Algorithms: Sorting + (Time) Analysis

Alessandro Chiesa, Ola Svensson

EPFL

School of Computer and Communication Sciences

Lecture 2, 19.02.2025

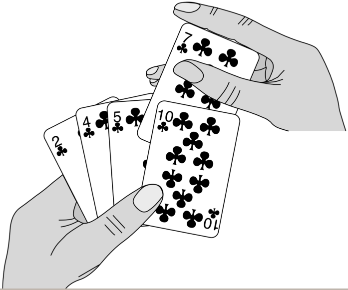
Recall Last Lecture

- CS-250: A lot of interesting and useful material!
- A computational problem is defined by an input/output relationship
  - Example: INPUT: n OUTPUT: $\sum_{i=1}^{n} i$
- An algorithm describes a specific computational procedure for achieving that input/output relationship
  - Example: return $n(n+1)/2$
- "Time + Space" is crucial for the usefulness of an algorithm

The Growth of Functions

- Three algorithms: A, B, C with different running times in ms.

	A ($1000 \log_2 n$ ms)	B ($2n^2$ ms)	C (2^n ms)



SORTING

Insertion Sort

The sorting problem

Definition

INPUT: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

OUTPUT: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

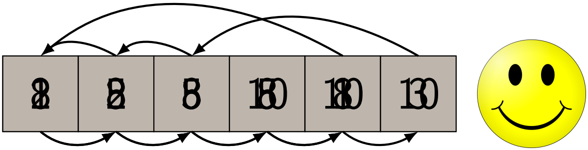
For example

- Given the input $\langle 5, 2, 4, 6, 1, 3 \rangle$
- a correct output is $\langle 1, 2, 3, 4, 5, 6 \rangle$

Insertion Sort - The Idea

Like sorting a hand of playing cards

- Start with an empty left hand of playing cards and the cards face down on the table
- Then remove one card at a time from the table, and insert it into the correct position in the left hand
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards, held in the left hand are sorted, and these cards were originally the top cards of the pile on the table

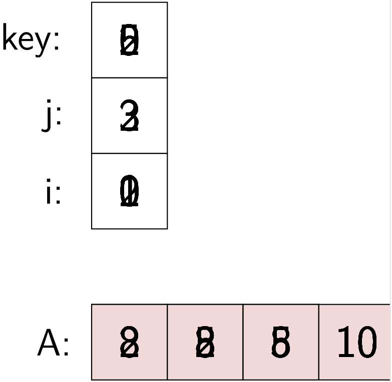


- Takes as parameters an array $A[1 \dots n]$ and the length n of the array

```
INSERTION-SORT(A, n)
 for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
           A[i + 1] = A[i]
           i = i − 1
      A[i + 1] = key
```

Insertion Sort

Example on $\langle 8, 2, 5, 10, 1, 3 \rangle$



And s

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```


```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
```

PROVING ALGORITHMS CORRECT

Loop invariants

Insertion Sort

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Loop invariant:

At the start of each iteration of the "outer" **for** loop – the loop indexed by j– the subarray $A[1 \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$ but in sorted order.

Need to verify:

Similar to induction

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant — usually along with the reason that the loop terminated — gives us a useful property that helps show that the algorithm is correct.

Insertion Sort

At the start of each iteration of the "outer" **for** loop – the loop indexed by j– the subarrary $A[1 \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$ but in sorted order.

```
INSERTION-SORT(A, n)
 for j = 2 to n
     key = A[j]
     // Insert A[j] into the sorted sequence A[1..j-1].
     i = j - 1
     while i > 0 and A[i] > key
         A[i + 1] = A[i]
         i = i - 1
     A[i + 1] = key
```

Initialization

- Before the first iteration of the loop we have $j = 2$.
- The subarray $A[1 \ldots j-1]$, therefore, consists of just the single element $A[1]$
- This is the original element in $A[1]$ and trivially sorted

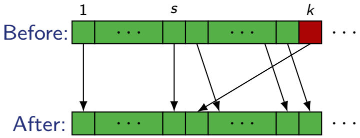
Insertion Sort

At the start of each iteration of the “outer” **for** loop – the loop indexed by j– the subarrary $A[1 \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$ but in sorted order.

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Maintenance:

- Assume invariant holds at the beginning of the iteration when $j = k$, i.e., for $A[1 \ldots k-1]$
- The body of the **for** loop works by moving $A[k-1], A[k-2]$ and so on one step to the right until it finds the proper position for $A[k]$, at which point it inserts the value of $A[k]$



s is largest index in $\{1, \ldots, k-1\}$ such that $A[s] \leq A[j] = A[k]$

or 0 if no such index exists

- The subarray $A[1 \ldots k]$ then consists of the elements originally in $A[1 \ldots k]$ in a sorted order. Incrementing j (to $k+1$) for the next iteration of the **for** loop then preserves the loop invariant :)

Insertion Sort

At the start of each iteration of the "outer" **for** loop – the loop indexed by j– the subarrary $A[1 \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$ but in sorted order.

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Termination

- The condition of the **for** loop to terminate is that $j > n$
- Hence, $j = n + 1$ when loop terminates
- The loop invariant then implies that $A[1 \ldots n]$ contain the original elements in sorted order

ANALYZING ALGORITHMS

Computational Model

We want to predict the resources that the algorithm requires. Usually, running time.

For that we need a computational model

Random-access machine (RAM) model

- Instructions are executed one after another
- Simplification basic instructions take constant (O(1)) time
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return
- We don't worry about precision, although it is crucial in certain numerical applications

Analyzing an algorithm's running time (1/2)

Time it takes depend on the input

- Sorting 1000 numbers take longer than sorting 3 numbers
- A given sorting algorithm may even take different amounts of time on two inputs of the same size

Input size: depends on the problem being studied

- Usually, the number of items in the input. Like the size n of the array being sorted
- If multiplying two integers, could be the total number of bits in the two integers
- Could be described by more than one number: e.g. graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Analyzing an algorithm's running time (2/2)

Running time: on a particular input, it is the number of primitive operations (steps) executed

- Figure that each line of pseudocode requires a constant amount of time
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i
- This is assuming that the line consists only of primitive operations
  - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine might not
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by x-coordinate"

Analysis of insertion sort

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

number of times line executed based on the value of j

Best case: The array is already sorted

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = \Theta(n)$$

A note on Worst-case analysis

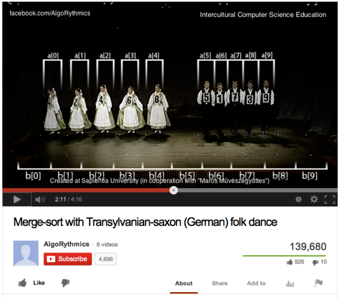
We usually concentrate on finding the **worst-case running time:** the longest running time for *any* input of size n

Reasons:

- Gives a guaranteed upper bound on the running time for any input
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present
- Average case often as bad as worst-case: Suppose that we randomly choose n numbers as the input to insertion sort

Order of growth: Focus on the important features

- Drop lower-order terms
- Ignore the constant coefficient in the leading term



SORTING BY DIVIDE-AND-CONQUER

Merge Sort

Divide-and-Conquer

Powerful algorithmic approach:

recursively divide problem into smaller subproblems

Divide-and-Conquer

Powerful algorithmic approach:

recursively divide problem into smaller subproblems

Divide the problem into a number of subproblems that are smaller instances of the same problem

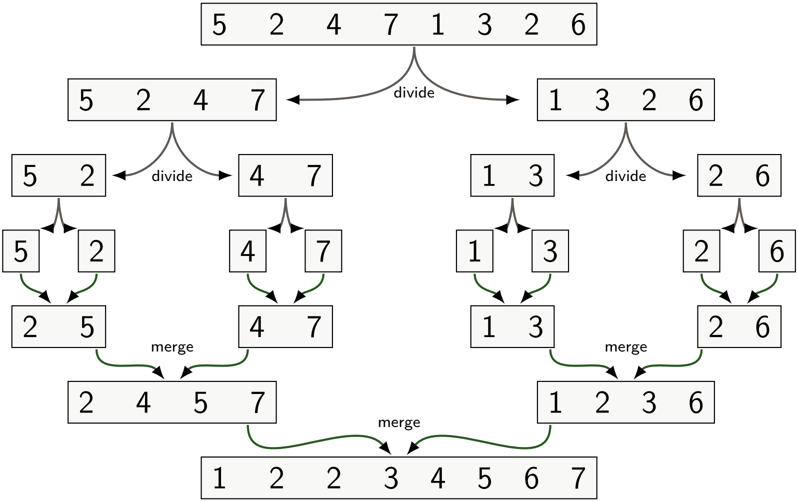
Conquer the subproblems by solving them recursively.

Base case: If the subproblems are small enough, just solve them by brute force

Combine the subproblem solutions to give a solution to the original problem

Merge Sort = D & C applied to sorting

Example $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$



Merge sort

To sort $A[p \dots r]$:

Divide by splitting into two subarrays $A[p \dots q]$ and $A[q+1, \dots, r]$, where q is the halfway point of $A[p \dots r]$

Conquer by recursively sorting the two subarrays $A[p \dots q]$ and $A[q+1, \dots r]$

Combine by merging the two sorted subarrays $A[p \dots q]$ and $A[q+1, \dots, r]$ to produce a singe sorted subarray $A[p \dots r]$

```
MERGE-SORT(A, p, r)
 if p < r                          // check for base case
     q = ⌊(p + r)/2⌋               // divide
     MERGE-SORT(A, p, q)           // conquer
     MERGE-SORT(A, q + 1, r)       // conquer
     MERGE(A, p, q, r)             // combine
```

Merging

What remains is the Merge procedure to solve the "merge" problem:

Definition

INPUT: Array A and indices $p \leq q < r$ such that subarrays $A[p \dots q], A[q+1 \dots r]$ are sorted.

OUTPUT: The two subarrays are merged into a single sorted subarray in $A[p \dots r]$.

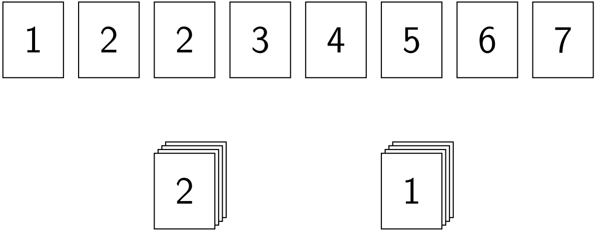
We will give a procedure that solves this problem in time $\Theta(n)$ where n is the size of the subproblem, i.e.,

$$n = r - p + 1$$

Idea behind linear-time merging

Think of two pile of cards that are placed face up

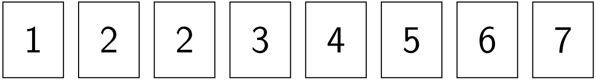
- Basic step: pick the smaller of the two cards and place it in the output pile



Idea behind linear-time merging

Think of two pile of cards that are placed face up

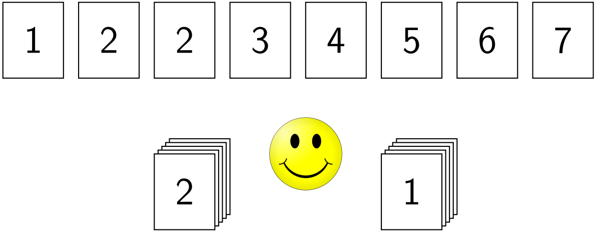
- Basic step: pick the smaller of the two cards and place it in the output pile
- There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input pile
- Therefore the procedure should take $\theta(n)$ time



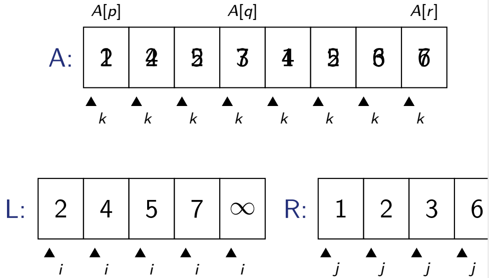
Implementation Simplification

Instead of checking whether a pile is empty:

- Put in the bottom of each input pile a special **sentinel** card of value ∞
- Stop once we have performed $n = r - p + 1$ basic steps (picked n cards)



Merging Algorithm




```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```


```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
```

Merging Algorithm

- Runtime analysis?

MERGE(A, p, q, r)

```
n_1 = q - p + 1
n_2 = r - q
let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i - 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Analyzing divide-and-conquer algorithms

Use a recurrence equation to describe the running time:

- Let $T(n) =$ "running time on a problem of size n"
- If n is small enough say $n \leq c$ for some constant c then $T(n) = \Theta(1)$ (by brute force)
- Otherwise, suppose we divide into a sub problems each of size n/b.
- Let $D(n)$ be the time to divide and let $C(n)$ the time to combine solutions.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Analysis of Merge Sort

```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```

Divide: takes constant time, i.e., $D(n) = \Theta(1)$

Conquer: recursively solve two subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: Merge on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

Summary

- Solving the recurrence for merge sort shows that it runs in time $\Theta(n \log n)$, i.e., much faster than Insertion sort for large instances
- For small instances insertion sort can still be faster
- Insertion sort is also **in place**: the numbers are rearranged within the array (with at most a constant number outside the array at any time)
- Merge sort is not in place!