Sorting Correctly and Efficiently

Week 7

COMP 1201 (Algorithmics)

Dr Andrew Sogokon `a.sogokon@soton.ac.uk`
ECS, University of Southampton

11 March 2020

Previously...

- Pseudocode
- Basics of sorting algorithms (sorts)
- **Stable** vs **unstable** sorts
- **In-place** sorts
- Simple sorts: **Insertion Sort**, **Selection Sort**
- Examples of their operation and pseudocode.

What we care about in algorithms

Algorithms need to be *correct*, *efficient* and *easy to implement*.

In that order.

Showing correctness requires mathematical proof

(A) Mathematics is common sense,

(B) Do not ask whether a statement is true until you know what it means,

(C) A proof is any completely convincing argument.

— *Errett Bishop, 1973*.

A simple program

Algorithm 1 A simple program

```
procedure Foo(n)                          ▷ Input is a positive integer n
    while n ≠ 1 do
        if n = 0 mod 2 then               ▷ If n is even, halve it
            n ← n/2
        else                              ▷ If n is odd, triple it and add 1
            n ← 3n + 1
    return true                           ▷ Stop when n = 1
```

A simple program

Algorithm 1 A simple program

procedure FOO(n) ▷ Input is a positive integer n
  while $n \neq 1$ **do**
    if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
      $n \leftarrow n/2$
    else ▷ If n is odd, triple it and add 1
      $n \leftarrow 3n + 1$
  return true ▷ Stop when $n = 1$

<u>Claim</u>: the above program terminates on any valid input.

A simple program

Algorithm 1 A simple program

procedure $\textsc{Foo}(n)$ ▷ Input is a positive integer n
  while $n \neq 1$ **do**
    if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
      $n \leftarrow n/2$
    else ▷ If n is odd, triple it and add 1
      $n \leftarrow 3n + 1$
  return true ▷ Stop when $n = 1$

<u>Claim</u>: the above program terminates on any valid input.

$$\textsc{Foo}(10) = \textbf{true}$$

A simple program

Algorithm 1 A simple program

procedure $\textsc{Foo}(n)$ ▷ Input is a positive integer n
  while $n \neq 1$ **do**
    if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
      $n \leftarrow n/2$
    else ▷ If n is odd, triple it and add 1
      $n \leftarrow 3n + 1$
  return true ▷ Stop when $n = 1$

Claim: the above program terminates on any valid input.

$$\textsc{Foo}(1000) = \textbf{true}$$

A simple program

Algorithm 1 A simple program

```
procedure Foo(n)                          ▷ Input is a positive integer n
    while n ≠ 1 do
        if n = 0 mod 2 then               ▷ If n is even, halve it
            n ← n/2
        else                              ▷ If n is odd, triple it and add 1
            n ← 3n + 1
    return true                           ▷ Stop when n = 1
```

<u>Claim</u>: the above program terminates on any valid input.

$$\textsc{Foo}(18061815) = \textbf{true}$$

A simple program

Algorithm 1 A simple program

```
procedure Foo(n)                              ▷ Input is a positive integer n
    while n ≠ 1 do
        if n = 0 mod 2 then                   ▷ If n is even, halve it
            n ← n/2
        else                                  ▷ If n is odd, triple it and add 1
            n ← 3n + 1
    return true                               ▷ Stop when n = 1
```

Claim: the above program terminates on any valid input.

$$\textsc{Foo}(21101805) = \textbf{true}$$

A simple program

Algorithm 1 A simple program

procedure $\textsc{Foo}(n)$ ▷ Input is a positive integer n
  while $n \neq 1$ **do**
    if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
      $n \leftarrow n/2$
    else ▷ If n is odd, triple it and add 1
      $n \leftarrow 3n + 1$
  return true ▷ Stop when $n = 1$

Claim: the above program terminates on any valid input.

$$\textsc{Foo}(25101415) = \textbf{true}$$

A simple program

Algorithm 1 A simple program

```
procedure Foo(n)                         ▷ Input is a positive integer n
    while n ≠ 1 do
        if n = 0 mod 2 then              ▷ If n is even, halve it
            n ← n/2
        else                             ▷ If n is odd, triple it and add 1
            n ← 3n + 1
    return true                          ▷ Stop when n = 1
```

Collatz conjecture (1937): the above program terminates on any valid input. (No proof. Major unsolved problem in mathematics.)

A simple program

Algorithm 1 A simple program

procedure $\textsc{Foo}(n)$ ▷ Input is a positive integer n
    while $n \neq 1$ **do**
        if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
            $n \leftarrow n/2$
        else ▷ If n is odd, triple it and add 1
            $n \leftarrow 3n + 1$
    return true ▷ Stop when $n = 1$

Collatz conjecture (1937): the above program terminates on any valid input. (No proof. Major unsolved problem in mathematics.)

$$\textsc{Foo}(n) = \textbf{true} \quad \text{checked for all } n \in [1, 87 \times 2^{60}]$$

A simple program

Algorithm 1 A simple program

procedure $\textsc{Foo}(n)$ ▷ Input is a positive integer n
    while $n \neq 1$ **do**
        if $n = 0 \mod 2$ **then** ▷ If n is even, halve it
            $n \leftarrow n/2$
        else ▷ If n is odd, triple it and add 1
            $n \leftarrow 3n + 1$
    return true ▷ Stop when $n = 1$

A more modest property we *can* prove about the above program: value of n is always above zero.

A simple program

Algorithm 1 A simple program

```
procedure Foo(n)                          ▷ Input is a positive integer n
    while n ≠ 1 do
        if n = 0 mod 2 then               ▷ If n is even, halve it
            n ← n/2
        else                              ▷ If n is odd, triple it and add 1
            n ← 3n + 1
    return true                           ▷ Stop when n = 1
```

A more modest property we *can* prove about the above program: value of n is always above zero.

This property is *true initially*, and is *maintained* by all the operations within the loop.

Loop invariants

A **loop invariant** is a *property of a loop* that helps us understand why an algorithm is correct.

A property is a loop invariant if we can show the following:

- i **Initialisation**: it holds true prior to the first iteration of the loop.
- ii **Maintenance**: If it is true before an iteration of the loop, it remains true before the *next* iteration.
- iii **Termination**: When the loop terminates, the invariant gives us a useful property that helps us show that the algorithm is correct.

Correctness of Sorting Algorithms

Fortunately, showing that sorting algorithms terminate is usually straightforward.

When is a sorting algorithm *correct*?

1. When its output is in non-decreasing order (i.e. the output is *sorted* according to some *total order*), **and**
2. the items in the output are a *permutation* of the items in the input.

We often prove correctness of sorting algorithms using loop invariants (though it can take a surprising amount of work to do this rigorously!) [**CLRS**, Ch. 2]

Correctness of Sorting Algorithms

Fortunately, showing that sorting algorithms terminate is usually straightforward.

When is a sorting algorithm *correct*?

1. When its output is in non-decreasing order (i.e. the output is *sorted* according to some *total order*), **and**
2. the items in the output are a *permutation* of the items in the input.

We often prove correctness of sorting algorithms using loop invariants (though it can take a surprising amount of work to do this rigorously!) [**CLRS**, Ch. 2]

(Proofs of correctness using loop invariants will not be examinable.)

Insertion Sort

Recall: Insertion Sort keeps a *sub-array of items on the left in (correctly) sorted order*.

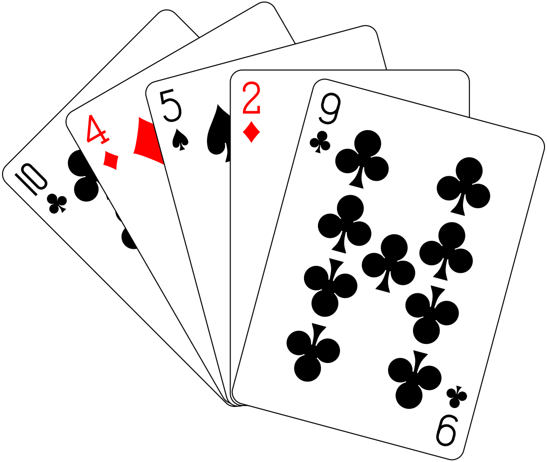
- This sub-array is increased by **inserting** the next item into its (relatively) correct position in the sorted sub-array.
- With each iteration we move the current item one to the right.

Insertion Sort

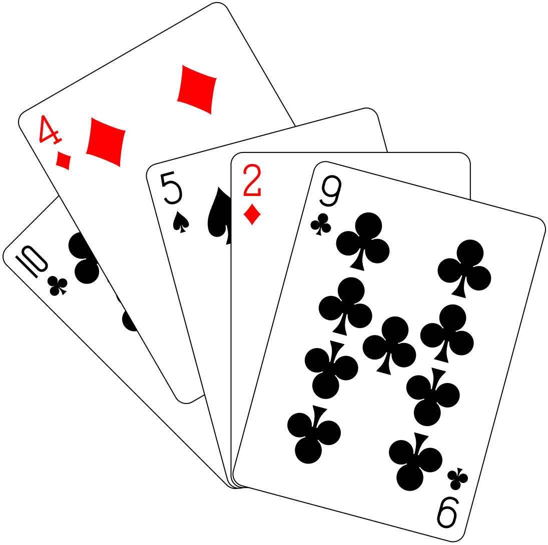
Algorithm 2 Insertion Sort

procedure InsertionSort($\mathbf{a}$)
    for $j \leftarrow 2$ to $\mathbf{a}$.length **do**
        $key \leftarrow a_j$
        $i \leftarrow j - 1$
        while $i > 0$ and $a_i > key$ **do**
            $a_{i+1} \leftarrow a_i$
            $i \leftarrow i - 1$
        $a_{i+1} \leftarrow key$
    return $\mathbf{a}$ ▷ Sorted sequence

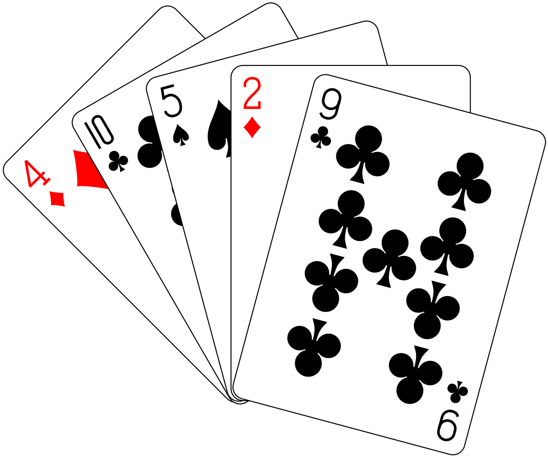
Insertion Sort



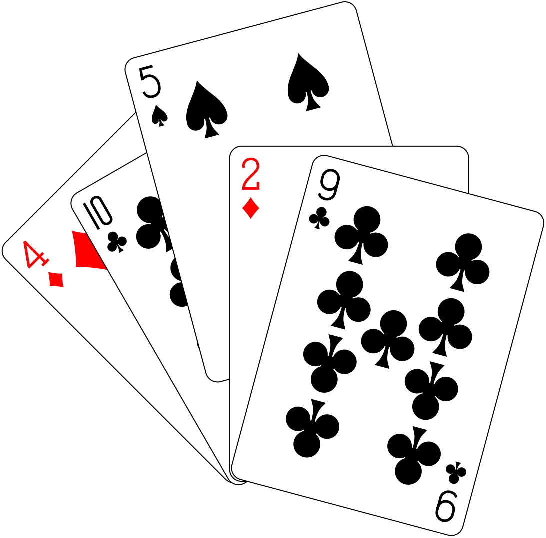
Insertion Sort



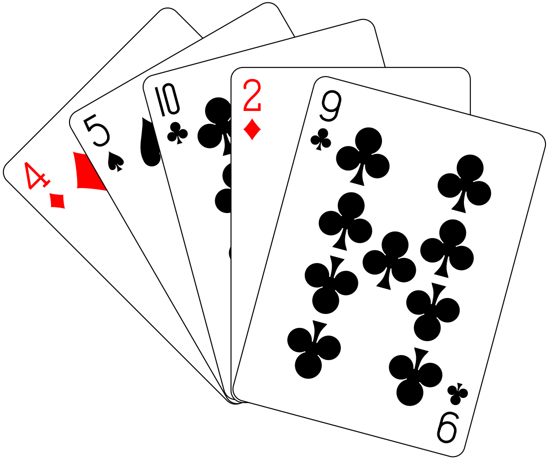
Insertion Sort



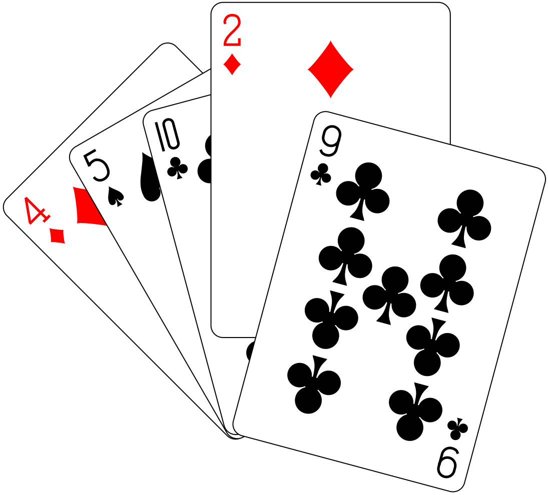
Insertion Sort



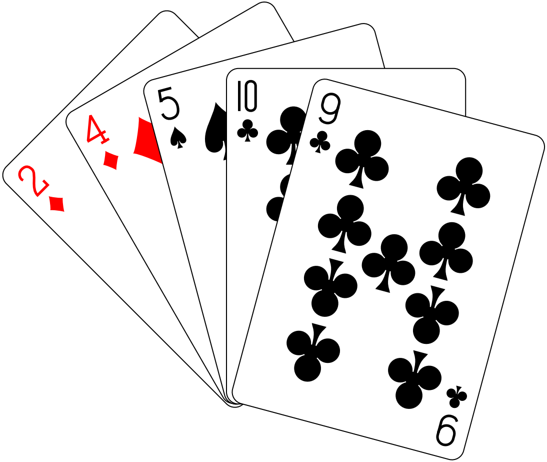
Insertion Sort



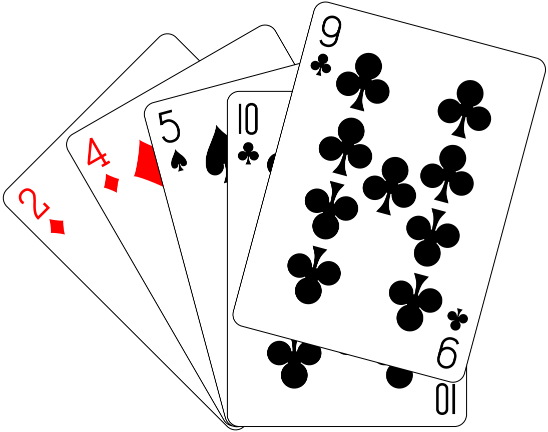
Insertion Sort



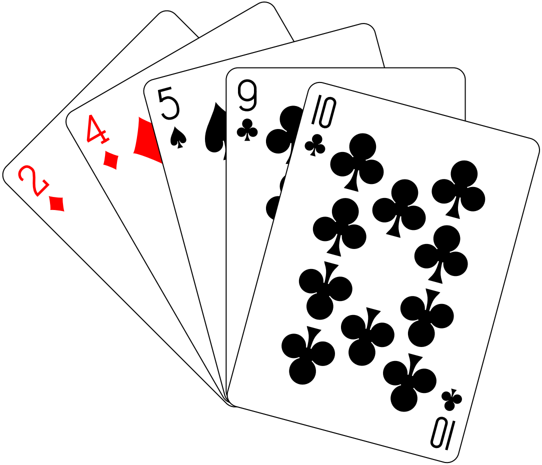
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

In order to prove correctness of Insertion Sort, we can use the following loop invariant:

"At the start of each iteration of the **for** loop, the sub-array $a_1 a_2 \ldots a_{j-1}$ consists of all items originally in $a_1 a_2 \ldots a_{j-1}$, but in sorted order."

Bubble Sort

Bubble sort is another example of *simple sorting algorithm*.

Main idea: keep swapping neighbouring items until the array is sorted.

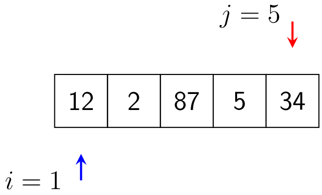
- Intuition: items "bubble up" through the array into their correct position.
- Bubble Sort is **stable** and **in-place** ($O(1)$ space complexity).
- Time complexity is $O(n^2)$.
- Not a bad simple sort, but does more work than insertion sort and selection sort.

Bubble Sort

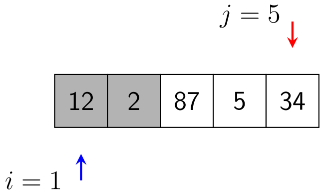
Algorithm 3 Bubble Sort

procedure BUBBLESORT($\mathbf{a}$)
    for $j \leftarrow \mathbf{a}$.length to 2 **do**
        for $i \leftarrow 1$ to $j - 1$ **do**
            if $a_i > a_{i+1}$ **then**
                swap(a_i, a_{i+1}) ▷ Exchange adjacent elements
    return $\mathbf{a}$ ▷ Sorted sequence

Bubble Sort



Bubble Sort



Bubble Sort

$j = 5$

2	12	87	5	34

$i = 1$

Bubble Sort

$j = 5$

2	12	87	5	34

$i = 2$

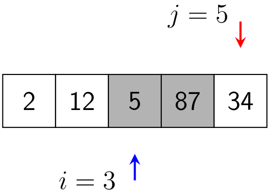
Bubble Sort

$j = 5$ ↓

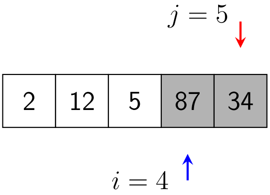
2	12	87	5	34

$i = 3$ ↑

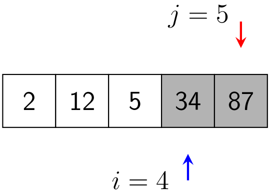
Bubble Sort



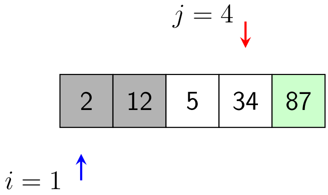
Bubble Sort



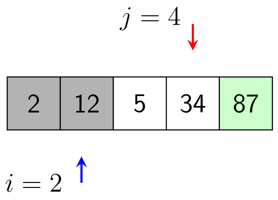
Bubble Sort



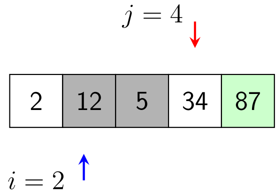
Bubble Sort



Bubble Sort



Bubble Sort



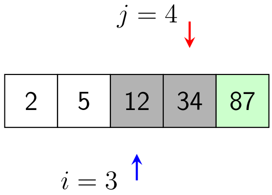
Bubble Sort

$j = 4$

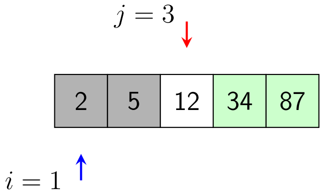
2	5	12	34	87

$i = 2$

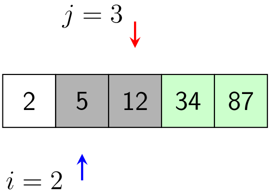
Bubble Sort



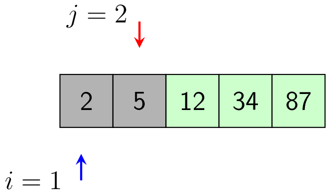
Bubble Sort



Bubble Sort



Bubble Sort



Bubble Sort

$j = 2$

2	5	12	34	87

$i = 1$

Bubble Sort

Algorithm 4 Bubble Sort

```
procedure BubbleSort(a)
    for j ← a.length to 2 do
        for i ← 1 to j − 1 do
            if a_i > a_{i+1} then
                swap(a_i, a_{i+1})        ▷ Exchange adjacent elements
    return a                              ▷ Sorted sequence
```

Question: can you think of a loop invariant for Bubble Sort?

More on correctness and loop invariants:

1. Chapter 2 in Cormen (CLRS) **Introduction to Algorithms**.

Optional material for those interested in correctness:

1. **Edsger Dijkstra**'s 1990 lecture "*Reasoning about programs*" https://www.youtube.com/watch?v=GX3URhx6i2E
2. Paper that exposed a bug in OpenJDK's implementation of TimSort is by **Stijn de Gouw et al.** "*OpenJDK's java.utils.Collection.sort() is broken:The good, the bad and the worst case*". Computer Aided Verification (CAV) 2015.

Back to efficiency

- Lower bound on the complexity of *comparison-based* sorts
- Efficient sorts (comparison-based):
  - **Merge Sort**
  - **Quicksort**

Comparison-based sorting algorithms

A **comparison-based sorting algorithm** (comparison-based sort):

- a sorting algorithm
- can *only* gain information about the items in the input sequence $a_1, a_2, \ldots, a_n$ by performing *pairwise-comparisons*.

A pairwise-comparison is a query such as "is $a_i < a_j$?"

Most general-purpose sorting algorithms are comparison-based.

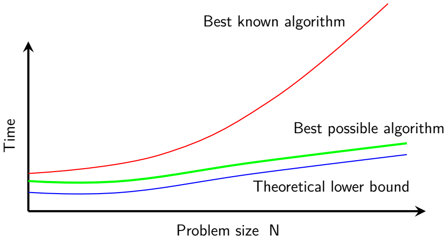
Examples:

- Insertion Sort, Selection Sort, Bubble Sort,
- Merge Sort, Quicksort.

Lower bounds on time complexity

Given a problem we would like to know what is the time complexity of the **best possible algorithmic solution**.

- A lower bound of $f(n)$ is a guarantee that no one can use fewer than $f(n)$ operations.
- Solving the general problem **requires** at least $f(n)$ operations.
- Lower bounds give the difficulty of the problem.

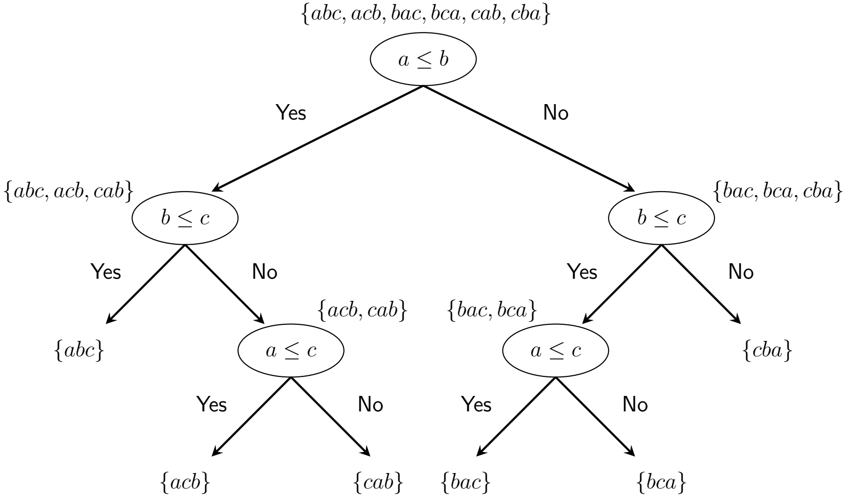


Decision Trees

We are interested in establishing a lower bound on the number of *comparisons* needed for sorting.

- **Decision trees** are a way to visualise many algorithms (at least in principle).
- A decision tree shows a series of *decisions* made during an algorithm.
- In the case of sorting algorithms, a decision tree will show what the algorithm does at every comparison.

Decision Tree for Insertion Sort



Decision Trees and Time Complexity

The time taken to complete the task is the *depth of the tree* at which we finish (i.e. the **leaf nodes**).

We can use decision trees to read off the time complexity:

- **Worst case**: depth of the deepest leaf.
- **Best case**: depth of the shallowest leaf.
- **Average case**: average depth of leaves.

Different sorting strategies will have different decision trees.

Decision trees are usually far too large to write down in practice.

Correctness Requirements for Sorting

Any sorting algorithm based on pairwise-comparisons must have a leaf in its decision tree *for every possible way of sorting the list*.

For an input abc we must consider all possible permutations:

$$\{abc,\ acb,\ bac,\ bca,\ cab,\ cba\}$$

These correspond to different paths in the decision tree.

- Each leaf of the decision tree gives one possible ordering of elements.
- $n!$ possible permutations (number of leaves).

$$\text{Height of the decision tree } \geq \log_2(n!)\,.$$

Lower Bound on Comparison-based Sorting

$$\begin{aligned}
\log_2(n!) &= \log_2(1) + \log_2(2) + \cdots + \log_2(n)\\
&\le \log_2(n) + \log_2(n) + \cdots + \log_2(n)\\
&= n\log_2(n).
\end{aligned}$$

So $\log_2(n!) = O(n\log_2(n))$. [Nice, but not what we need.]

$$\begin{aligned}
\log_2(n!) &= \log_2(1) + \log_2(2) + \cdots + \log_2(n)\\
&\ge \log_2\left(\frac{n}{2}\right) + \cdots + \log_2(n)\\
&\ge \frac{n}{2}\log_2\left(\frac{n}{2}\right) = \frac{n}{2}\log_2(n) - \frac{n}{2}.
\end{aligned}$$

So $\log_2(n!) = \Omega(n\log_2(n))$. [We have a lower bound!]

Merge Sort

- Invented by **John von Neumann** in 1945.
- Employs a **divide-and-conquer** strategy.
- The problem is divided into a number of parts *recursively*.
- The solution is obtained by recombining the parts.

John von Neumann

Basic idea: divide the array into two halves and recursively sort each half; then **merge** the two sorted halves to obtain the solution.

Merge Sort

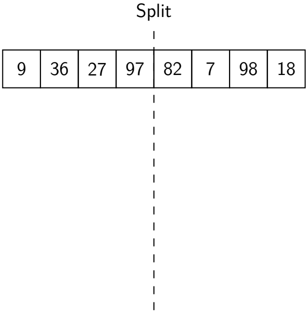
Algorithm 5 Merge Sort

procedure $\textsc{MergeSort}(\mathbf{a}, start, end)$
    if $start < end$ **then**
        $mid \leftarrow \lfloor (start + end)/2 \rfloor$ ▷ Divide problem
        $\textsc{MergeSort}(\mathbf{a}, start, mid)$ ▷ Conquer part 1
        $\textsc{MergeSort}(\mathbf{a}, mid + 1, end)$ ▷ Conquer part 2
        $\textsc{Merge}(\mathbf{a}, start, mid, end)$ ▷ Combine
    else
        return ▷ Base case

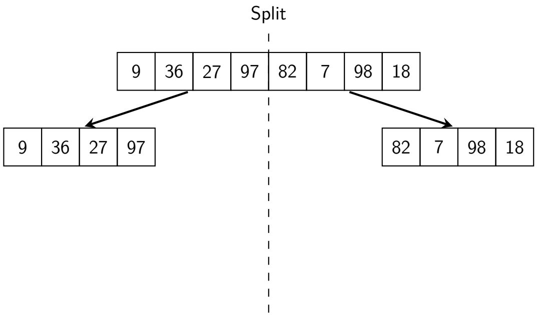
Merge Sort

9	36	27	97	82	7	98	18

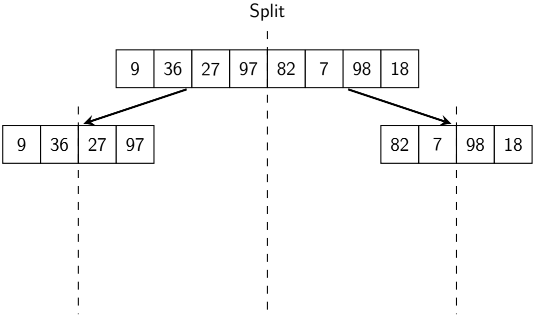
Merge Sort



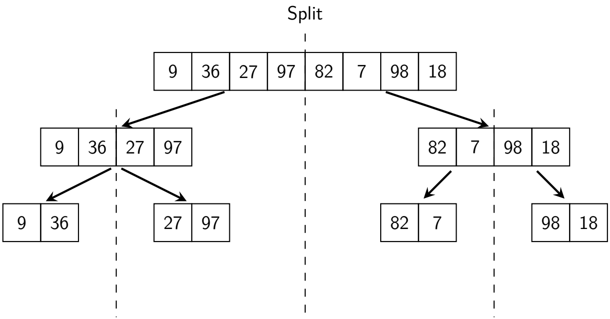
Merge Sort



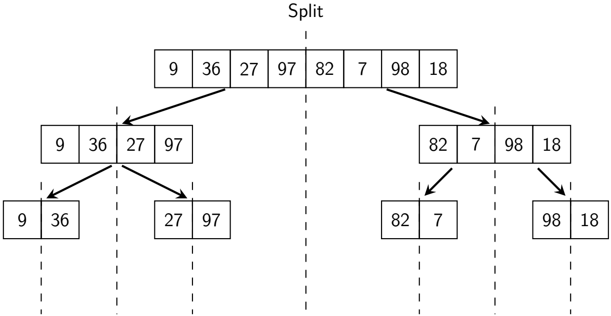
Merge Sort



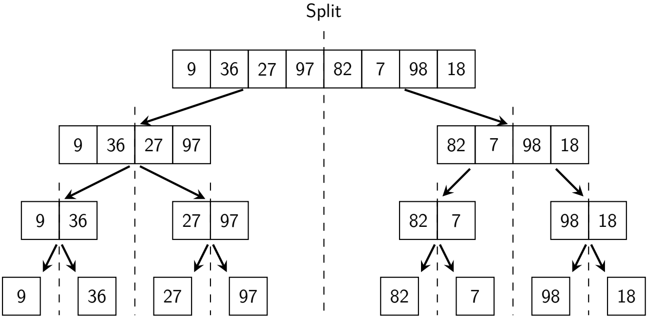
Merge Sort



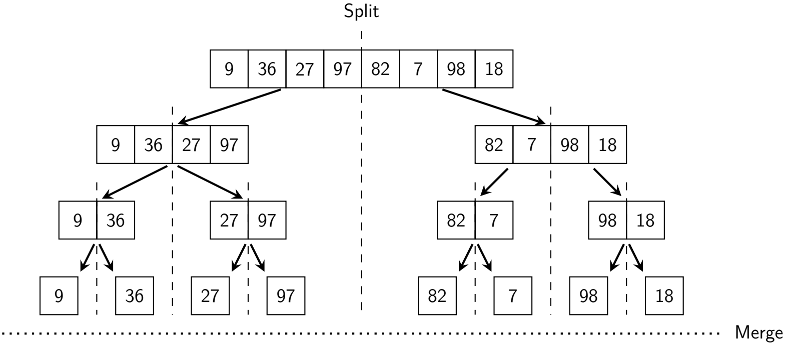
Merge Sort



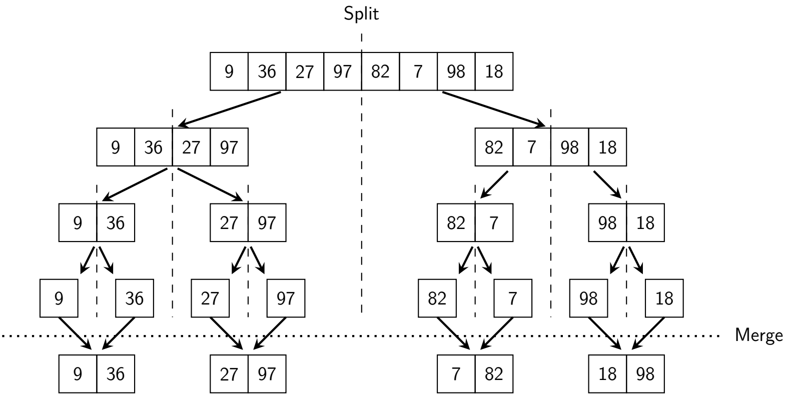
Merge Sort


Split
9
36
27
97
82
7
98
18
9
36
27
97
82
7
98
18
9
36
27
97
82
7
98
18
9
36
27
97
82
7
98
18

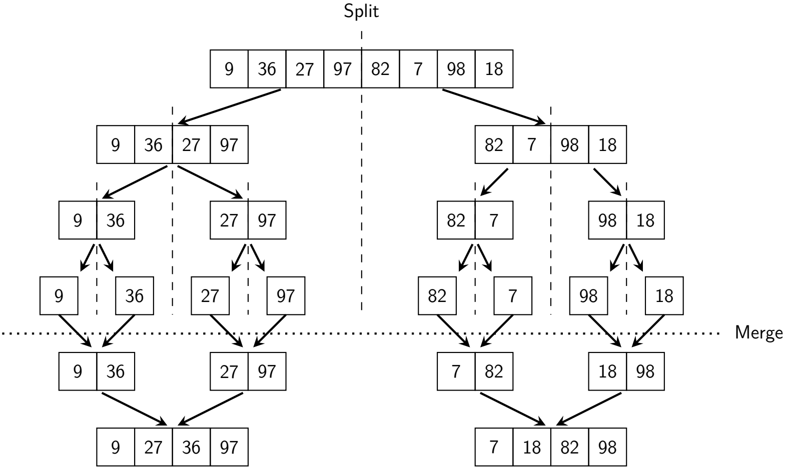
Merge Sort


Split
9 36 27 97 82 7 98 18
9 36 27 97
82 7 98 18
9 36
27 97
82 7
98 18
9
36
27
97
82
7
98
18
Merge

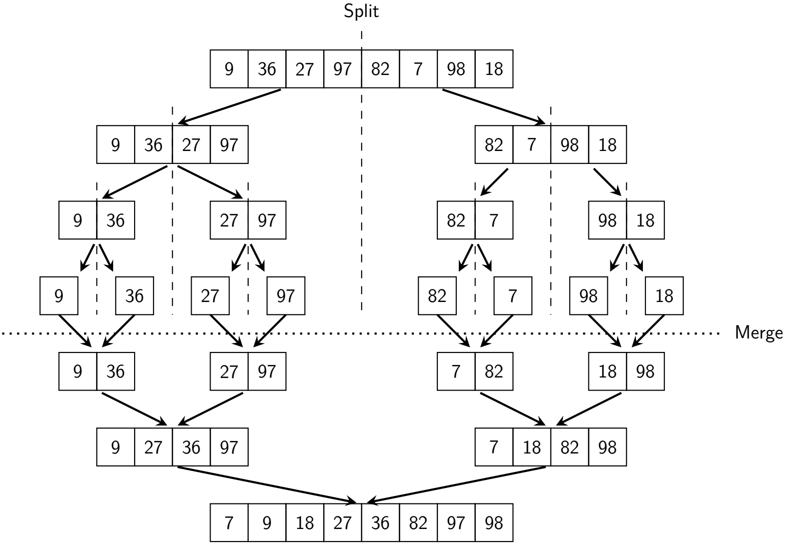
Merge Sort


Split
9 36 27 97 82 7 98 18
9 36 27 97
82 7 98 18
9 36
27 97
82 7
98 18
9
36
27
97
82
7
98
18
Merge
9 36
27 97
7 82
18 98

Merge Sort


Split
9 36 27 97 82 7 98 18
9 36 27 97
82 7 98 18
9 36
27 97
82 7
98 18
9
36
27
97
82
7
98
18
Merge
9 36
27 97
7 82
18 98
9 27 36 97
7 18 82 98

Merge Sort

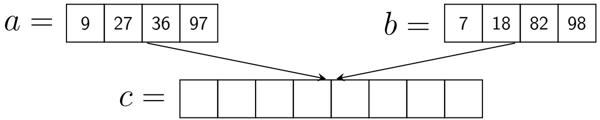

Split
9 36 27 97 82 7 98 18
9 36 27 97
82 7 98 18
9 36
27 97
82 7
98 18
9
36
27
97
82
7
98
18
Merge
9 36
27 97
7 82
18 98
9 27 36 97
7 18 82 98
7 9 18 27 36 82 97 98

Merge operation

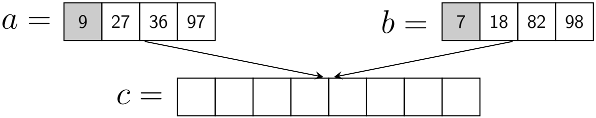
In order to merge two sorted arrays $\mathbf{a}$, $\mathbf{b}$ into one sorted array $\mathbf{c}$ we follow a simple procedure:

- Compare current elements of $\mathbf{a}$ and $\mathbf{b}$,
- Choose the smaller one and store it in $\mathbf{c}$,
- Move to the next element in the array of the chosen element.

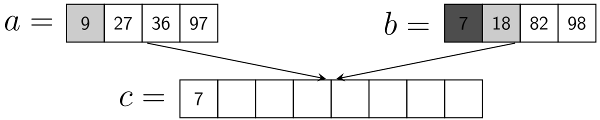
Merge operation



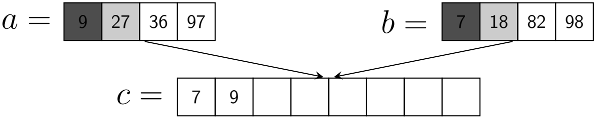
Merge operation



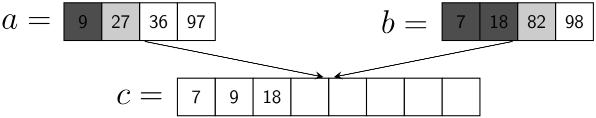
Merge operation



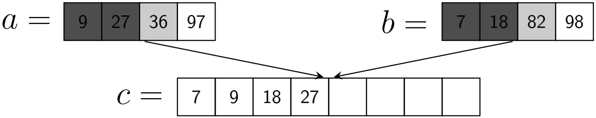
Merge operation



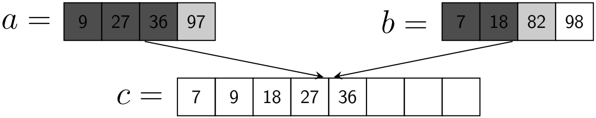
Merge operation



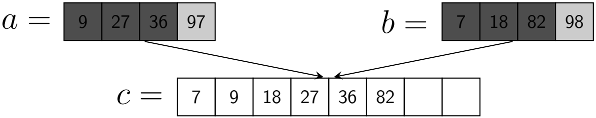
Merge operation



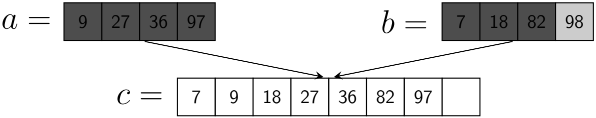
Merge operation



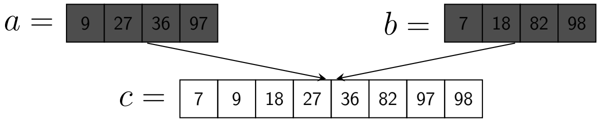
Merge operation



Merge operation



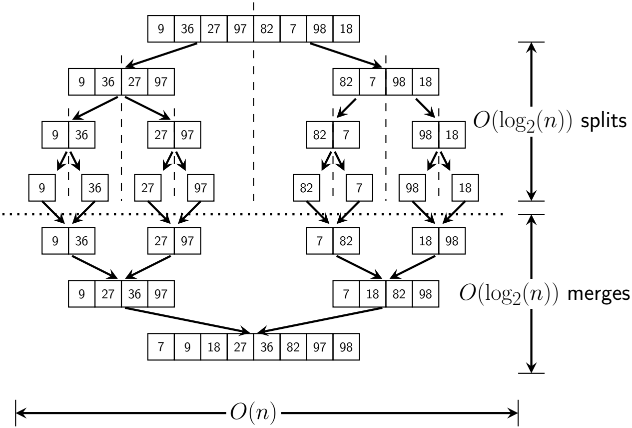
Merge operation



Properties of Merge Sort

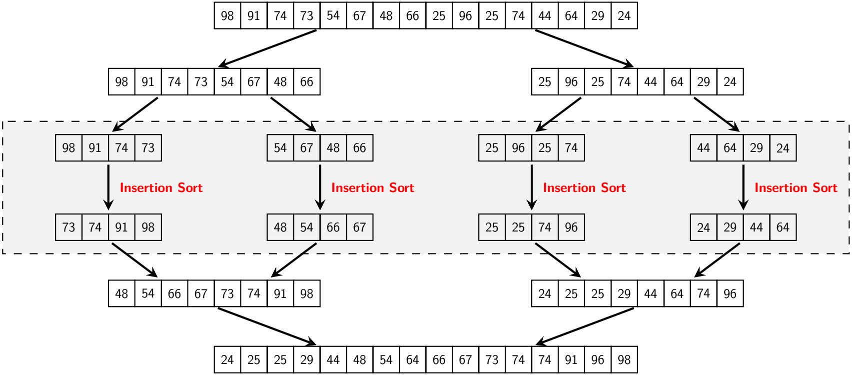
- Merge Sort is **stable**, i.e. it preserves the order of two entries with same value (provided we merge carefully).
- Merge Sort is **not in-place**: we need an array of at most size n to do the merging! (Space complexity is $O(n)$.)
- Merging sub-arrays is **quick**: given two arrays of size n, we need to perform at most $n-1$ comparisons to merge them.
- Recurrence relation: $T(n) = 2T(\frac{n}{2}) + O(n)$.
  - **Worst case** time complexity: $O(n \log(n))$.
- Merge Sort is **asymptotically optimal**.

Complexity of Merge Sort



Improving Merge Sort with Insertion Sort

Main idea: if sub-array size falls below a certain threshold, we switch to Insertion Sort (which is fast for short arrays).



Quicksort

- Invented by **Sir Tony Hoare** in 1959.
- Later implemented in ALGOL-60 (using recursion) and published in 1961.
- One of the most influential algorithms in computer science.
- Improvements made by **Bob Sedgewick** in the 1970s.

Sir Tony Hoare

Basic idea: **divide-and-conquer** by separating the array into two parts depending on whether the elements are smaller or greater than some **pivot** element; recurse on both parts until the array is sorted.

Quicksort

Algorithm 6 Quicksort

procedure $\textsc{QuickSort}(\mathbf{a}, start, end)$
    if $start < end$ **then**
        $pivot \leftarrow \textsc{ChoosePivot}(\mathbf{a}, start, end)$
        $part \leftarrow \textsc{Partition}(\mathbf{a}, pivot, start, end)$
        $\textsc{QuickSort}(\mathbf{a}, start, part - 1)$ ▷ Recurse
        $\textsc{QuickSort}(\mathbf{a}, part + 1, end)$ ▷ Recurse
    else
        return ▷ Base case

Optimising Partitioning

Choose pivot:

$$\mathbf{a} = a_1, a_2, a_3, \ldots, a_{n-1}, \overbrace{a_n}^{p}$$

Partition:

$$\underbrace{a'_1, a'_2, a'_3, \ldots, a'_{m-1}}_{<p}, p, \underbrace{a'_{m+1}, a'_{m+2}, \ldots, a'_n}_{\geq p}$$

There are many different ways of performing partitioning.

- **Worst case scenario**: pivot is the smallest or the largest element (this results in an inefficient partitioning: an array of size $n-1$ and an array of size 1).

Optimising Partitioning

Main question: how to *efficiently* choose the pivot?

Some possibilities:

- Choose the first element in the array.
- Choose the median of the first, middle and last element of the array [Bentley-McIlory, 1993]. (This increases the likelihood of the pivot being close to the median of the whole array.)
- Choose the pivot randomly (makes worst case unlikely).

Quicksort with Insertion Sort

- Idea: We recursively partition the array until each partition is small enough to sort using Insertion Sort.

Algorithm 7 Quicksort

procedure $\textsc{QuickSort}(\mathbf{a}, start, end)$
    if $end - start < threshold$ **then**
        $\textsc{InsertionSort}(\mathbf{a}, start, end)$
    if $start < end$ **then**
        $pivot \leftarrow \textsc{ChoosePivot}(\mathbf{a}, start, end)$
        $part \leftarrow \textsc{Partition}(\mathbf{a}, pivot, start, end)$
        $\textsc{QuickSort}(\mathbf{a}, start, part - 1)$ ▷ Recurse
        $\textsc{QuickSort}(\mathbf{a}, part + 1, end)$ ▷ Recurse
    else
        return ▷ Base case

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	73

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

$\downarrow$ Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	34	**73**	87	92	95	76	87	89

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

$\downarrow$ Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	**34**	**73**	87	92	95	76	87	**89**

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

$\downarrow$ Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	**34**	**73**	87	92	95	76	87	**89**

$\downarrow$ Quicksort $\downarrow$ Quicksort

25	7	29	5	2	**34**	52	67	36	48	66	61	34	**73**	87	87	76	**89**	92	95

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

↓ Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	**34**	**73**	87	92	95	76	87	**89**

↓ Quicksort ↓ Quicksort

25	7	29	5	2	**34**	52	67	36	**48**	66	61	34	**73**	87	87	76	**89**	92	95

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	**34**	**73**	87	92	95	76	87	**89**

Quicksort Quicksort

25	7	29	5	2	**34**	52	67	36	**48**	66	61	34	**73**	87	87	76	**89**	92	95

Quicksort Quicksort **Insertion Sort**

2	7	5	**25**	29	**34**	34	36	**48**	67	66	61	52	**73**	76	87	87	**89**	92	95

Quicksort example

61	66	87	5	34	76	2	67	29	95	89	25	34	7	87	92	48	52	36	**73**

Quicksort

61	66	36	5	34	52	2	67	29	48	7	25	**34**	**73**	87	92	95	76	87	**89**

Quicksort Quicksort

25	7	29	5	2	**34**	52	67	36	**48**	66	61	34	**73**	87	87	76	**89**	92	95

Quicksort Quicksort **Insertion Sort**

2	7	5	**25**	29	**34**	34	36	**48**	67	66	61	52	**73**	76	87	87	**89**	92	95

Insertion Sort

2	7	5	**25**	29	**34**	34	36	**48**	67	66	61	52	**73**	76	87	87	**89**	92	95

Time Complexity of Quicksort

- Partitioning an array of size n takes $\Theta(n)$ operations.
- When the pivot element is the smallest for each partitioning, we need $n-1$ partitioning rounds, i.e. $O(n)$.
- **Worst case**: $O(n^2)$ time complexity.
- Ideally, the pivot is the median value and splits the array in half. In this case we have $\Omega(\log(n))$ partitions.
- On **average**, Quicksort is $O(n \log(n))$.

In **practice**, Quicksort is **very fast** (close to $O(n)$).

- 39% more comparisons than Merge Sort,
- But faster because there is less data movement.

Summary

Sorting is important: one of the most common operations.

- Lower bound on the complexity of comparison-based sorts is: $\boxed{\Omega(n \log_2(n))}$
- We can achieve this optimal bound with efficient comparison-based sorting algorithms.
- Today we've seen two efficient sorting algorithms: Merge Sort and Quicksort.

Further Reading:

1. Merge Sort: Chapter 2 in Cormen (CLRS) **Introduction to Algorithms**.
2. Quicksort: Chapter 7 in Cormen (CLRS) **Introduction to Algorithms**.

Optional material:

1. **Sir Tony Hoare** speaking about his discovery of Quicksort https://www.youtube.com/watch?v=tAl6wzDTrJA&t=13m

Acknowledgements: Partly based on earlier COMP 1201 slides by Drs Adam Prugel-Bennett, Long Tran-Thanh and Baharak Rastegari, University of Southampton.