Algorithms - Ch2 - Sorting

Goals:

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of "divide and conquer" in the context of merge sort.

Analyzing Algorithms

predicting resources that an algorithm requires
- memory, communication bandwidth, hardware, computational time
- compare several candidate algorithms, identify most efficient one

Analyzing Algorithms

one-processor, random-access machine (RAM)

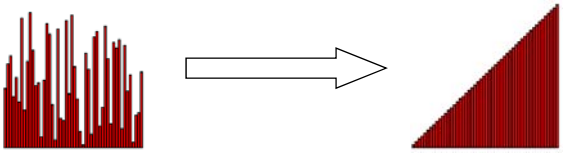
- instructions executed one after another, no concurrent operations
- common instructions:
  - arithmetic (+, -, *, /, %, $\lfloor\ \rfloor$, $\lceil\ \rceil$)
  - data movement (load, store, copy)
  - control (branch, subroutine call and return)
- each of these instructions takes a constant amount of time
- do not consider memory hierarchy

mathematic tools include combinatorics, probability theory, identify most significant terms in a formula

Sorting

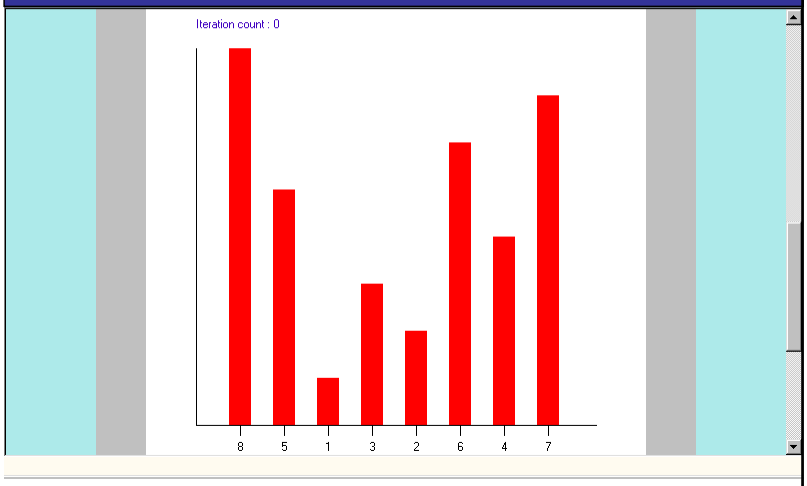
Input: a sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: a permutation (reordering) of $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots a'_n$.



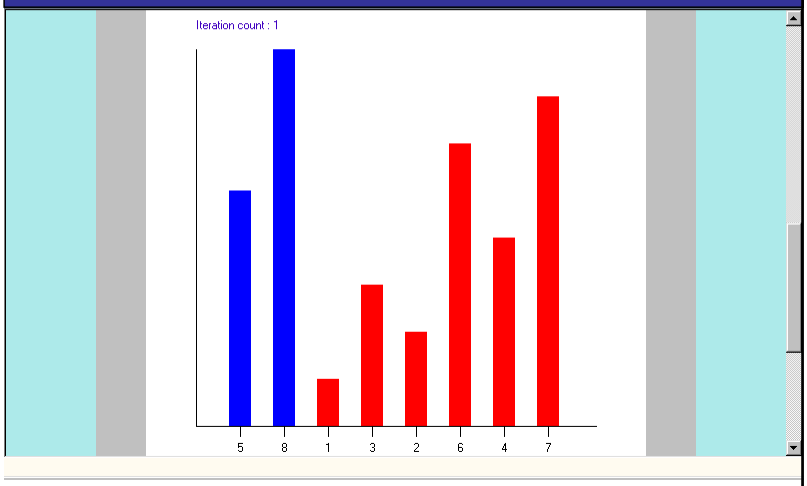
Insertion Sort Animation

Finding a place for item with value 5 in position 1:
Swap item in position 0 with item in position 1.



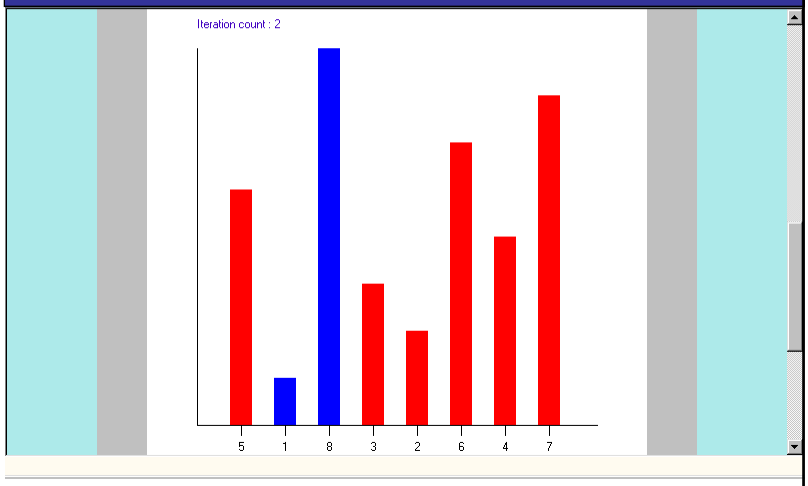
Insertion Sort Animation

Positions 0 through 1 are now in non-decreasing order.



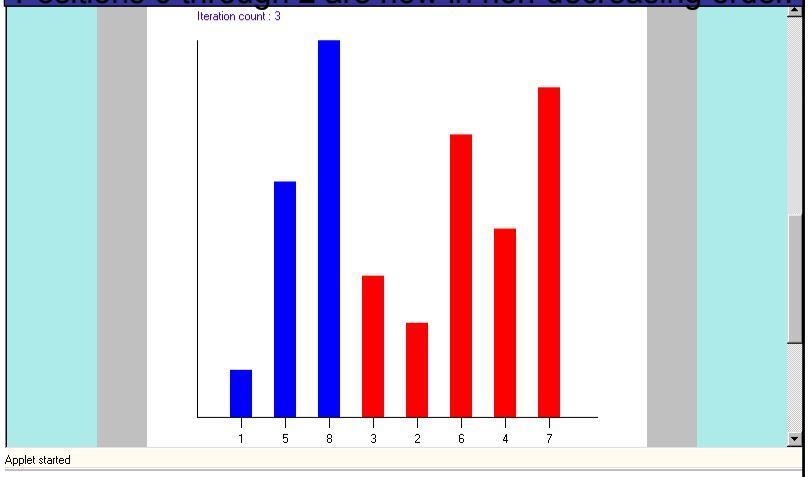
Insertion Sort Animation

Finding a place for item with value 1 in position 2:
Swap item in position 1 with item in position 2.



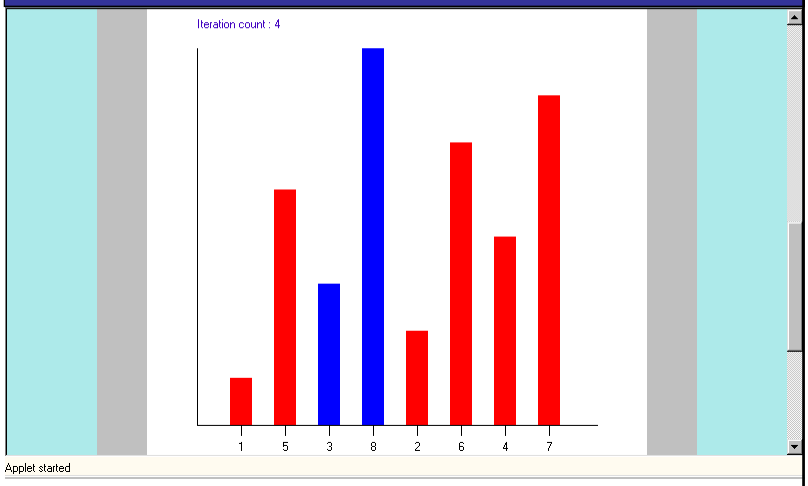
Insertion Sort Animation

Finding a place for item with value 1:
Swap item in position 0 with item in position 1.
Positions 0 through 2 are now in non-decreasing order.



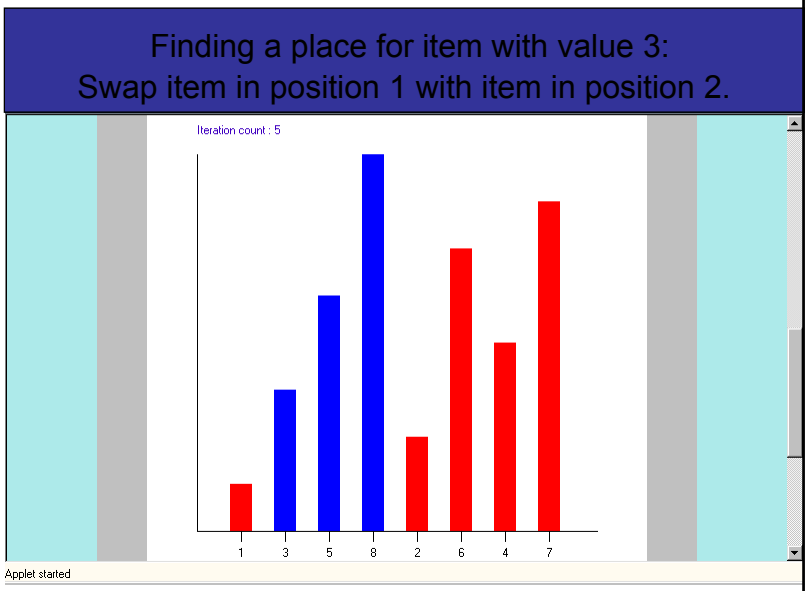
Insertion Sort Animation

Finding a place for item with value 3 in position 3:
Swap item in position 2 with item in position 3.

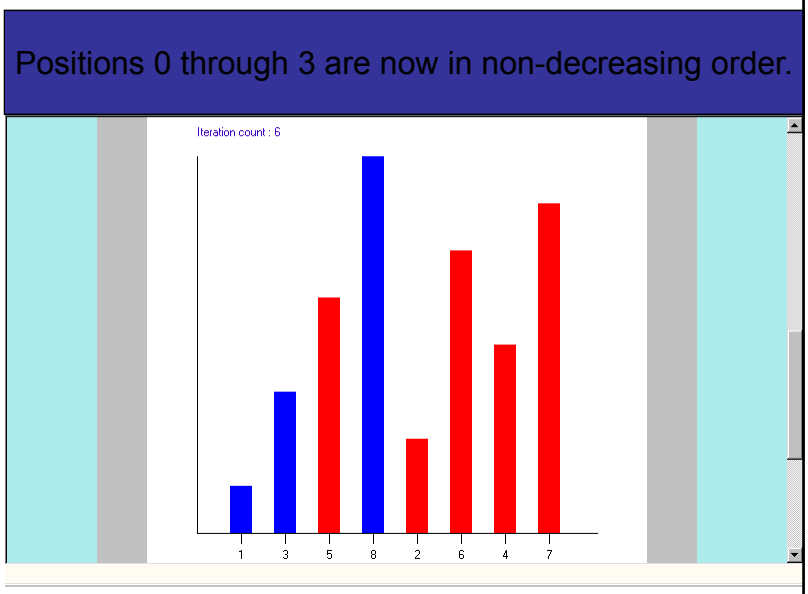


Insertion Sort Animation

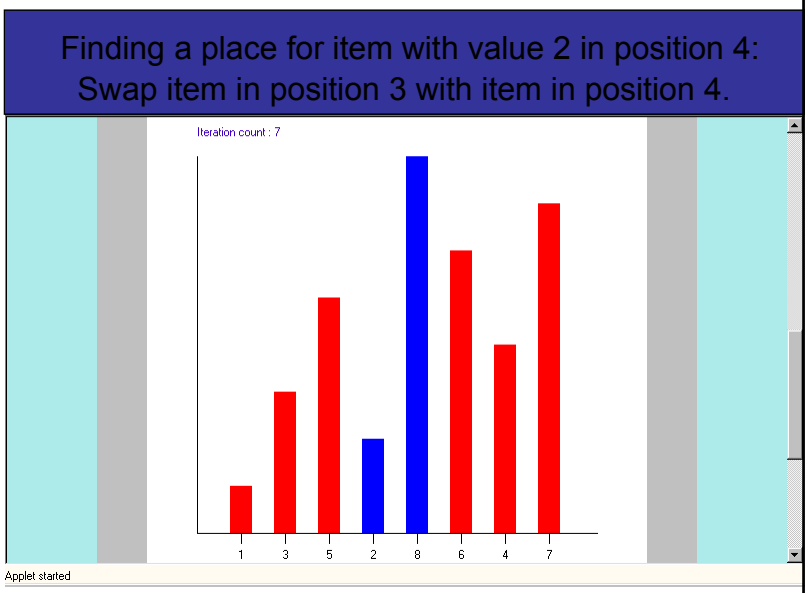
Finding a place for item with value 3:
Swap item in position 1 with item in position 2.



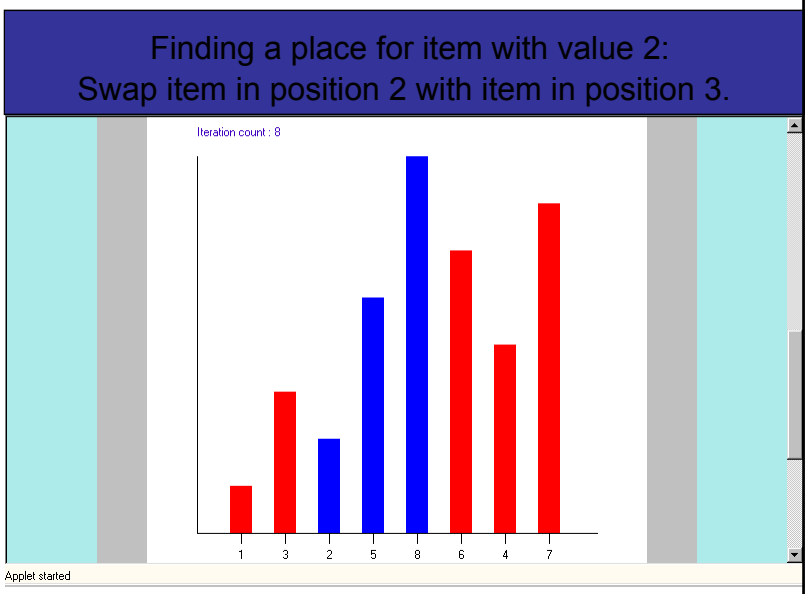
Insertion Sort Animation



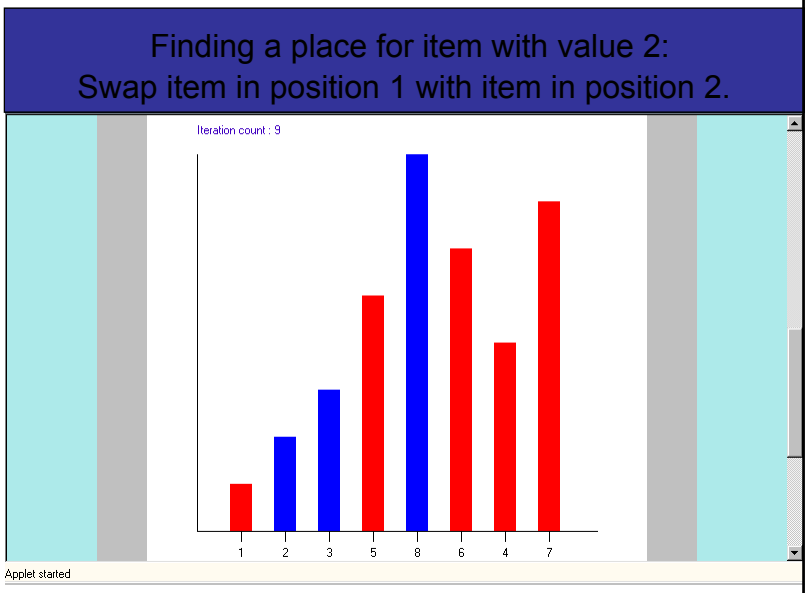
Insertion Sort Animation



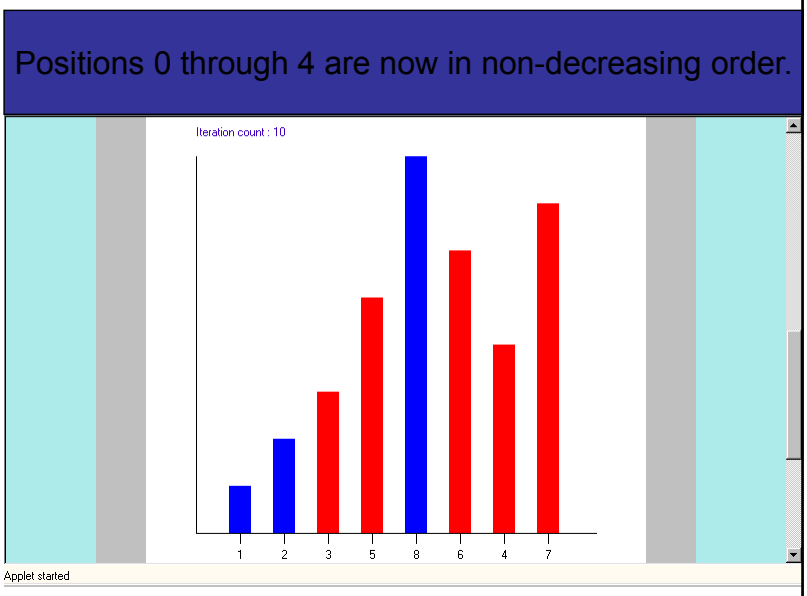
Insertion Sort Animation



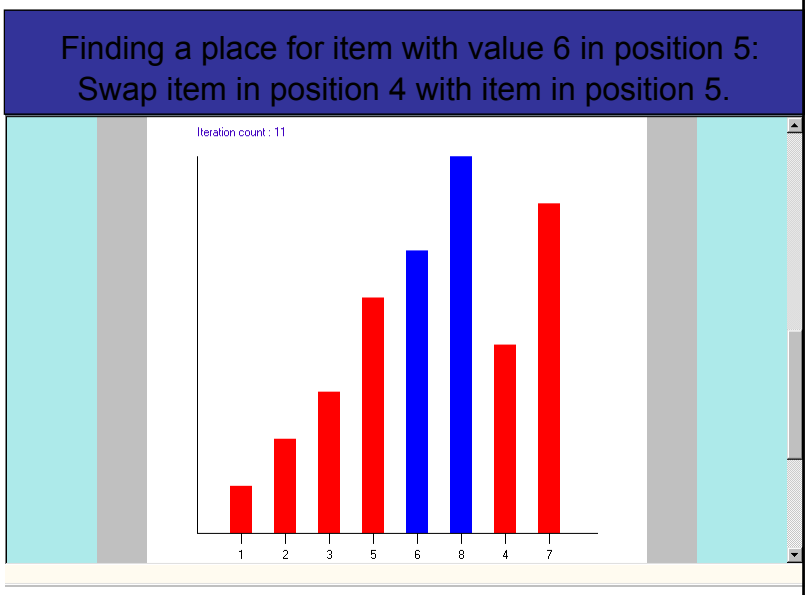
Insertion Sort Animation



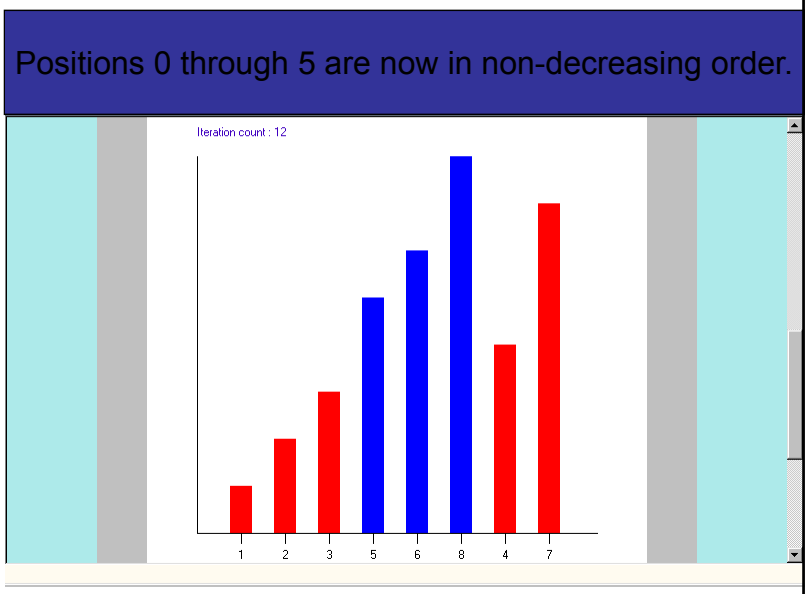
Insertion Sort Animation



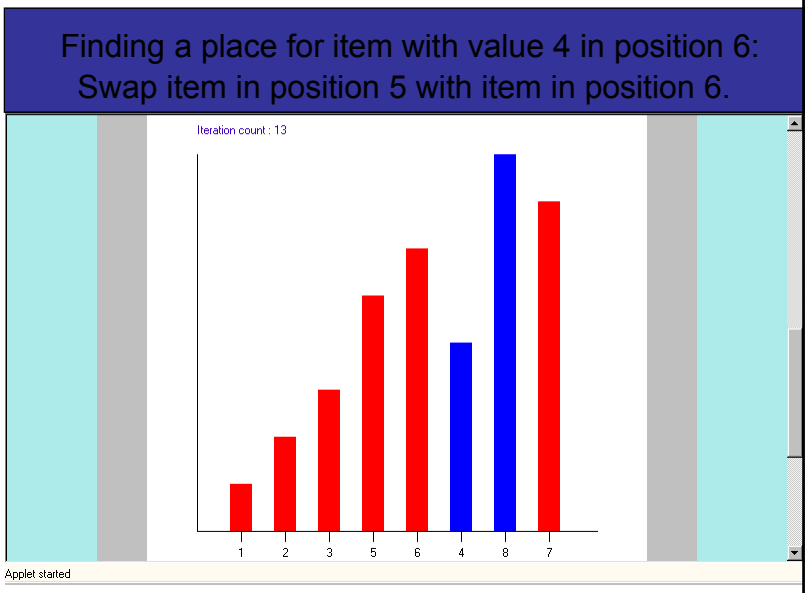
Insertion Sort Animation



Insertion Sort Animation

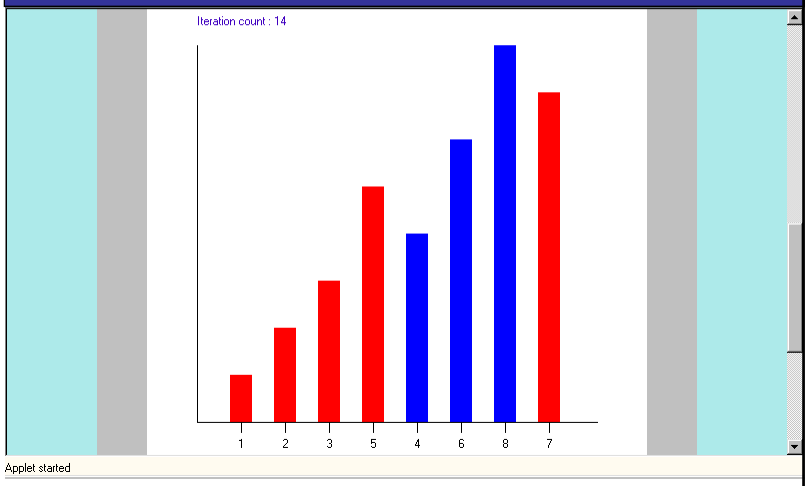


Insertion Sort Animation



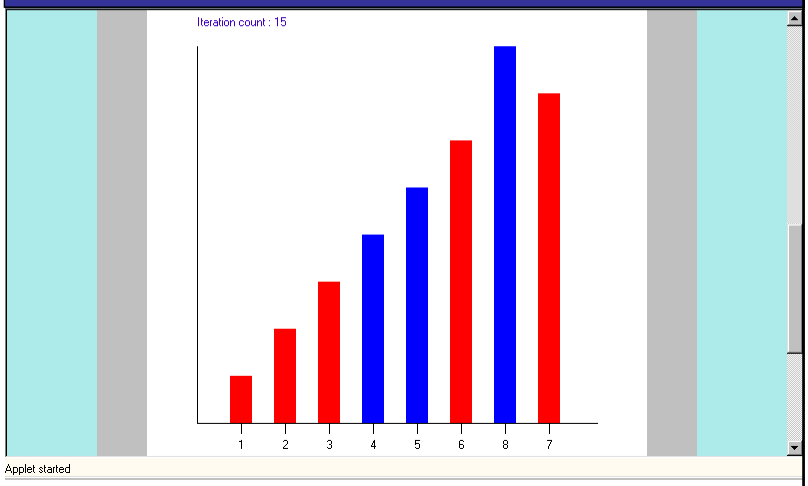
Insertion Sort Animation

Finding a place for item with value 4:
Swap item in position 4 with item in position 5.



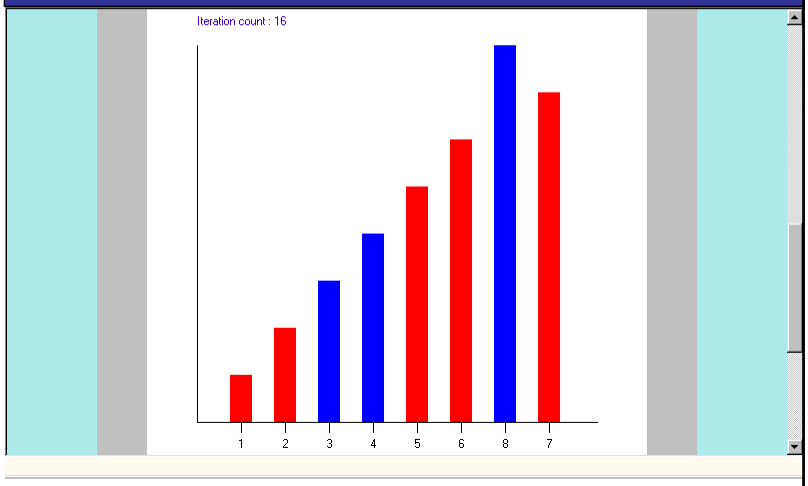
Insertion Sort Animation

Positions 0 through 6 are now in non-decreasing order.



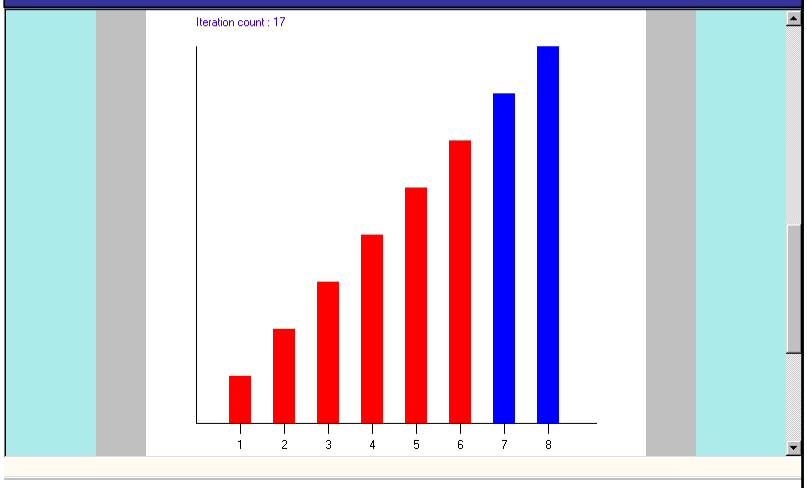
Insertion Sort Animation

Finding a place for item with value 7 in position 7:
Swap item in position 6 with item in position 7.



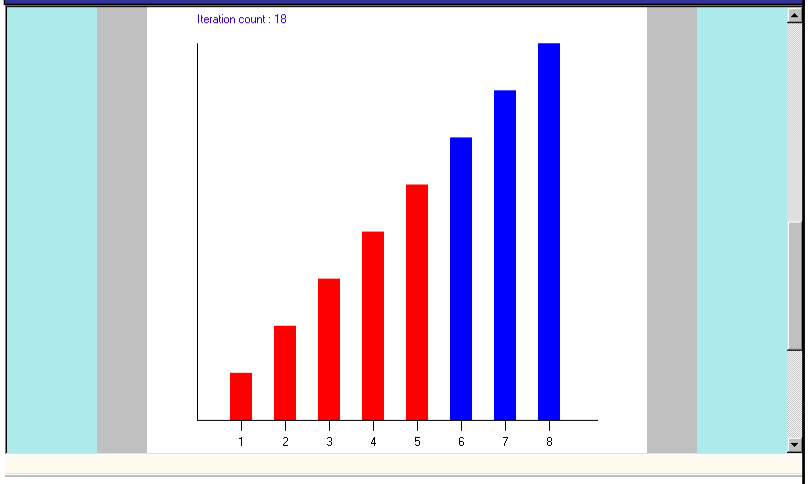
Insertion Sort Animation

Positions 0 through 7 are now in non-decreasing order.



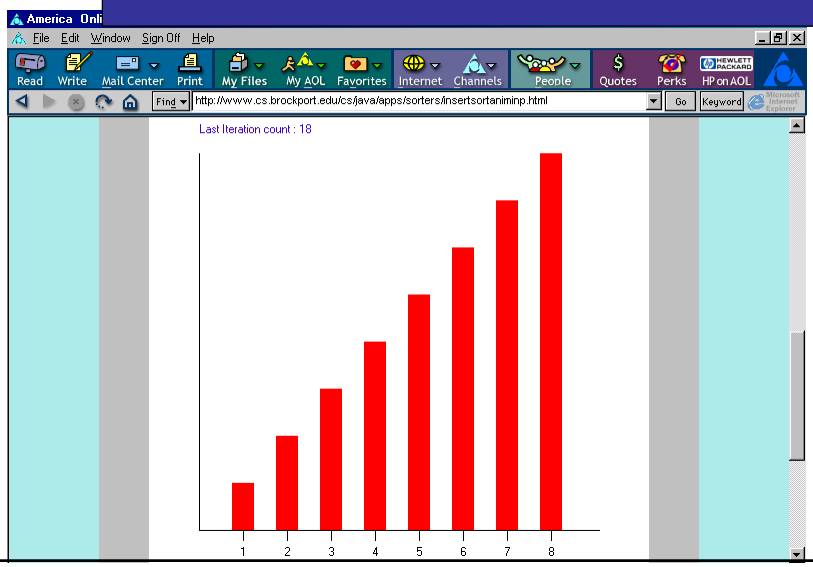
Insertion Sort Animation

Positions 0 through 7 are now in non-decreasing order.



Insertion Sort Animation

Positions 0 through 7 are now in non-decreasing order.



Algorithms - Ch2 - Sorting

Here is the algorithm for Insertion Sort

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

Algorithms - Ch2 - Sorting

Correctness - how?

Loop Invariants: At the start of each iteration of the **for** loops of lines 1-8, the sub-array A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order.

1. Initialization: The statement is true before the first execution of the body of the loop.
2. Maintenance: If the statement is true before any iteration of the loop, it remains true before the next iteration
3. Termination: At loop termination, the statement can be used to derive a property that helps to show the algorithm correct.

Algorithms - Ch2 - Sorting

1. Initialization: since j = 2, the supposedly sorted part of the array is A = A[1..j-1] = A[1..1]: any array of ONE element is sorted (obvious, right?).

2. Maintenance: at the beginning of each iteration we have a sorted array A[1..j-1], and an element A[j] that is to be inserted in the correct position, possibly moving some elements in A[1..j-1]. We have to convince ourselves that this DOES the job we want: A[1..j] IS sorted before the next iteration. Does it? Keep challenging it until you are completely convinced it works.

3. Termination: j = n + 1. But, by part 2, A[1..n] IS SORTED.

Algorithms - Ch2 - Sorting

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

Algorithms - Ch2 - Sorting

- Running time: on a particular input, it is the number of primitive operations (steps) executed
- want to define steps to be machine-independent
- 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 same line takes same amount of time
- Assume a line consists only of primitive operations
  - Subroutine call: actual call takes constant time, execution of subroutine being called might not
  - if line specifies operations other than primitive calls, it might take more than constant time, e.g., “sort the points by x-coordinate”

Algorithms - Ch2 - Sorting

- Analysis…Add things up, to get T(n), the total time.
- Look at the best case (array already sorted in the order you wanted it): the while loop test gets done only ONCE for each iteration of the for loop ($t_j = 1$). Thus each iteration of the for loop takes a constant amount of time (add up all the constants with their repetition). Total time is a linear function of n.
- Look at the worst case (array sorted in inverse order): the while loop tests are carried out the maximum possible number of times ($t_j = j$). , with its body executed just once less. Total time is a quadratic function of n.

Algorithms - Ch2 - Sorting

Why worst-case analysis

- worst case running time gives a guaranteed upper bound on running time for any input
- for some algorithms, worst case occurs often, e.g., search for an item that does not exist
- average case often about as bad as the worst case

Algorithms - Ch2 - Sorting

• Look at the average case - if you can figure out what it is… The average case is, probably, the most important metric - especially if you can prove that the probability your data will ever be presented to you in "worst case format" is small… After all, why should you care much if the worst case is only remotely likely, and the behavior of your algorithm on all "real" cases is very good???

Unfortunately, like all good questions, this is very hard to answer…

In this case, we can do it, and we conclude that the (now) expected time T(n) is still given by a quadratic function of n.

Algorithms - Ch2 - Sorting

Orders of growth

- Abstraction of ease analysis and focus on important features
- Look only at leading term of the formula for running time
  - drop lower-order terms
  - ignore constant coefficient in leading term

Example: $T(n) = an^2+bn+c = \Theta(n^2)$

- Will look in more details at orders of growth and their notation: O, o, Ω, ω, and Θ.

Algorithms - Ch2 - Sorting

Divide and Conquer

The previous algorithm uses a method that could be called "incremental" since it solves a problem "one step at a time": given a problem with an input of size n, we solve it for size 1, use that solution to provide a solutions for size 2, and so on…

A second method involves splitting the input of size n into two sets, solving the problem, independently, for the two sets, and then gluing the two solved problems together in such a way that we solve the original problem. This method is called "divide and conquer" - for obvious reasons…

Note: when is divide and conquer the same as the incremental method?

Algorithms - Ch2 - Sorting

Divide and Conquer

1. Divide the problem into two or more subproblems
2. Conquer each subproblem using recursion (= apply the same method until the size of the subproblem is 1 or small enough to be worth solving by a direct method)
3. Combine all solutions to the subproblems into a solution for the original problem.

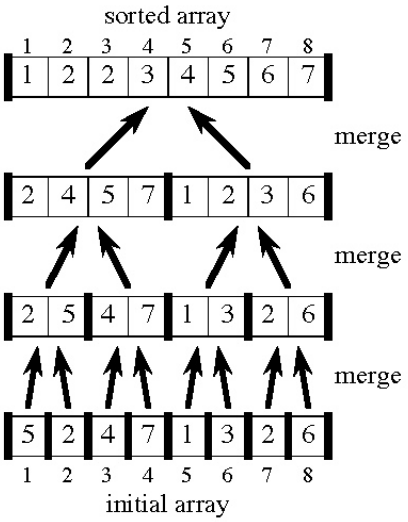
Algorithms - Ch2 - Sorting

Divide and Conquer: MergeSort.

This is the "classic example" of a divide and conquer solution to the sorting problem. Start with an array A of size n - the formal parameters below expect actual values between 1 and n. The initial call will be MERGE-SORT$(A, 1, n)$:

```
MERGE-SORT(A, p, r)
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

Algorithms - Ch2 - Sorting



Algorithms - Ch2 - Sorting

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

Algorithms - Ch2 - Sorting

1. A[p..q] and A[q+1..r] are two contiguous sorted subarrays of A[1..n].
2. They are copied into new arrays L[1..q-p+1] and R[1..r-q].
3. Furthermore L[q-p+2] and R[r-q+1] receive, say, MaxInt (for convenience - whatever can be used for ∞ in this case).
4. L[1] and R[1] are the smallest elements of the subarrays that have NOT yet been recopied into A.
5. At the next pass of the loop (that increments i or j), A[p] will contain a smallest element of L and R, and the new L[i] and L[j] are the smallest elements of the subarrays that have NOT yet been recopied into A.
6. Repeat until the loop terminates…

Algorithms - Ch2 - Sorting

What's a loop invariant for this algorithm?

Here are the two parts of this algorithm:

1. The initial call to Merge-Sort: this involves a check for size and, if passed, recursive calls on the two halves, followed by a call to Merge.
2. The call to Merge, which involves a loop.

So the whole thing requires a bit more than just a "loop invariant".

- "Recursive" Induction - assuming the Merge is correct - will allow us to conclude the full Merge-Sort is correct

1. Loop invariant (essentially a "linear induction") to prove that Merge is correct.

Algorithms - Ch2 - Sorting

"Recursive" induction

1. Base Cases:
   1. The array is empty: clearly sorted and the body of Merge-Sort - after the test - is a NoOp.
   2. The array has one element: n=1, the array is clearly sorted and the body of Merge-Sort - after the test - is a NoOp.
   3. The array has two elements: n = 2, q = 1, and the body consists of recursive calls on subarrays of size 1, sorted by Case 2 above. We assume Merge correct, so the result is correct
2. Inductive Case:
   1. The array has n ≥ 3 elements. The split results into two subarrays each of size < n. The calls to Merge-Sort result in sorted subarrays by the induction hypothesis; the final result is correct by the correctness of Merge.

Algorithms - Ch2 - Sorting

Loop Invariant for Merge

The loop invariant:

At the start of each iteration of the **for** loop, the subarray A[p,k-1] contains the k-p smallest elements of L[1 .. n1+1] and R[1..n2+1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Algorithms - Ch2 - Sorting

Loop Invariant for Merge: Three Phases

1. **Initialization**. Before the first pass through the loop, we have k = p -- the subarray A[p..k-1] is empty. This empty subarray contains the k-p=0 smallest elements of L and R, and, since i = j = 1, L[i] and R[j] are the smallest elements of their arrays not yet copied back into A.
2. **Maintenance**. Each iteration maintains the loop invariant.
   1. Suppose L[i] ≤ R[j]. Then L[i] is the smallest element not yet copied into A. A[p..k-1] contains (by induction) the k-p smallest elements, so the copying (**then** branch of the conditional) will ensure that A[p..k] will contain the k-p+1 smallest elements. Incrementing k and I re-establishes the loop invariant for the next iteration
   2. Suppose L[i] > R[j]. The **else** branch of the conditional will copy R[j] and increment k and j, maintaining the loop invariant.

Algorithms - Ch2 - Sorting

Loop Invariant for Merge: Three Phases

3. **Termination**. k = r+1. By the loop invariant, the array A[p..k-1] = A[p..r] contains the k-p = r-p+1 smallest elements of L[1..n_1+1] and R[1..n_2+1], in sorted order. The arrays L and R together contain n_{1} + n_{2} + 2 = r - p + 3 elements. All but the two largest have been copied back into A, and these two largest elements are sentinels, not in A to begin with.

Algorithms - Ch2 - Sorting

Analyzing MergeSort: Divide and Conquer

Recursive calls can often be analyzed via ***recurrence equations***.

Such equations describe the running time on a problem of size n in terms of the running times on smaller problems.

- Assume that we start with a set of n elements,
- that we break this set of n elements into a subsets, each of n/b items. In many cases a = b, but we don't need it (and algorithms that manipulate pictures need to "bleed over" so that each subpicture has at least two extra rows and 2 extra columns),
- and that we keep breaking the subsets in the same ratios until we reach a size at which we solve the problem directly.

Algorithms - Ch2 - Sorting

Analyzing MergeSort: Divide and Conquer

We start the analysis from the last bullet: if T(n) denotes the time to run the algorithm on n elements, we must have:

- $T(n) = \Theta(1)$ if $n \leq c$ (for some appropriate "small" size c).

Now we look at the larger sets (the two prior bullets):

- $T(n) = a\,T(n/b) + D(n) + C(n)$

Where D(n) is the cost of carrying out the divide operation, and C(n) is the cost of the Combine one. Hopefully those costs will be $\Theta(n)$ or less…

Algorithms - Ch2 - Sorting

Analyzing MergeSort: Divide and Combine(?)

What is the cost of **Divide**? That depends:

- dividing an **array** involves finding the midpoint which can be done in time independent of the size of the array: assuming the size of the array is known and is small enough to fit in the hardware. $\Theta(1)$.
- or can be done in time $\Theta(n)$, if we use **linked lists**.

What is the cost of **Combine**?

- whether we use lists or arrays, we must compare the elements of the two subsets. At each comparison, we place one element. Total cost for n elements: $\Theta(n)$.

Algorithms - Ch2 - Sorting

Analyzing Merge: Divide and Conquer

The recursion relation becomes:

Termination:

$T(n) = \Theta(1) = c$ for $n = 1$;

Dividing and Combining:

$T(n) = 2\,T(n/2) + D(n) + C(n) = 2\,T(n/2) + \Theta(n)$ if $n > 1$.

Solution method: Figure 2.5 – p. 38.