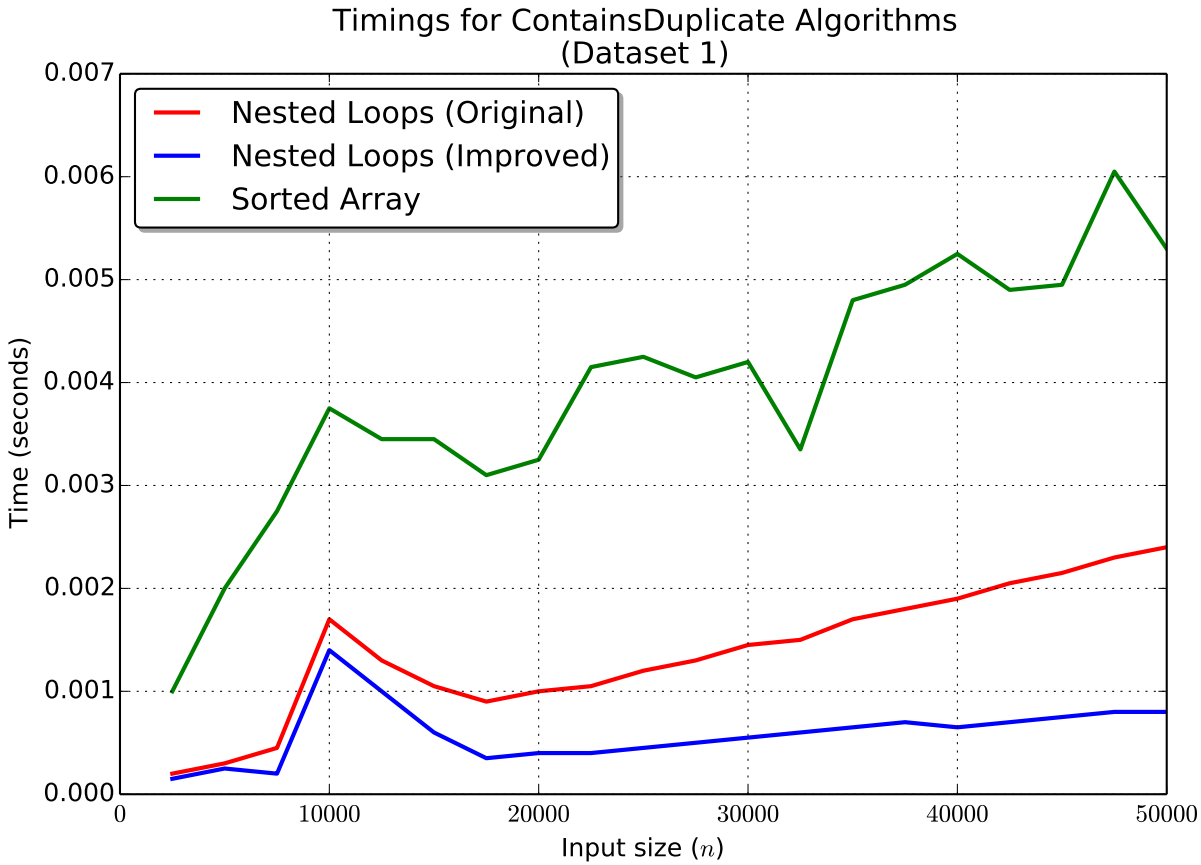
One option is to implement each algorithm, then measure the running time of each implementation on different collections of data.

Algorithms for ContainsDuplicate (4)



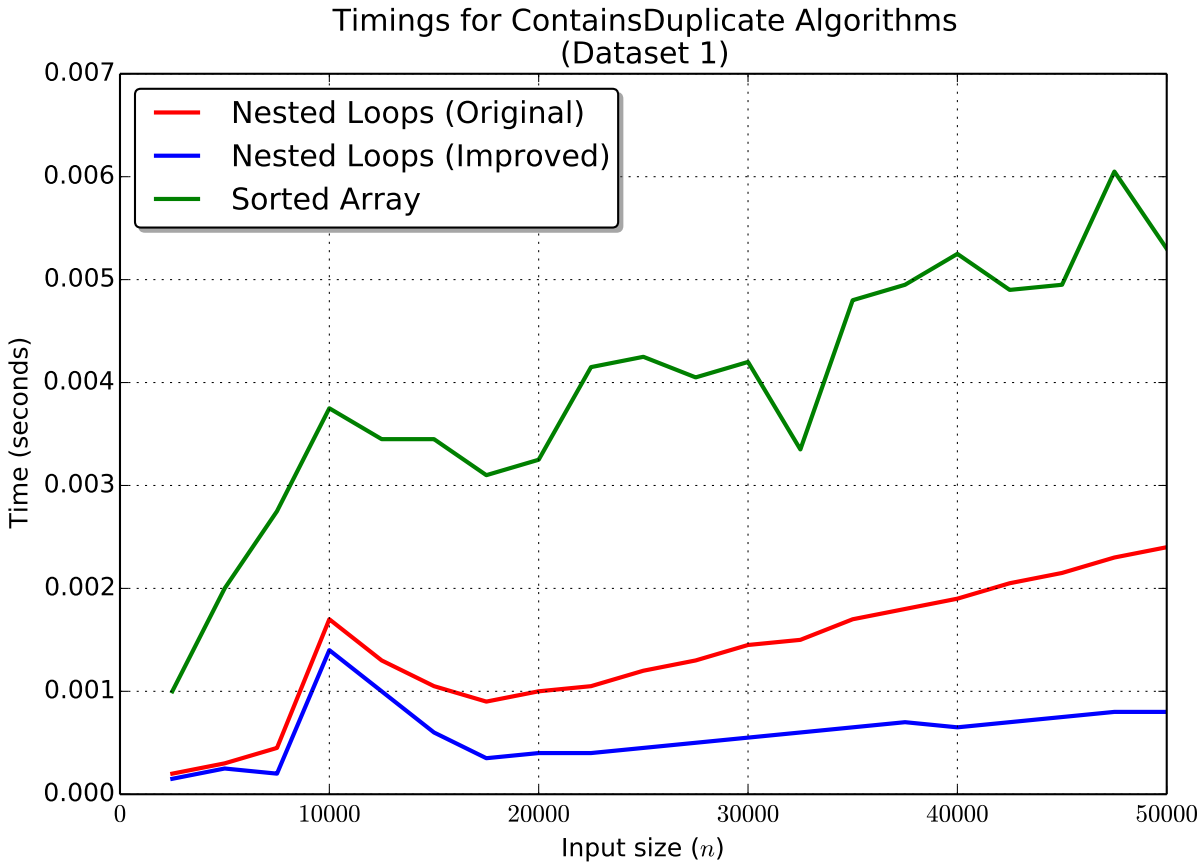
The plot above compares Java implementations of the three ContainsDuplicate algorithms on input datasets of between 2500 and 50000 elements.

Algorithms for ContainsDuplicate (5)



From the data in the plot, what can we conclude about which algorithm is the fastest?

Algorithms for ContainsDuplicate (6)



From the data in the plot, what can we conclude about which algorithm is the fastest?

NOTHING.

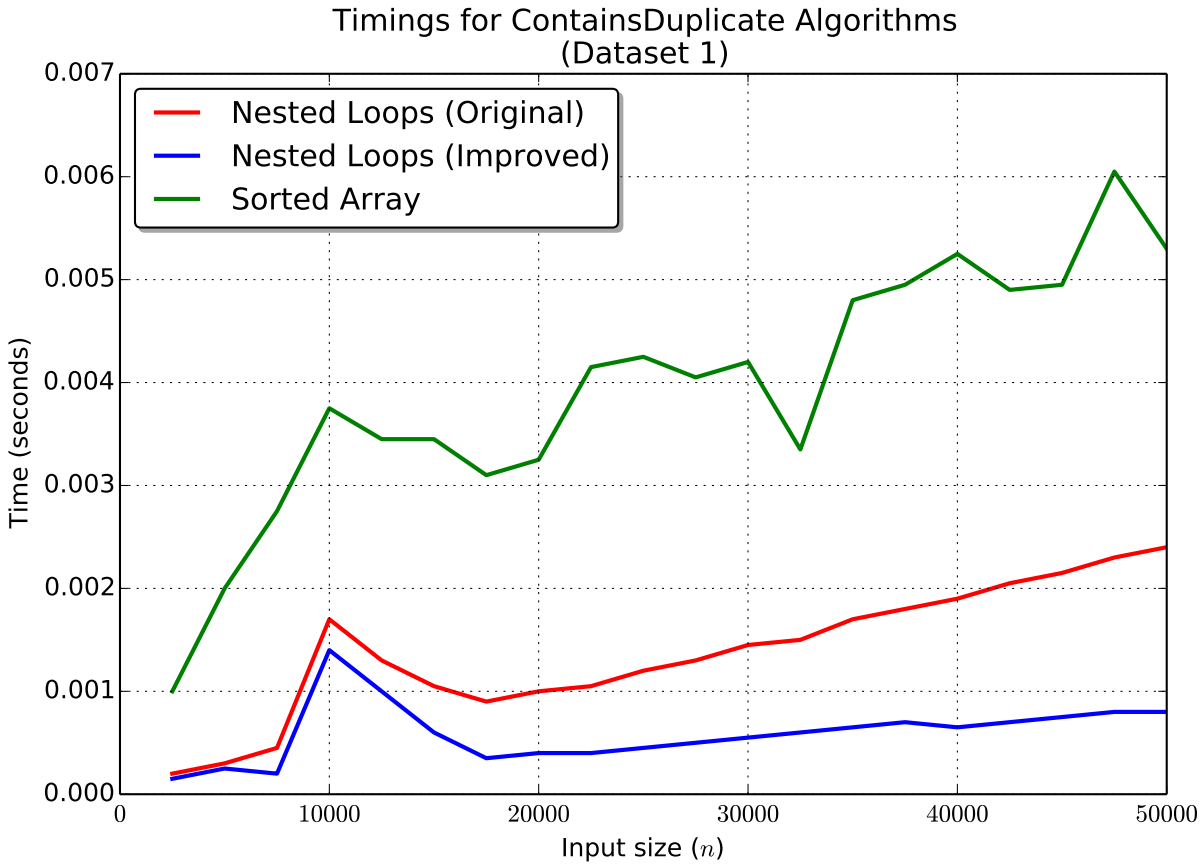
Timing Experiments (1)

- Collecting timings of different implementations on different inputs is a form of scientific experiment.
- As with any experiment, the results do not prove anything about the behavior of algorithms outside the controlled conditions in which the timings were collected.
- A timing experiment can be very useful in comparing the relative performance of different **implementations**, different **architectures** or different **input datasets** (but not all at once).
- Usually, timing experiments are only useful near the end of the development process, after algorithms have been analysed theoretically and the exact constraints of the input data are known.
- There are tools (for example, `gprof` for C code) for instrumenting programs to collect timing data.

Timing Experiments (2)

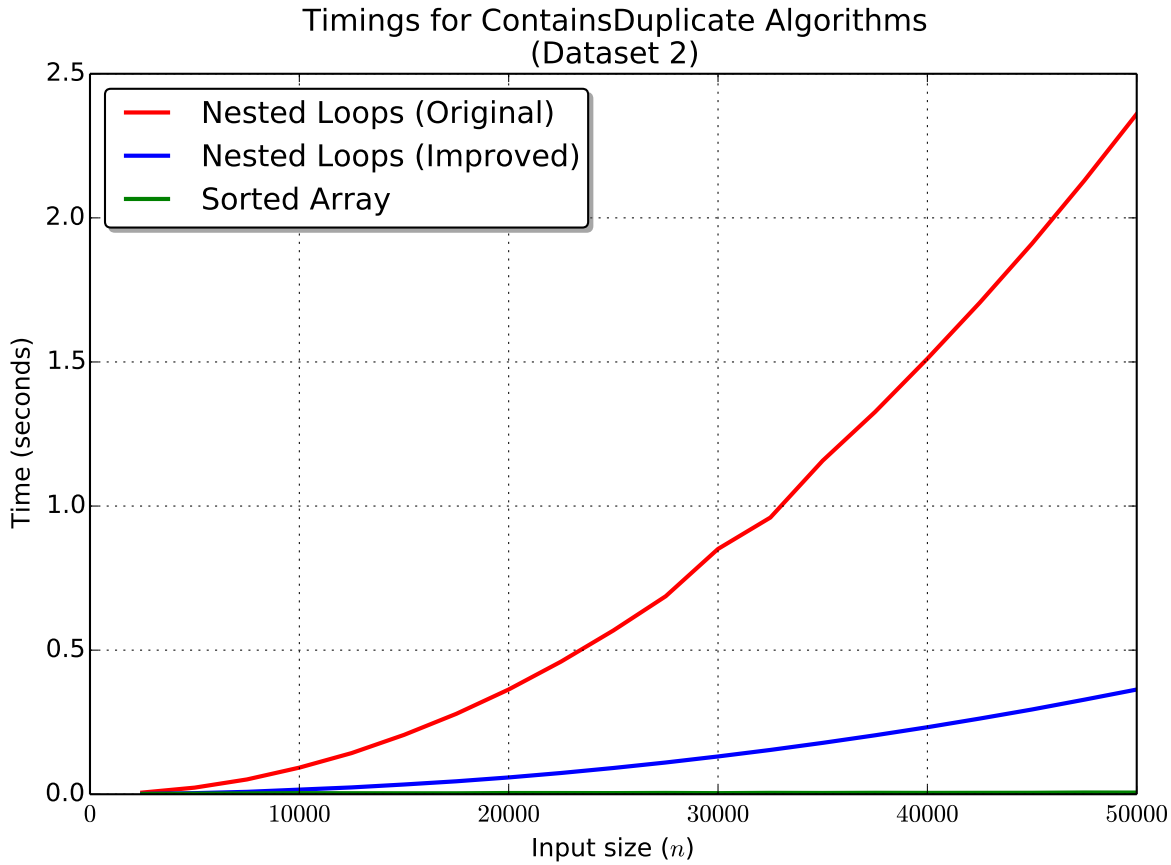
- The plot on the previous slide seems to suggest that the nested loop algorithms are faster.
- Assuming the data was gathered accurately, the results do support the hypothesis that the nested loop algorithms are faster when implemented with particular Java code and run on a particular machine with particular input data.
- However, the implementations could be badly written, the machine could use an unusual architecture, or the input data could have been cherrypicked to give the desired results.
- **Never trust timings unless you can reproduce them**. Timing data should be accompanied by detailed descriptions of the implementations (usually including code), machines and input data used for the experiment.

Timing Experiments (3)



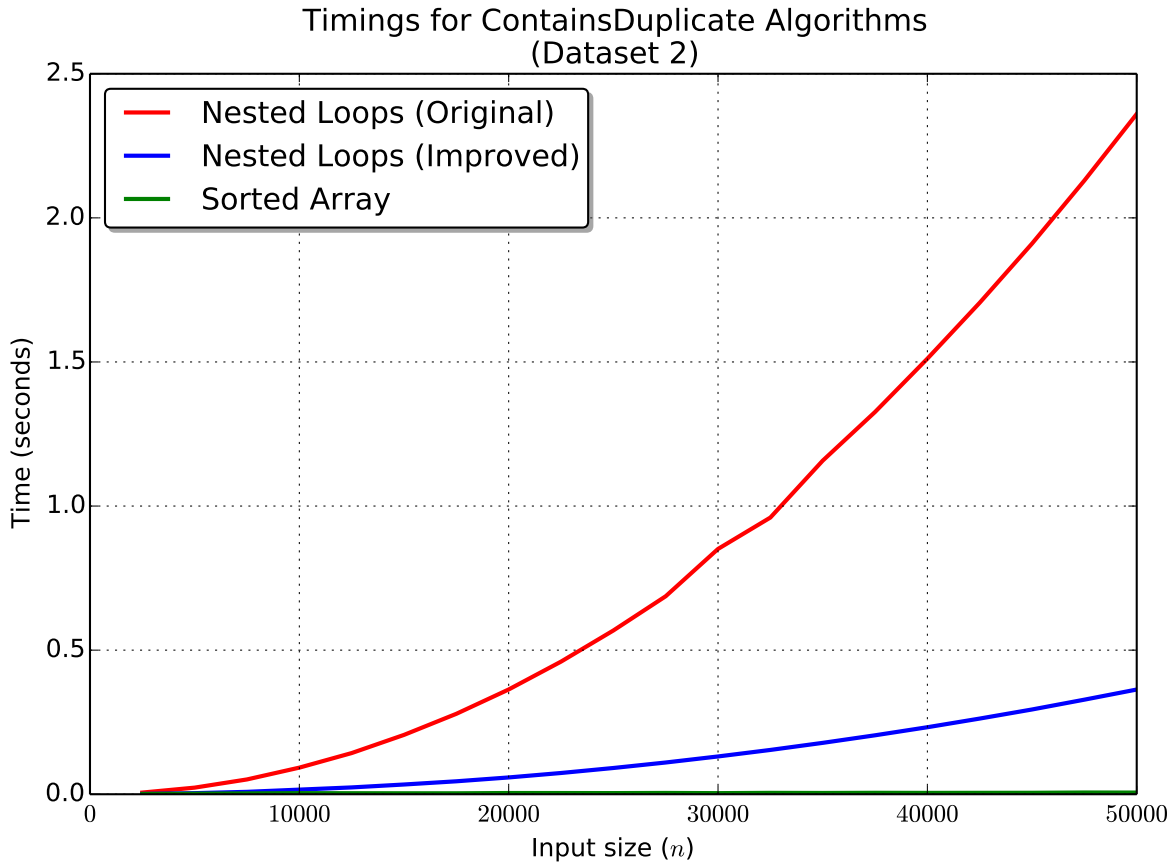
The input data *was* cherrypicked to give these results.

Timing Experiments (4)



Using the same implementations and machine, but different input data, gives completely different results.

Timing Experiments (5)



On the largest input, the first nested loop algorithm takes 2.36 seconds (1000 times longer than in the previous plot).

The sorted array algorithm takes about 0.005 seconds on the largest input.

Worst Case Running Time (1)

The **worst case running time** of an algorithm is the maximum running time of the algorithm on **any** input.

A **worst case input** for an algorithm is an input which results in the worst case running time.

Worst case running time is usually expressed in terms of input size, which is denoted by n.

Worst Case Running Time (2)

```
procedure ContainsDuplicate(A)
    n ← Length of A
    for i = 0, 1, 2, ..., n − 1 do
        for j = 0, 1, 2, ..., n − 1 do
            if i ≠ j and A[i] = A[j] then
                return True
            end if
        end for
    end for
    return False
end procedure
```

The worst case running time of the ContainsDuplicate function above occurs when the algorithm returns False, indicating that no pair is found.

Worst Case Running Time (3)

```
procedure SumArray(A)
    n ← Length of A
    sum ← 0
    for i = 0, 1, 2, ..., n − 1 do
        sum ← sum + A[i]
    end for
    return sum
end procedure
```

- The SumArray function above adds together the elements of an array.
- SumArray has the same running time for every array of a given size.
- As a result, every input is a worst case input.

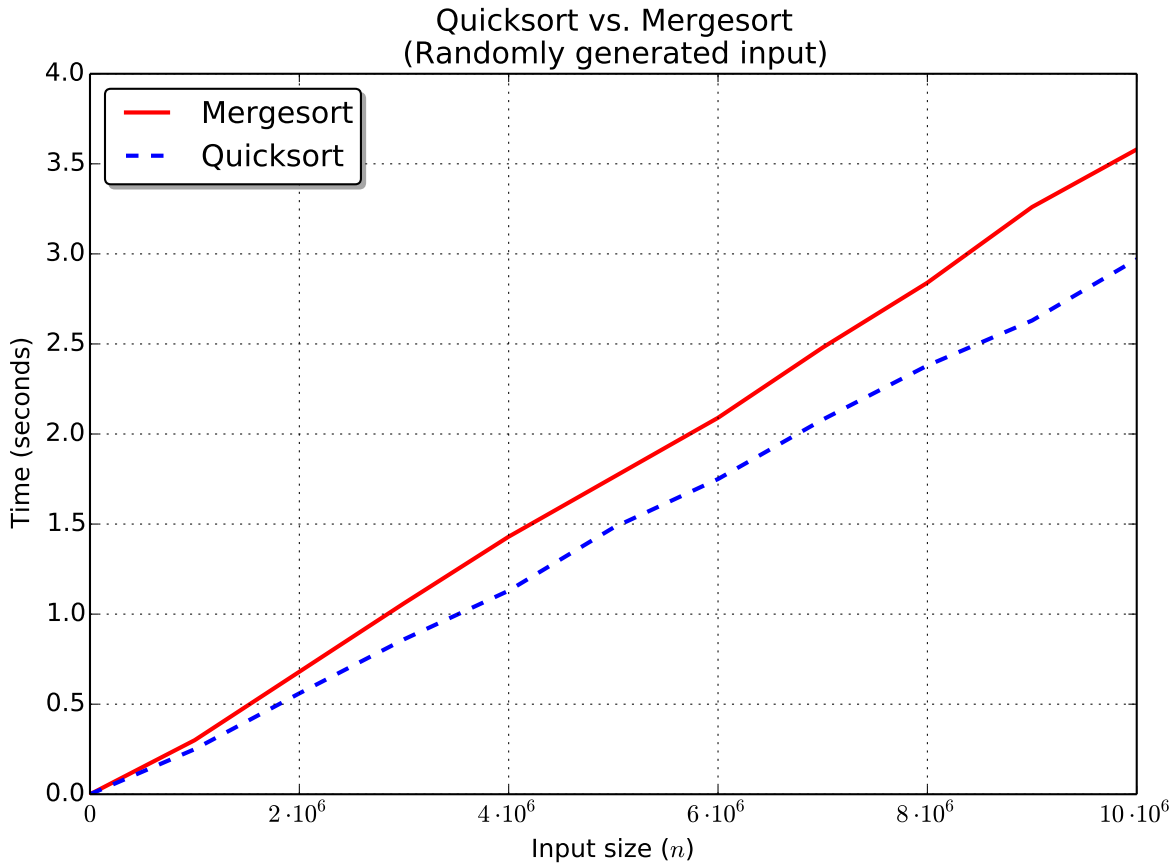
Best Case, Expected Case

The **best case running time** of an algorithm is the minimum running time of the algorithm on **any** input. Normally, the best case running time is only useful for understanding the algorithm, not as a measurement of performance.

The **expected case running time** of an algorithm is the average running time of the algorithm on **all** possible inputs.

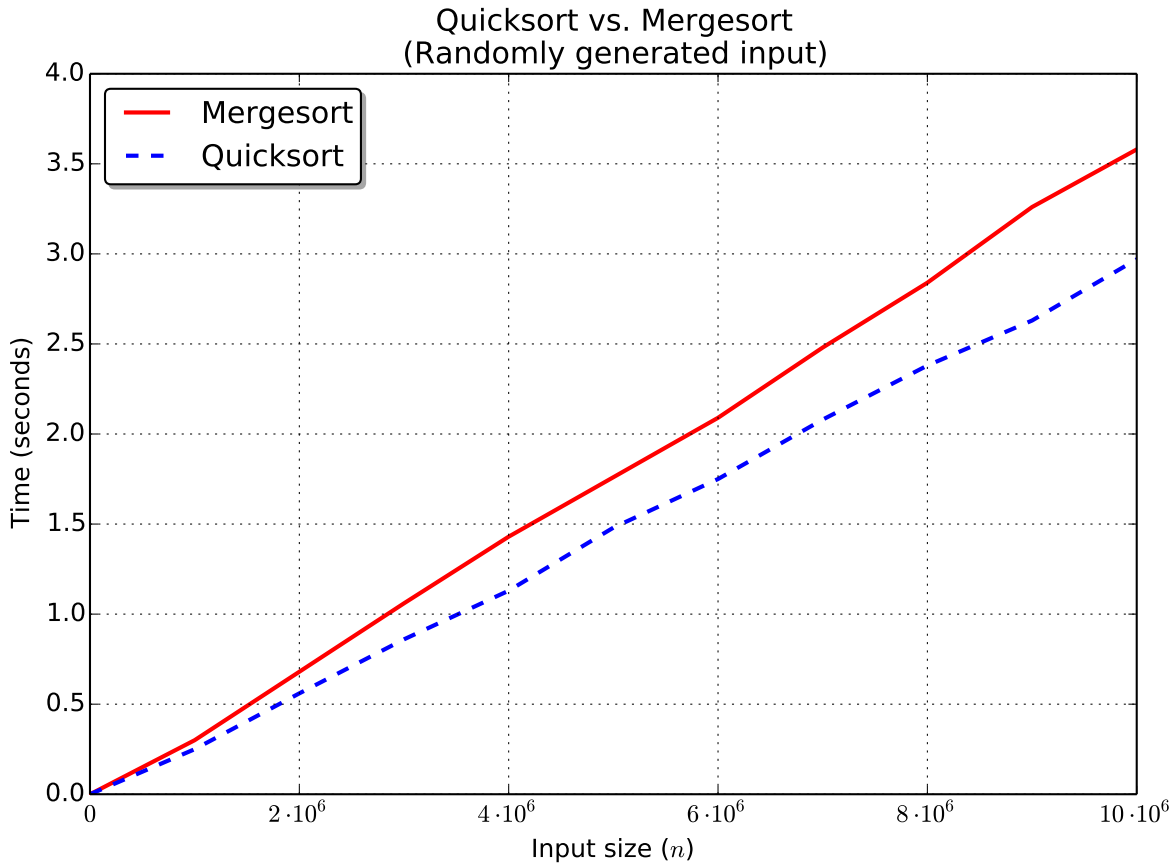
For practical purposes, algorithms are usually selected on the basis of both worst case and expected case running times.

Example: Sorting Algorithms (1)



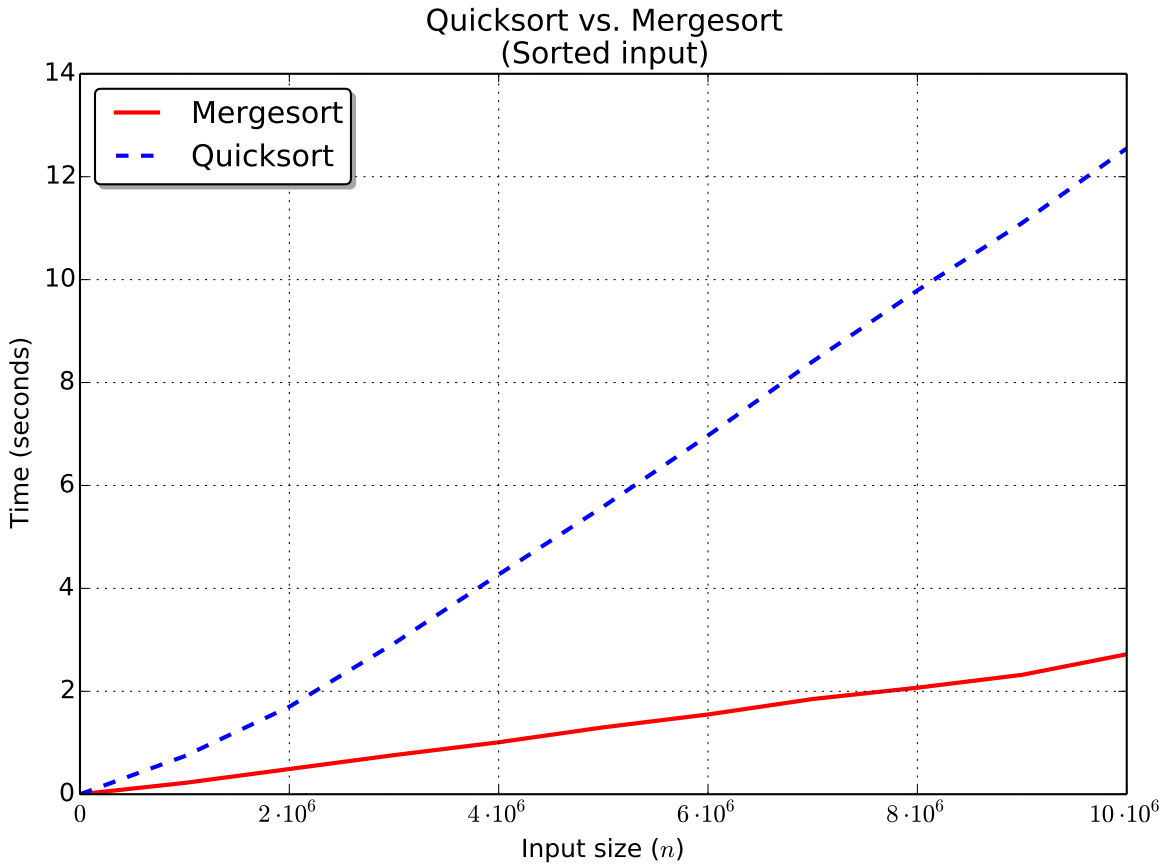
Quicksort and Mergesort are fundamental sorting algorithms. We will study both of them later in the course.

Example: Sorting Algorithms (2)



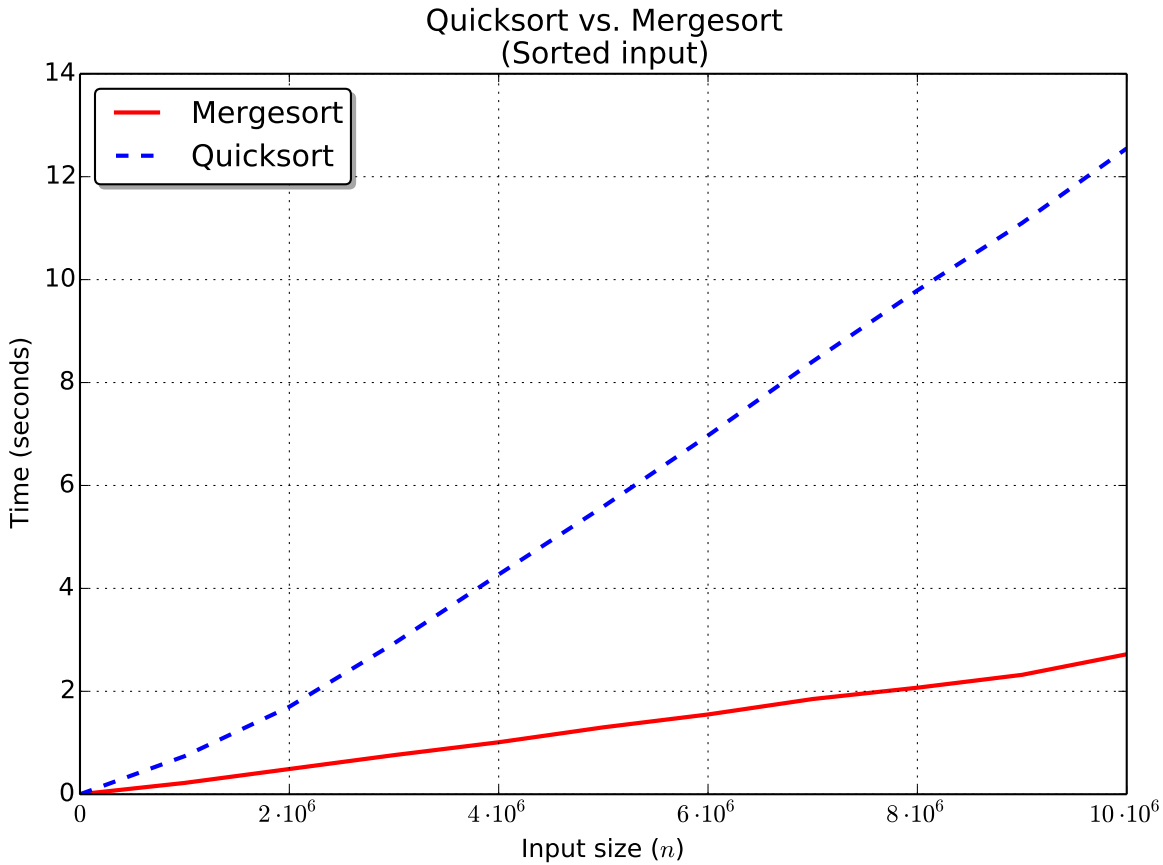
The expected case running times of Quicksort and Mergesort are similar. Quicksort tends to be slightly faster in practice.

Example: Sorting Algorithms (3)



However, in the worst case, Quicksort is significantly slower than Mergesort.

Example: Sorting Algorithms (4)



A pre-sorted array is a worst-case input for Quicksort. Mergesort's worst case running time is the same as its expected case.

Linear Search (1)

```
1: procedure LinearSearch(A, k)
2:     n ← Length of A
3:     for i = 0, 1, 2, ..., n − 1 do
4:         if A[i] = k then
5:             return i
6:         end if
7:     end for
8:     return −1
9: end procedure
```

- The LinearSearch function searches for a value k in the integer array A.
- If k is found, the function returns the first index i where $A[i] = k$.
- Otherwise, the function returns -1.

Linear Search (2)

```
1: procedure LinearSearch(A, k)
2:     n ← Length of A
3:     for i = 0, 1, 2, ..., n − 1 do
4:         if A[i] = k then
5:             return i
6:         end if
7:     end for
8:     return −1
9: end procedure
```

- In the best case, the element k is found at index 0 and the function returns after the first iteration of the loop.
- Lines 2 and 8 require the same amount of time regardless of the size of A.

Linear Search (3)

```
1: procedure LinearSearch(A, k)
2:     n ← Length of A
3:     for i = 0, 1, 2, ..., n − 1 do
4:         if A[i] = k then
5:             return i
6:         end if
7:     end for
8:     return −1
9: end procedure
```

- Therefore, LinearSearch requires a fixed amount of time in the best case, since only one element will be examined, regardless of the input size n.

Linear Search (4)

```
1: procedure LinearSearch(A, k)
2:     n ← Length of A
3:     for i = 0, 1, 2, ..., n − 1 do
4:         if A[i] = k then
5:             return i
6:         end if
7:     end for
8:     return −1
9: end procedure
```

- In the worst case, the element k is never found, and the loop on lines 3 - 7 must iterate over all n elements of the array.
- Since the number of iterations of the main loop is equal to the input size n, we say that the algorithm runs in worst case **linear time**.

Linear Search (5)

```
1: procedure LinearSearch(A, k)
2:     n ← Length of A
3:     for i = 0, 1, 2, ..., n − 1 do
4:         if A[i] = k then
5:             return i
6:         end if
7:     end for
8:     return −1
9: end procedure
```

Alternatively, we can say that the algorithm is $O(n)$ (pronounced "big-O of n").

We will formally define our analysis method and big-O notation in the next lecture.