Chapter 2: Getting Started

A **computational problem** is a mathematical problem, specified by an input/output relation.

An **algorithm** is a computational procedure for solving a computational problem.

Sorting

A well-known example of computational problems is the **sorting** problem, which can be formally specified as follows:

- Given some n numbers $a_1, \ldots, a_n$, compute their enumeration $a'_1, \ldots, a'_n$ such that $a'_1 \leq \cdots \leq a'_n$.

We assume that the numbers to be sorted are given in an array. The objects in the array are sometimes refered to as the **element** and the values with respect to which those elements need to sorted are sometimes refered to as the **keys**. When there is no confusion **the elements are identified as their keys.**

Two Sorting Algorithms

Here we study two sorting algorithms, **Insertion Sort** and **Mergesort**.

Insertion Sort sorts by inserting into a sorted list the elements of the input array one after the other.

Mergesort sorts by recursively dividing the input array into halves, sorting the halves separately, and then merging them into a full sorted list.

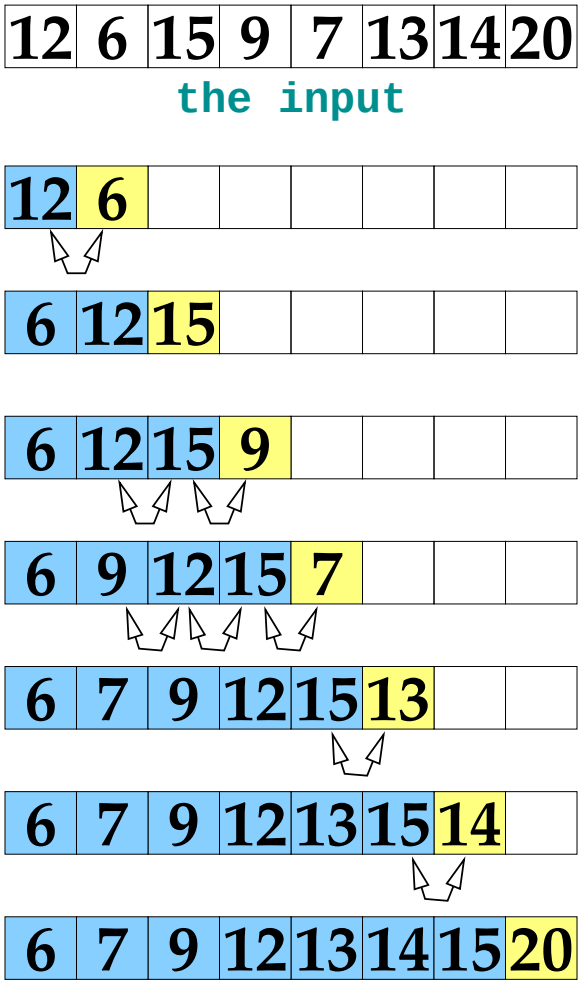
Insertion Sort

Let $A[1\,..\,n]$ be an input array.

Idea: Given an array of size n, obtain for each i, $1 \leq i \leq n$, a completely sorted list of the first i elements. To incorporate a new element find the position at which the new element should be inserted.

```
1: for j ← 2 to n do
2:     ▷ incorporate the j^th element
3: {       x ← A[j]; i ← j − 1
4:         while (i > 0) and (A[i] > x) do
5:         {   A[i + 1] ← A[i]
6:             i ← i − 1 }
7:         A[i + 1] ← x
8: }
```

An Example:


12 6 15 9 7 13 14 20
the input
12 6
6 12 15
6 12 15 9
6 9 12 15 7
6 7 9 12 15 13
6 7 9 12 13 15 14
6 7 9 12 13 14 15 20

Proving Correctness of Algorithms

Goal Identify what property needs to be established at the end of the algorithm and argue that the property is indeed achieved.

For long algorithms it may be necessary to do this by a line of arguments:

- Identify a number of points, $p_1, \ldots, p_m$, in the algorithm and properties corresponding to them, $Q_1, \ldots, Q_m$. Here p_1 and p_m are respectively the beginning and the end of the algorithm.
- Argue that Q_1 holds and that Q_m implies that the algorithm works correctly.
- For each i, $1 \leq i \leq m-1$, argue that the condition Q_i implies Q_{i+1}.

Making Arguments about Branches

Strategy Argue that the cases are exhaustive and each case is correctly handled.

Making Arguments about Loops

Strategy Use a property that is maintained during the execution of the loop. Such a property is called a **loop invariant**. Pick a reference point on the loop (usually either the beginning or the end of the loop-body). Then we show that the following three properties hold:

Initialization The loop invariant holds before the first iteration of the loop.

Maintenance The loop invariant is maintained in each iteration of the loop-body.

Termination Due of the above two properties the loop invariant holds after the last iteration of the loop.

Using a Loop Invariant to Prove the Correctness of Insertion Sort

Loop Invariant At the beginning of the for-loop, the following condition hold:

(*) The subarray $A[1..j-1]$ holds in sorted order the elements that were originally in $A[1..j-1]$.

Initialization

This is easy! At the beginning, $j = 2$.

$A[1 \,..\, j-1]$ is sorted by itself.

So, (*) holds.

Maintenance

Suppose we are at the beginning of the for-loop and (*) holds. Let us look at the subsequent iteration of the loop.

What the loop does is essentially

- **finding the first i in the sequence $j-1, j-2, \ldots, 1$ such that $A[i] \leq A[j]$** and then
- **inserting $A[j]$ right after $A[i]$.**

Since (*) holds at the beginning, this implies that $A[1..j]$ is sorted at the end of the iteration.

Thus, (*) is preserved during one iteration.

Termination

This is easy, too! At the end, $j = n + 1$. So, by (*), $A[1 \,..\, n]$ is sorted. Thus, (*) holds. This completes the proof.

Running-Time Analysis

The running time of an algorithm depends on the actual implementation and the instance, which makes analysis highly complicated. We simplify the process by the following policies:

- Use the number of primitive operations that are executed as the running time.
- Group the instances according to their sizes and analyze the "global behavior" of the "running time" on the instances of the same size.

There are three kinds of analysis:

Worst-Case Analysis For each n, we calculate the largest value of the running time over all instances of size n.

Best-Case Analysis For each n, we calculate the smallest value of the running time over all instances of size n.

Average-Case Analysis For each n, we calculate the average of the running time over all instances of size n where the instances are subject to a certain distribution (for example, the uniform distribution).

Worst-Case & Best-Case Analysis

The running-time of Insertion Sort is a complicated function, but it is a linear function of the array size n and the number of comparisons that are executed.

Since there are at least $n - 1$ comparisons, analyzing the number of comparisons will be sufficient. This number is

- **maximized** when the input numbers are **sorted in the decreasing order** and is
- **minimized** when they are **sorted in the increasing order.**

We use the former to obtain the worse-case running time and the latter to obtain the best-case running time.

Running-Time Analysis (cont'd)

Worst-Case Analysis The number of comparisons is

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

So, the worst-case running time is $\Theta(n^2)$.

Best-Case Analysis The number of comparisons is

$$\sum_{i=1}^{n-1} 1 = n - 1.$$

So, the best-case running time is $\Theta(n)$.

Mergesort

Mergesort is a well-known example of an algorithm design strategy called **divide-and-conquer**.

Divide-and-conquer consists of the following three steps:

Divide Divide the given instance into smaller instances.

Conquer Solve all of the smaller instances.

Combine Combine the outcomes of the smaller instances.

Mergesort in the Divide&Conquer Framework

Divide Split the array into halves

Conquer Sort the halves

Combine Merge the sorted halves into a single sorted list

The Algorithm

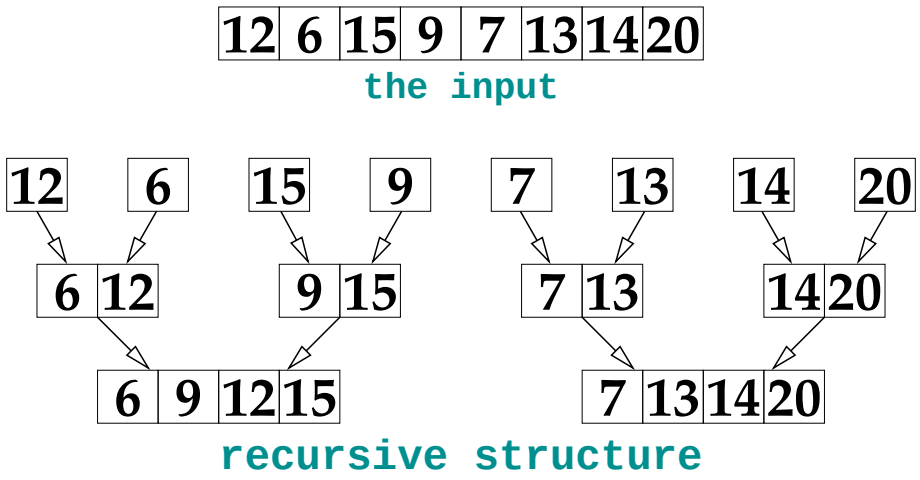
Mergesort(A, p, q)

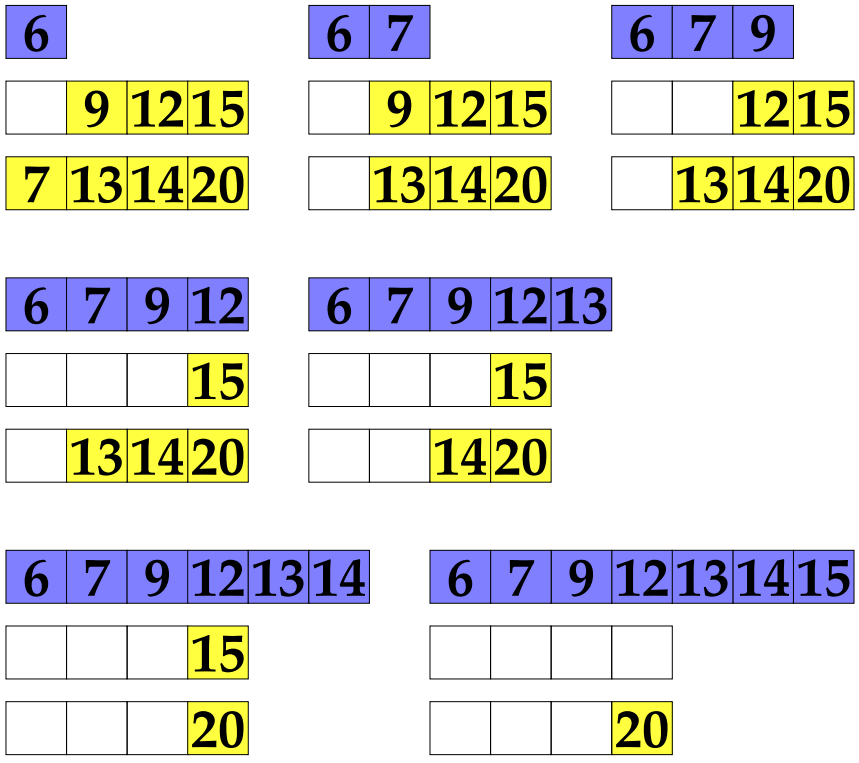
1: ▷ sort $A[p\,..\,q]$
2: $n \leftarrow q - p + 1$; $m \leftarrow p + \lfloor n/2 \rfloor$
3: ▷ Halve the indices
4: **if** $n = 1$ **then return**
5: **Mergesort**$(A, p, m - 1)$
6: ▷ Sort the first half
7: **Mergesort**(A, m, q)
8: ▷ Sort the second half
9: **Merge**(A, p, q, m)
10: ▷ Merge the two sorted lists

Merge(A, p, q, m)

1: ▷ Merge $A[p\,..\,m-1]$ & $A[m\,..\,q]$
2: ▷ into $B[1\,..\,q-p+1]$
3: $i \leftarrow p$; $j \leftarrow m$; $t \leftarrow 1$
4: ▷ Initialization
5: **while** $(i \leq m-1)$ **or** $(j \leq q)$ **do**
6: **if** $j = q+1$ **then**
7: ▷ $A[m\,..\,q]$ has been emptied
8: $\{\ B[t] \leftarrow A[i];\ i \leftarrow i+1;\ t \leftarrow t+1\ \}$
9: **else if** $i = p+1$ **then**
10: ▷ $A[p\,..\,m-1]$ has been emptied
11: $\{\ B[t] \leftarrow A[j];\ j \leftarrow j+1;\ t \leftarrow t+1\ \}$
12: **else if** $A[i] < A[j]$ **then**
13: ▷ Neither have been emptied and $A[i] < A[j]$
14: $\{\ B[t] \leftarrow A[i];\ i \leftarrow i+1;\ t \leftarrow t+1\ \}$
15: **else**
16: ▷ Neither have been emptied and $A[i] \geq A[j]$
17: $\{\ B[t] \leftarrow A[j];\ j \leftarrow j+1;\ t \leftarrow t+1\ \}$
18: **for** $t = 1$ **to** $q-p+1$ **do**
19: $A[p+t-1] \leftarrow B[t]$

An example


12 6 15 9 7 13 14 20
the input
12 6 15 9 7 13 14 20
6 12 9 15 7 13 14 20
6 9 12 15 7 13 14 20
recursive structure


6
9 12 15
7 13 14 20
6 7
9 12 15
13 14 20
6 7 9
12 15
13 14 20
6 7 9 12
15
13 14 20
6 7 9 12 13
15
14 20
6 7 9 12 13 14
15
20
6 7 9 12 13 14 15
20

Proving Correctness of Mergesort

Theorem A For all $n \geq 1$, Mergesort correctly sorts any subarray of size n.

The proof is by **induction** on n.

The base case is when $n = 1$.

How do you argue that the algorithm works correctly on size-one arrays?

One-element arrays are
already sorted by themselves.

Given a subarray of size one,
Mergesort stops without
modifying the subarray.

So, it correctly works when
$n = 1$.

For the induction step, let $n \geq 2$ and suppose:

the claim holds for smaller values of n.

This is our **inductive hypothesis.**

Let (A, p, q) be an input to Mergesort such that $q - p + 1 = n$. Since $n \geq 2$ the last three command lines of the code will be executed:

$$\textbf{Mergesort}(A, p, m-1)$$
$$\textbf{Mergesort}(A, m, q)$$
$$\textbf{Merge}(A, p, q, m)$$

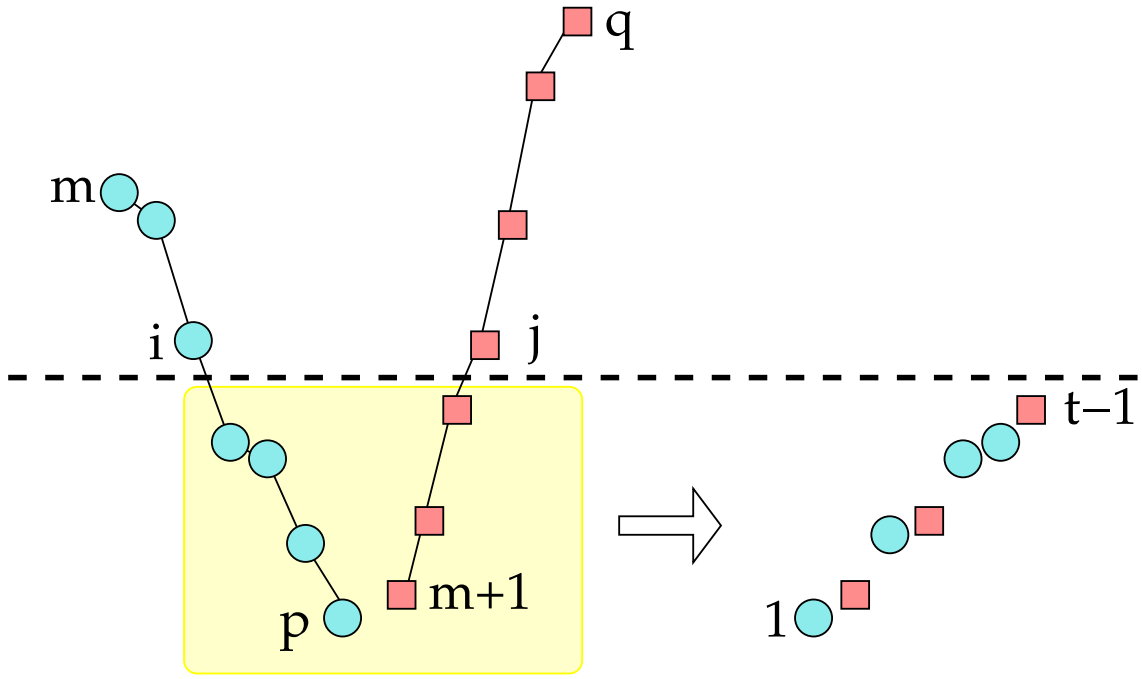
Here $m = p + \lfloor n/2 \rfloor$. Since the two subarrays have size smaller than n, by our inductive hypothesis the first two lines work correctly.

So, it suffices to show that $\textbf{Merge}$ correctly merges two sorted lists.

Using a Loop Invariant to Prove the Correctness of Merge

Loop Invariant At the beginning of the while-loop, the following conditions hold:

1. $B[1\,..\,t-1]$ holds the elements that were originally in $A[p\,..\,i-1]$ and $A[m\,..\,j-1]$.
2. Both $A[i\,..\,m-1]$ and $A[j\,..\,q]$ are sorted.
3. $B[1\,..\,t-1]$ is sorted.
4. Each element in $A[i\,..\,m-1]$ and $A[j\,..\,q]$ is greater than or equal to any element in $B[1\,..\,t-1]$.



Two subarrays of A

Array B

Initialization

At the very beginning
$t = 1$, $i = p$, and $j = m$.
So, (1) holds.
Since $B[1\,..\,t-1]$ is empty,
both (3) and (4) hold.
The two subarrays of A are
sorted, so (2) holds.

Maintenance

We'll examine one-round execution of the loop-body.

Suppose that the execution is **at the beginning of the while-loop**. Suppose that the condition $[(i \leq m-1) \vee (j \leq q)]$ holds and that the loop invariant holds.

Let t', i', and j' be the values of t, i, and j at the beginning of the loop-body, respectively, and let t'', i'', and j'' be their values at the end of the loop-body.

Proving Maintenance

The loop does not modify A, so (2) is preserved.

The property (1) is preserve because during the execution of the loop-body exactly one of $A[i]$ and $A[j]$ is appended to B.

The element appended to B is, by (4), greater than or equal to any element in $B[1\,..\,t-1]$. By (3), this implies that $B[1\,..\,t''-1]$ is sorted. Thus, (3) is preserved.

By (2), the element appended to B is the smallest of the elements in $A[i'\,..\,m-1]$ and $A[j'\,..\,q]$. Since (4) is preserved, the element appended to B is the largest element in the updated B. Thus, each element in $A[i''\,..\,m-1]$ and $A[j''\,..\,q]$ is greater than or equal to any element in $B[1\,..\,t''-1]$. Thus, (4) is preserved.

Termination

By (1), $B[1\,..\,t-1]$ consists of the elements in $A[p\,..\,q]$.

By (3), this array B is sorted.

Now, in the very last two lines, the contents of B are copied to A while keeping the order.

Thus, the algorithm works correctly.

Running-Time Analysis of Mergesort

Let $T(n)$ be the running time of Mergesort. We may assume that for all n it holds that $T(n+1) \geq T(n)$. T is a complicated function, but we observe that it is linear in the number of "assignment" operations performed on the arrays A and B. This number is $2n$ for merging. So,

$$T(n) \leq \alpha n + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

The ceiling and the floor functions make the analysis cumbersome.

To get rid of them, let $m = 2^{\lceil \log n \rceil}$. Then $m \leq 2n$ and $T(n) \leq T(m)$. We'll obtain an upper bound for $T(m)$.

Each divisor of m is a power of 2. So, for each $d \geq 2$ that divides m,
$\lceil m/2 \rceil = \lfloor m/2 \rfloor = m/2$.

Running-Time Analysis (cont'd)

We have

$$\begin{aligned} & T(m) \\ & \leq \quad \alpha m + 2T(m/2) \\ & \leq \quad \alpha m + 2(\alpha m/2 + 2T(m/2)) \\ & = \quad 2\alpha m + 4T(m/4) \\ & \leq \quad 2\alpha m + 4(\alpha m/4 + 2T(m/4)) \\ & \leq \quad 3\alpha m + 8T(m/8) \\ & \leq \quad \cdots \\ & \leq \quad \alpha m \log m + mT(1) \\ & = \quad \alpha' m \log m \end{aligned}$$

So, $T(n) = O(m \log m) = O(n \log n)$.

By using m the floor function in place of ceiling function and by using $>$ in place of $<$, we have that $T(n) = \Omega(n \log n)$.

Comparing the Two Algorithms

Suppose that $n(n+1)/4$ is the running time of Insertion Sort and $3n \log n$ is the running time of Mergesort

n	Insertion Sort	Mergesort
10	25	9.3
10^2	2.5×10^3	1.8×10^2
10^3	2.5×10^5	2.7×10^3
10^4	2.5×10^7	3.6×10^4
10^5	2.5×10^9	4.5×10^5
10^6	2.5×10^{11}	5.4×10^6
10^7	2.5×10^{13}	6.3×10^7

The gap grows as n grows!