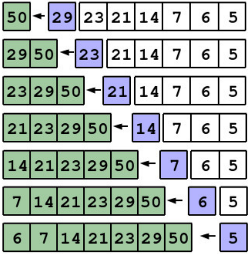
Sorting

Dr Timothy Kimber

January 2015



The Sorting Problem

- Sorting data is one of the most thoroughly explored algorithmic problems.

Problem *(Sort)*

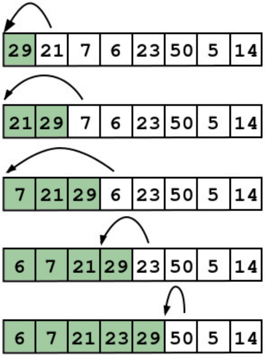
Input: a sequence A of values $\langle a_1, a_2, \ldots, a_N \rangle$

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

- There are many solutions.
- We shall be exploring several of the best known ones.
- Sorting is an important problem. It is part of the solution to many other problems.
- Understanding the complexity of sorting algorithms helps design good solutions to these other problems.

Insertion Sort

- The Insertion Sort algorithm divides A into a sorted part, initially just $\langle a_1 \rangle$, and the remaining unsorted part
- Elements from the unsorted part are then inserted into the sorted part



Insertion Sort

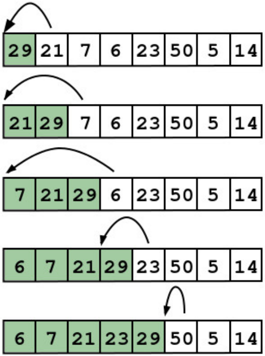
Insertion Sort(Input: sequence $A = \langle a_1, \ldots, a_N \rangle$)

- For each element a_i in $\langle a_2, \ldots, a_N \rangle$
    - $j = i$
    - While $j > 1$ and $a_j < a_{j-1}$
        - Exchange a_j and a_{j-1}
        - Decrement j
- HALT

- Insertion Sort is a so-called comparison sort algorithm
- To insert a_i in the correct position it must be compared with elements in the sorted part of the list
- Is Insertion Sort correct?

Insertion Sort

- The subsequence $\langle a_1, \ldots, a_{i-1} \rangle$ is sorted initially
- An insertion maintains this invariant
- Formalising this argument is the business of reasoning about programs
- Invariants are a tool we will use in designing correct algorithms



Insertion Sort

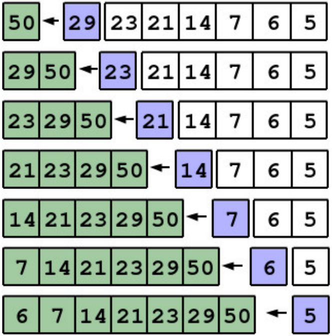
Insertion Sort(Input: sequence $A = \langle a_1, \ldots, a_N \rangle$)

- For each element a_i in $\langle a_2, \ldots, a_N \rangle$
  - $j = i$
  - While $j > 1$ and $a_j < a_{j-1}$
    - Exchange a_j and a_{j-1}
    - Decrement j
- HALT

- What is the worst case input?
- What is the best case input?
- What is the time complexity in the best and worst cases?

Worst Case

- Running time of Insertion Sort has two dimensions:
  - Number of insertions
  - 'Size' of insertion



Insertion Sort: Java

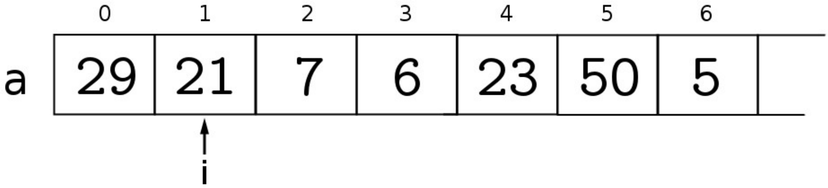
```java
public static void sort(int[] a) {
    int l = a.length;
    int j, t;

    for (int i = 1; i < l; i++) {
        t = a[i];
        j = i - 1;
        while(j >= 0 && t < a[j]) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = t;
    }
}
```

- The sorted part of the array is everything before `a[i]`
- Swaps are not fully implemented
- The element being inserted is saved into temp variable `t`
- Sorted elements greater than `t` are each shuffled to the right

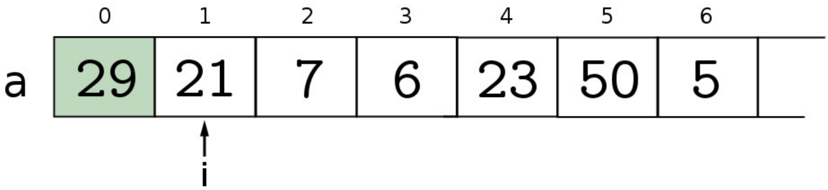
Insertion Sort

	0	1	2	3	4	5	6
a	29	21	7	6	23	50	5

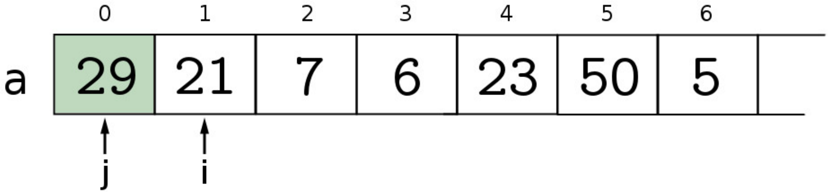
Insertion Sort



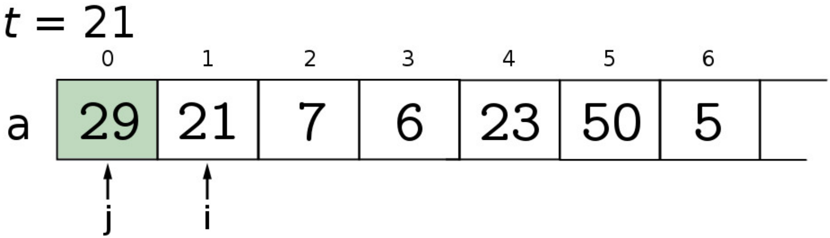
Insertion Sort



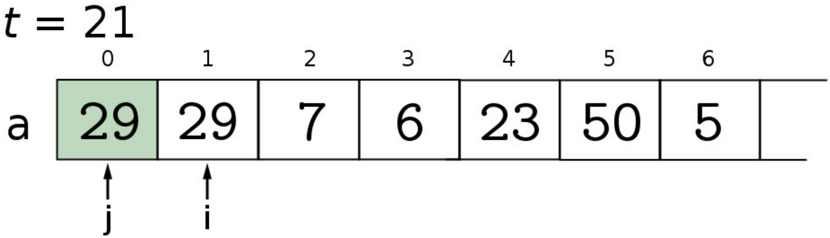
Insertion Sort



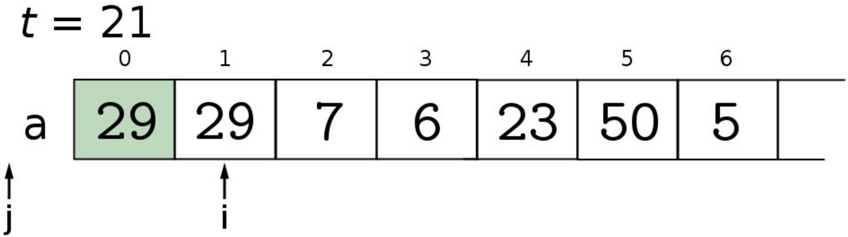
Insertion Sort



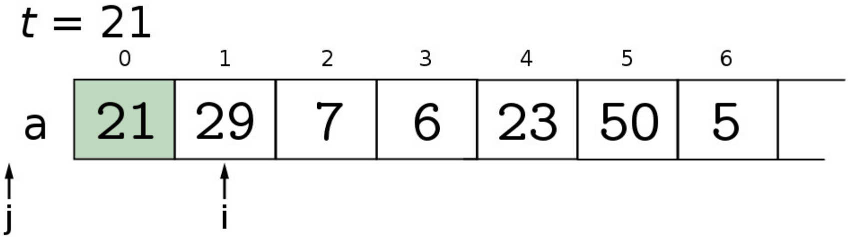
Insertion Sort



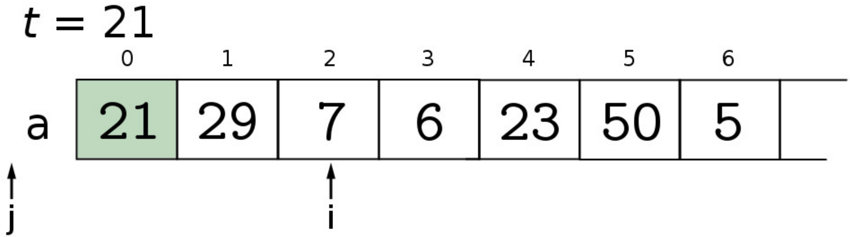
Insertion Sort



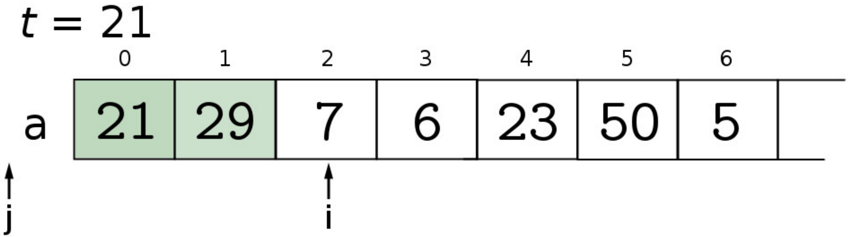
Insertion Sort



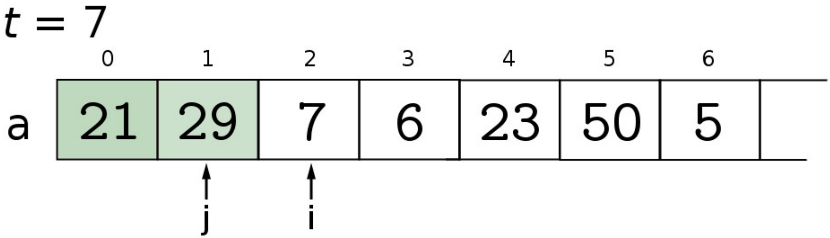
Insertion Sort



Insertion Sort

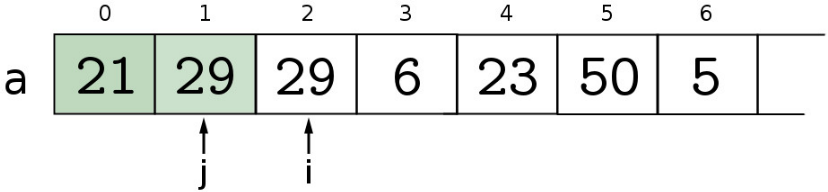


Insertion Sort

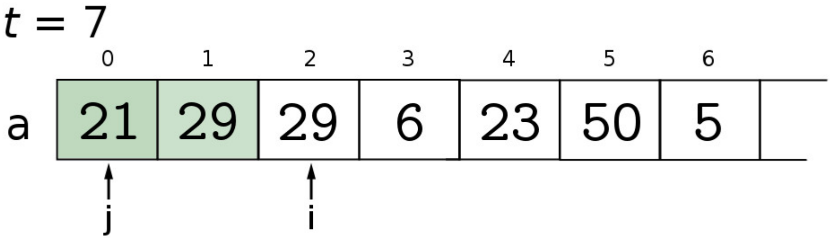


Insertion Sort

$t = 7$

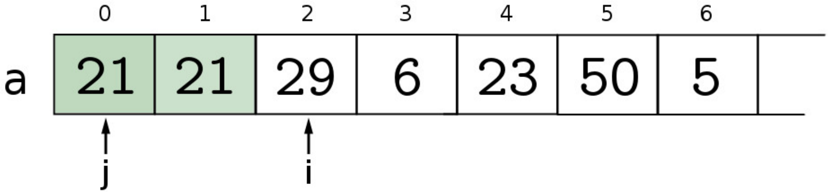


Insertion Sort



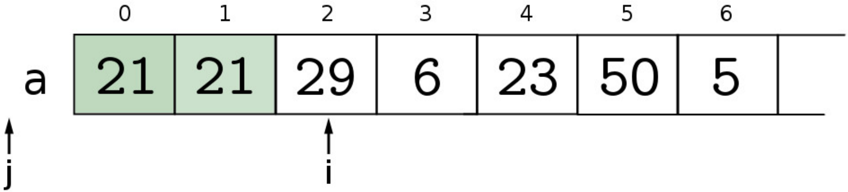
Insertion Sort

$t = 7$

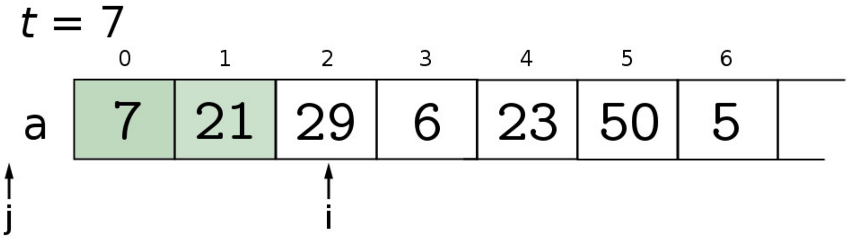


Insertion Sort

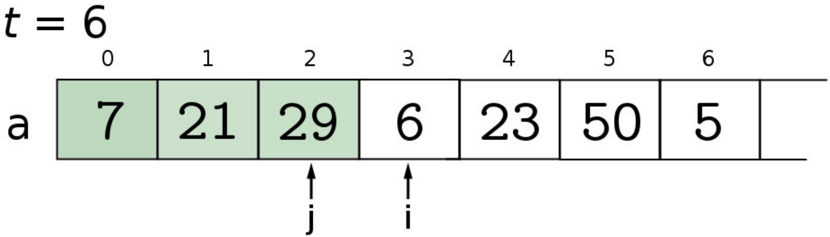
$t = 7$



Insertion Sort

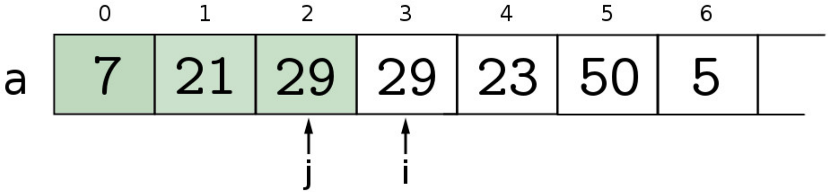


Insertion Sort

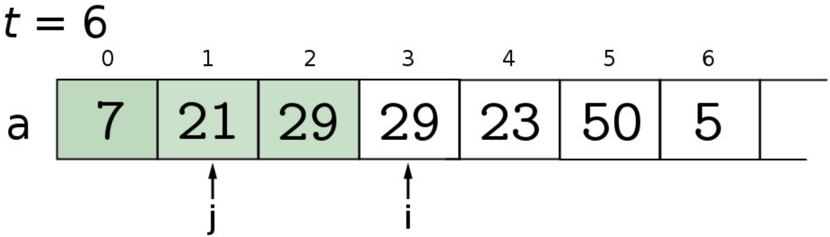


Insertion Sort

$t = 6$

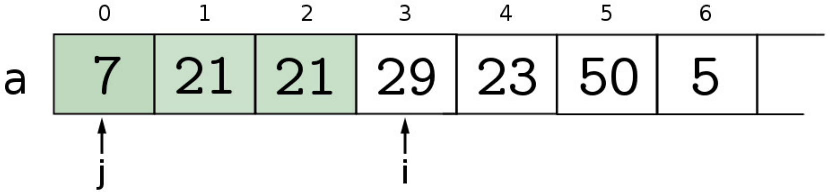


Insertion Sort



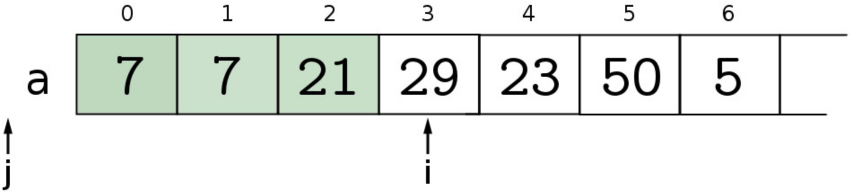
Insertion Sort

$t = 6$

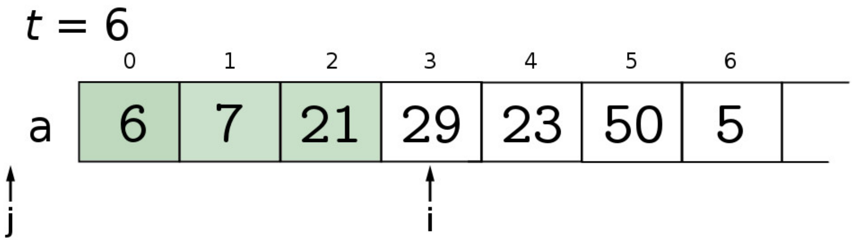


Insertion Sort

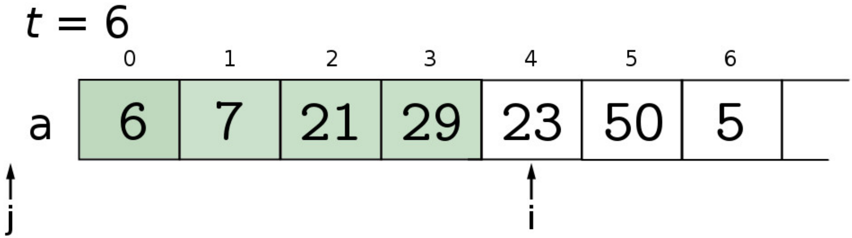
$t = 6$



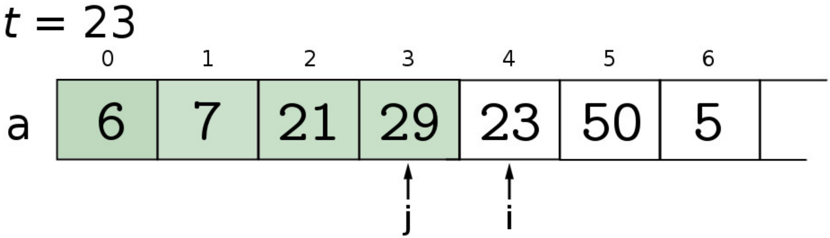
Insertion Sort



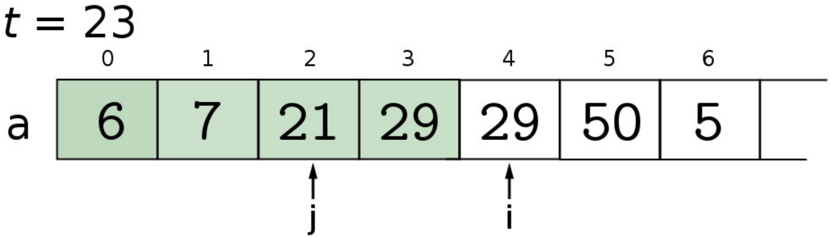
Insertion Sort



Insertion Sort

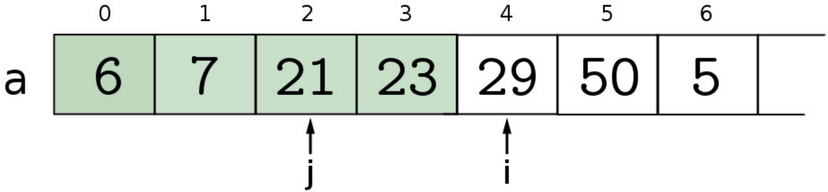


Insertion Sort



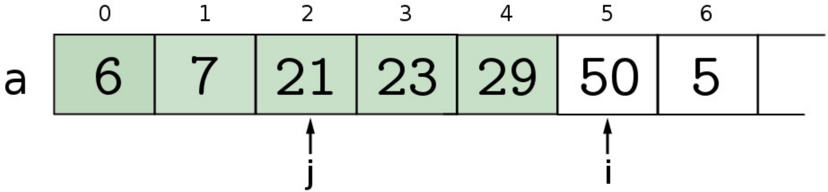
Insertion Sort

$t = 23$

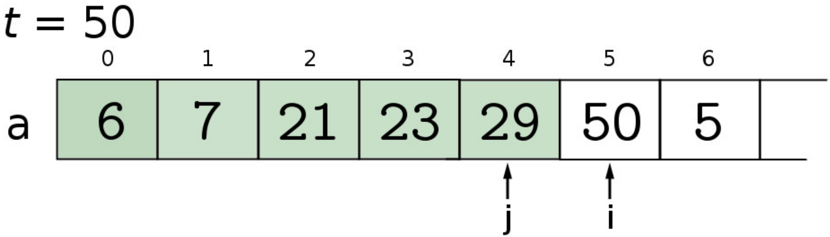


Insertion Sort

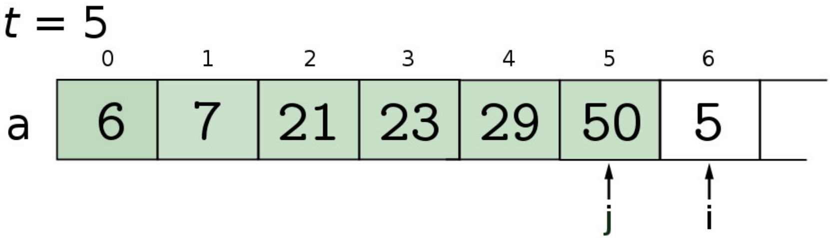
$t = 23$



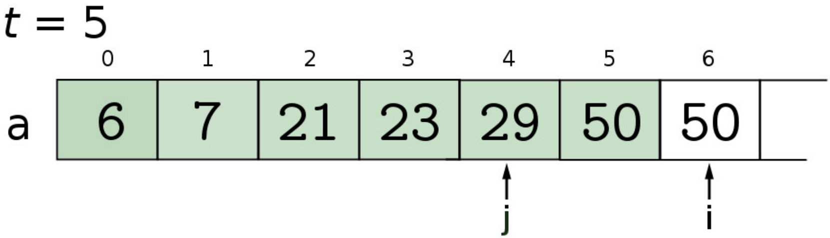
Insertion Sort



Insertion Sort

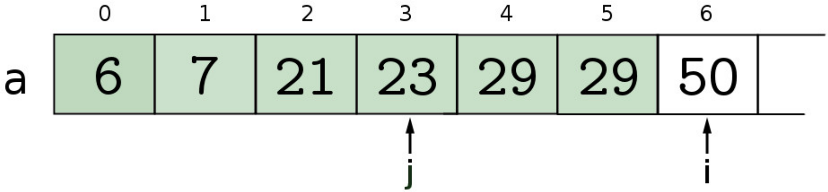


Insertion Sort

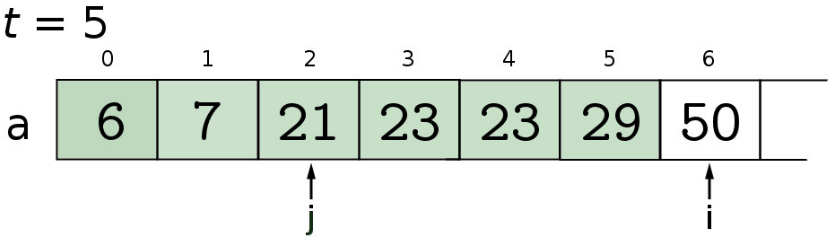


Insertion Sort

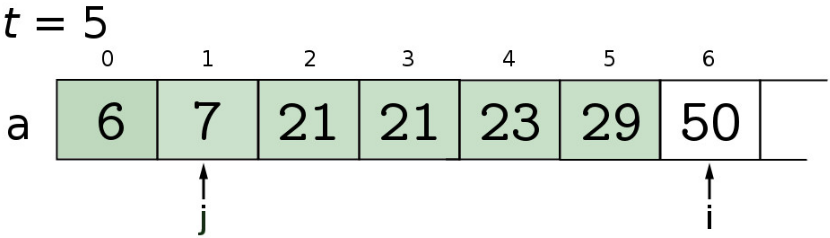
$t = 5$



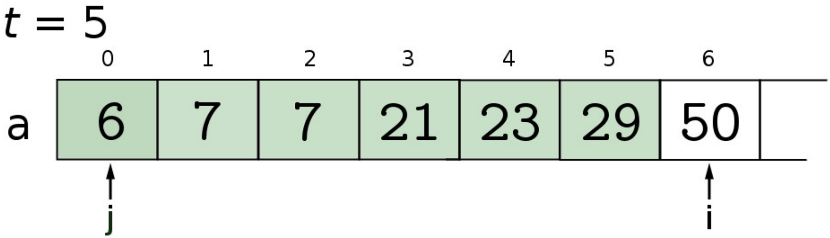
Insertion Sort



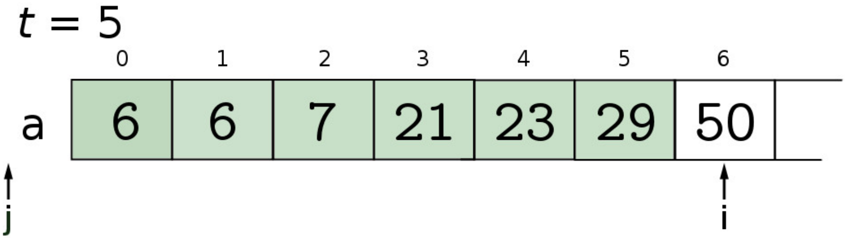
Insertion Sort



Insertion Sort

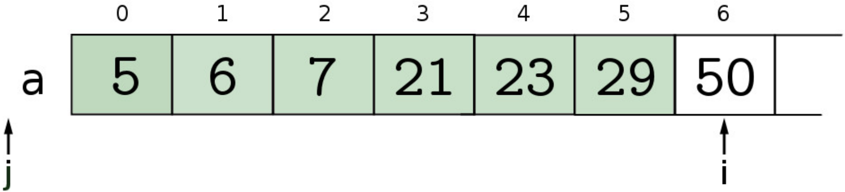


Insertion Sort

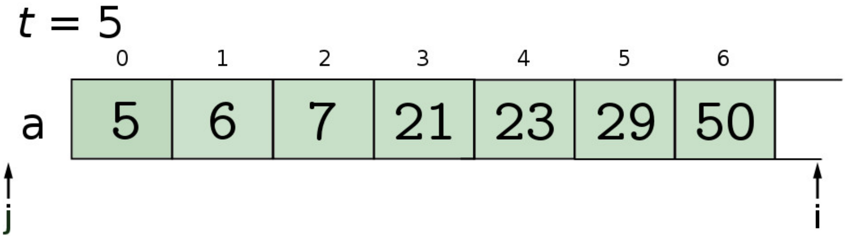


Insertion Sort

$t = 5$



Insertion Sort



Properties of Insertion Sort

- Sorts the array in place – do not need to create a new array to hold the answer
- Best case input is already sorted array ($\Theta(N)$)
- Worst case input is reverse sorted array ($\Theta(N^2)$)
- Time complexity in general is $O(N^2)$
- What about the 'average' case?

Merge Sort

Will a divide and conquer approach work?

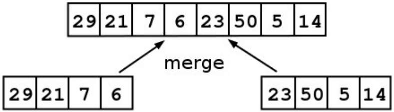
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.

29	21	7	6	23	50	5	14

Merge Sort

Will a divide and conquer approach work?

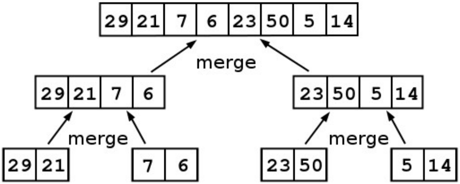
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

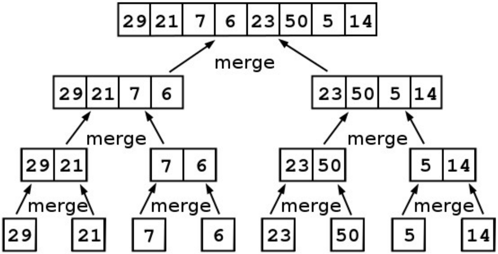
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

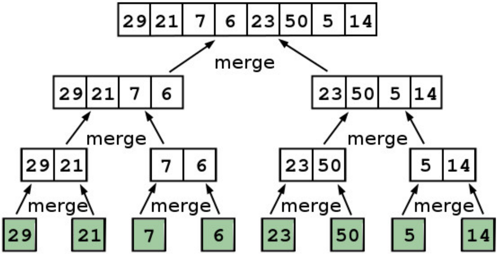
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

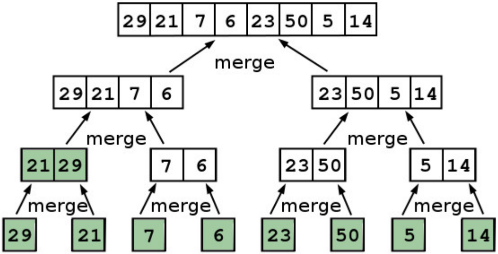
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

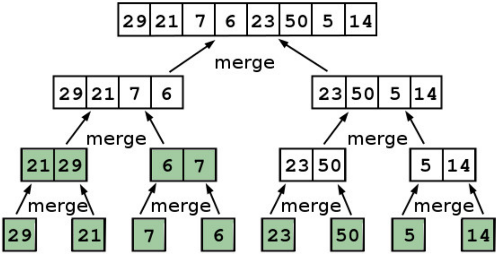
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

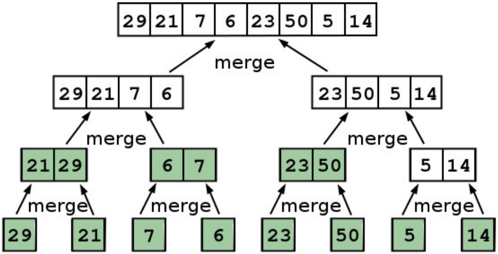
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

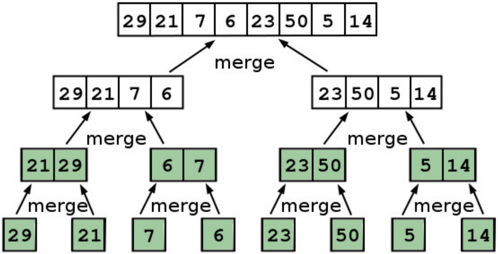
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

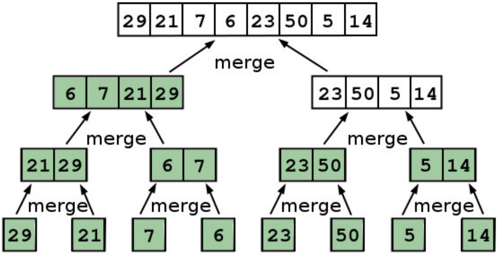
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

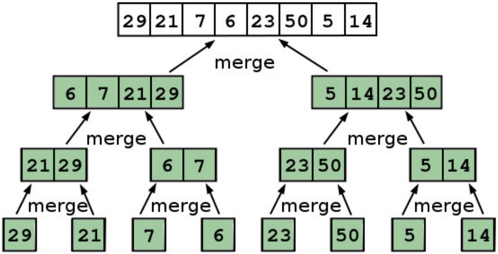
- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



Merge Sort

Will a divide and conquer approach work?

- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.

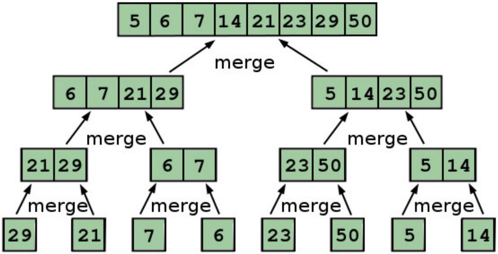


- What are the 'dimensions' of Merge Sort?

Merge Sort

Will a divide and conquer approach work?

- Combining (merging) already sorted lists is fast ($\Theta(N)$).
- The list $\langle a \rangle$ is already sorted.
- So, we can sort $A = \langle a_1, \ldots, a_N \rangle$ by repeated merging of sublists.



- What are the 'dimensions' of Merge Sort?

Merge Sort

Merge Sort (Input: sequence $A = \langle a_1, \ldots, a_N \rangle$, where $N \geq 1$)

- If A has one element
  - HALT
- Otherwise
  - Merge Sort $\langle a_1, \ldots, a_{\lfloor N/2 \rfloor} \rangle$
  - Merge Sort $\langle a_{\lfloor N/2 \rfloor + 1}, \ldots, a_N \rangle$
  - Merge $\langle a'_1, \ldots, a'_{\lfloor N/2 \rfloor} \rangle$ and $\langle a'_{\lfloor N/2 \rfloor + 1}, \ldots, a'_N \rangle$
  - HALT

- The sorting appears to be happening in place, but the list is copied during `Merge`

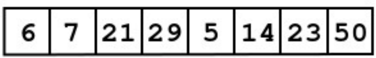
Merge Sort

- The Merge procedure takes two sublists of A and combines them
- The sublists $\langle a_L, \ldots, a_M \rangle$ and $\langle a_{M+1}, \ldots, a_N \rangle$ must be sorted
- The result is a globally sorted list $\langle a'_L, \ldots, a'_N \rangle$

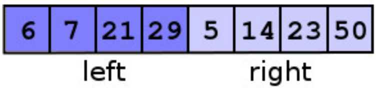
Merge (Input: sequence A, indices L, M and N, where $N > M \geq L$)

- Copy $\langle a_L, \ldots, a_M \rangle$ to a new list P and $\langle a_{M+1}, \ldots, a_N \rangle$ to a new list Q
- Copy the elements of P and Q back to form $\langle a'_L, \ldots, a'_N \rangle$ as follows:
    - p is the smallest element of P not yet used
    - q is the smallest element of Q not yet used
    - the next element of $\langle a'_L, \ldots, a'_N \rangle$ is the smaller of p and q

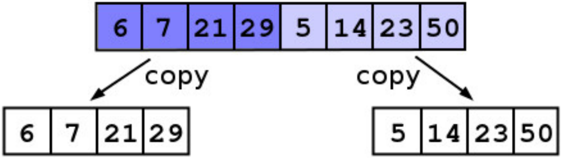
Merging Sublists



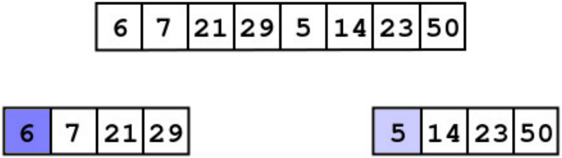
Merging Sublists



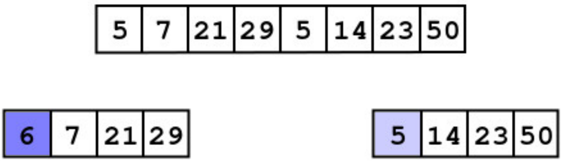
Merging Sublists



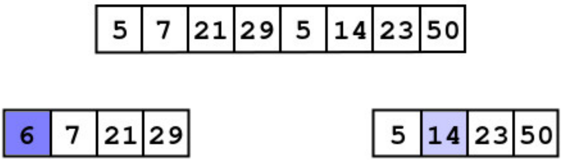
Merging Sublists



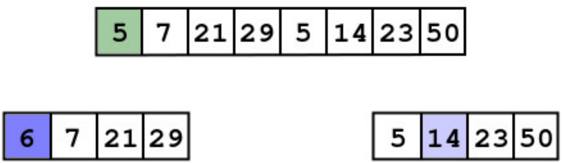
Merging Sublists



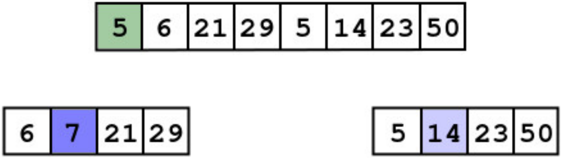
Merging Sublists



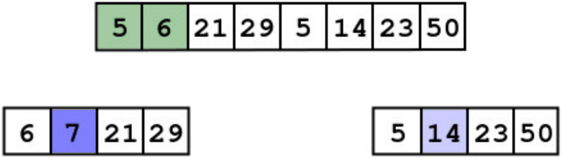
Merging Sublists



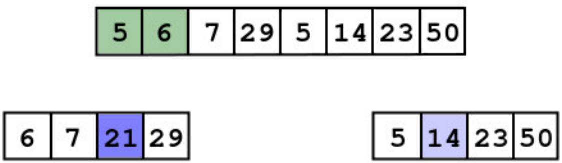
Merging Sublists



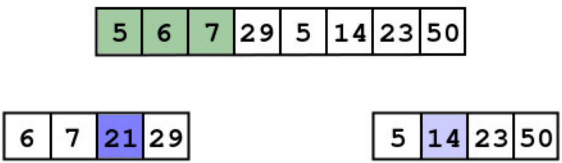
Merging Sublists



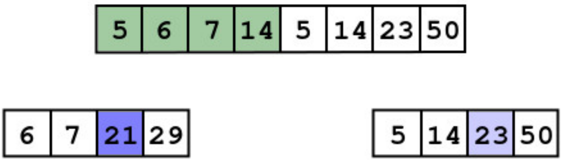
Merging Sublists



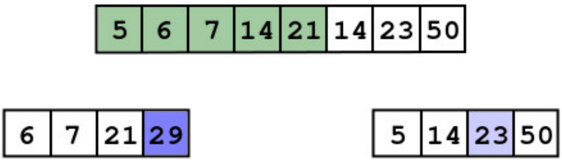
Merging Sublists



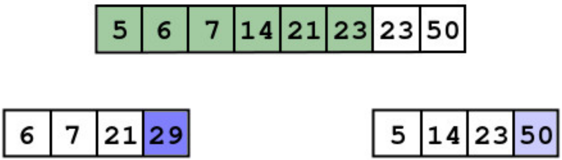
Merging Sublists



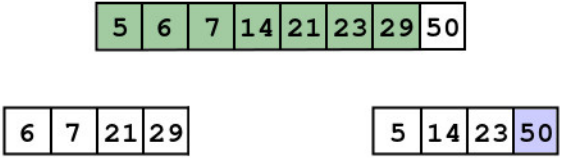
Merging Sublists



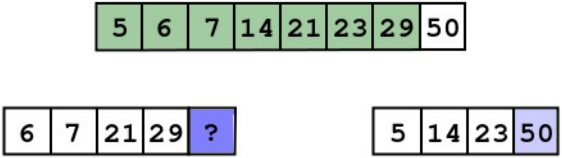
Merging Sublists



Merging Sublists

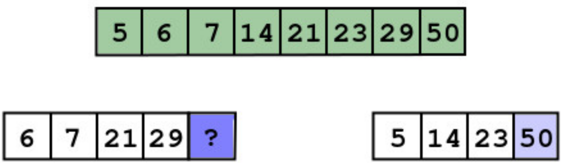


Merging Sublists



- We can add a sentinel value to the copied lists
- This avoids the need for extra conditions at the end of the lists
- What should the sentinel be?

Merging Sublists



- We can add a sentinel value to the copied lists
- This avoids the need for extra conditions at the end of the lists
- What should the sentinel be?

Merge Sort: Java

```
public static void sort(int[] a) {
  int l = a.length;
  sort(a, 0, l);
}

private static void sort(int[] a, int from, int to) {

  if (to - from <= 1) return;

  int mid = (from + to) / 2;
  sort(a, from, mid);
  sort(a, mid, to);
  merge(a, from, mid, to);
}
```

- The recursive method is called from a public ‘interface’ method
- The recursive method can sort any sub-array
- A sub-array including `a[i]...a[j]` is represented by `(a,i,j + 1)`

Merge Sort: Java

```java
private static void merge(int[] a, int from, int mid, int to) {

    int[]  left = copyOf(a, from, mid);
    int[] right = copyOf(a, mid, to);

    int i = 0, j = 0, k = from;
    while (k < to) {
        if (left[i] < right[j]) {
            a[k++] = left[i++];
        } else {
            a[k++] = right[j++];
        }
    }
}

private static int[] copyOf(int[] a, int from, int to) {
    int    len = to - from;              // length of range copied
    int[] copy = new int[len + 1];       // new array with sentinel
    int      i = 0;                      // index within copy

    for (int j = from; j < to; j++) {
        copy[i++] = a[j];
    }
    copy[len] = Integer.MAX_VALUE;
    return copy;
}
```

Merge Sort: Analysis

Generalised divide and conquer scheme:

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

where

- c is a small value of N (e.g. 0 or 1) corresponding to a base case that runs in constant time
- a is the number of subproblems
- N/b is the 'size' of each subproblem
- $D(N)$ is the cost of dividing up the original problem
- $C(N)$ is the cost of combining the subproblem solutions

Merge Sort: Analysis

Applying this scheme to Merge Sort:

- Use simplifying assumption that N is a power of 2
- (For $N > 1$) we get two subproblems of size $N/2$, so:

$$T(N) = \begin{cases} \Theta(1) & \text{, if } N \leq 1 \\ 2T(N/2) + D(N) + C(N) & \text{, otherwise} \end{cases}$$

- What are $D(N)$ and $C(N)$?

Merge Sort: Analysis

Applying this scheme to Merge Sort:

- Use simplifying assumption that N is a power of 2
- (For $N > 1$) we get two subproblems of size $N/2$, so:

$$T(N) = \begin{cases} \Theta(1) & \text{, if } N \leq 1 \\ 2T(N/2) + D(N) + C(N) & \text{, otherwise} \end{cases}$$

- What are $D(N)$ and $C(N)$?

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

Evaluating Equations with Asymptotic Notation

In

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

- $\Theta(1)$ is some function $f(N) = \Theta(1)$
- $\Theta(N)$ is some function $g(N) = \Theta(N)$
- $f(N) + g(N)$ will also be a linear function of N, for all f, g, so

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

- Likewise, if $T(N/2)$ is
  - $\Theta(N^2)$ then $T(N)$ is $\Theta(N^2)$
  - $\Theta(1)$ then $T(N)$ is $\Theta(N)$

Merge Sort: Analysis

Let $T(1) = d$ and cost of merge per element be c

$$T(N) = \begin{cases} d & \text{, if } N \leq 1 \\ 2T(N/2) + cN & \text{, otherwise} \end{cases}$$

Substituting in the same way we saw for binary search:

$$\begin{aligned} T(N) &= cN + 2T(N/2) && (1) \\ &= cN + 2(cN/2 + 2T(N/4)) && (2) \\ &= cN + cN + 4T(N/4) && (3) \\ &= cN + cN + cN + 8T(N/8) && (4) \\ &= cN \times \log_2(N) + dN && (5) \end{aligned}$$

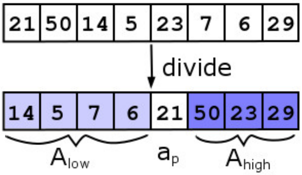
So, $T(N) = \Theta(N \log_2 N)$

Properties of Merge Sort

- Uses (extra) memory proportional to N
- Time complexity is $\Theta(N \log_2 N)$
- Faster than Insertion Sort for large, unsorted lists
- Slower than Insertion Sort if the list is already sorted
- Slower than Insertion Sort for small N

Quick Sort

- Merge Sort works bottom up – all the action happens when recombining the solutions to subproblems
- Quicksort is a top down divide and conquer sorting algorithm
- The input list is divided into three parts: A_{low}, $\langle a_p \rangle$, A_{high}
- a_p is called the pivot and the division ensures that $\forall a \in A_{low}(a < a_p)$ and $\forall a \in A_{high}(a \geq a_p)$



- We are left with the subproblems of sorting A_{low} and A_{high}

Quicksort

Quicksort (Input: sequence $A = \langle a_1, \ldots, a_N \rangle$)

- If A has more than one element
    - $p =$ Partition A
    - Quicksort $\langle a_1, \ldots, a_{p-1} \rangle$
    - Quicksort $\langle a_{p+1}, \ldots, a_N \rangle$
    - HALT
- Otherwise
    - HALT

- The Quicksort divide step is called partitioning
- The Partition procedure returns the final index of the pivot
- The base case must include $\langle\rangle$ since p might be 1 or N
- Subproblem solutions do not need to be combined

Quicksort

- The Partition procedure maintains two sublists which grow
- The sublists can both grow left-to-right (as below), or one left-to-right and one right-to-left

Partition (Given sequence $A = \langle a_1, \ldots, a_N \rangle$)

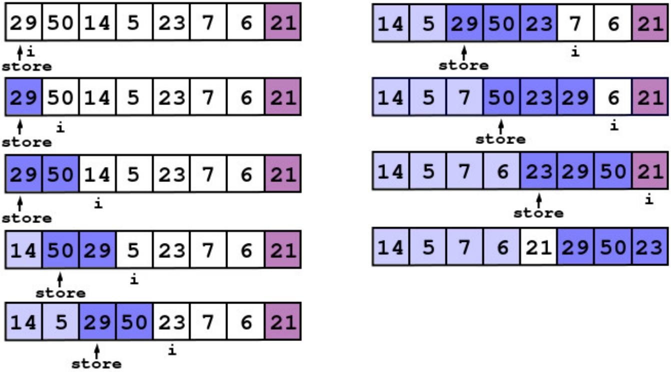
- Choose a pivot element $p \in A$
- Exchange p with a_N
- Let $storeIndex = 1$
- For each a in $\langle a_1, \ldots, a_{N-1} \rangle$
  - If $a < p$: exchange a with $a_{storeIndex}$ and increment $storeIndex$
- Exchange a_N with $a_{storeIndex}$
- Return $storeIndex$

Quicksort in Java

```java
private static void sort(int[] a, int from, int to) {
    if (to - from > 1) {
        int pivot = partition(a, from, to);
        sort(a, from, pivot);
        sort(a, pivot + 1, to);
    }
}

public static int partition(int[] a, int from, int to) {
    int pivot = to - 1;
    int store = from;
    for (int i = from; i < pivot; i++) {
        if (a[i] < a[pivot]) { swap(a, i, store++); }
    }
    swap(a, pivot, store);
    return store;
}
```

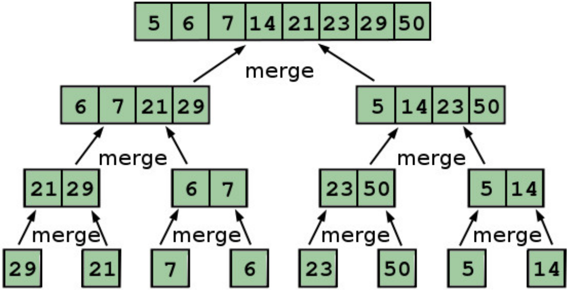
- Sub-array representation as for Mergesort
- Within `partition`, the final element (`a[to - 1]`) is the pivot
- Elements before `a[store]` are less than the pivot
- Elements between `a[store]` and `a[i]` are not less than the pivot

The partition method



Quicksort Performance

- The ‘dimensions’ of Quicksort are much like those of Merge Sort
- With balanced partitioning there will be a similar tree of subproblems

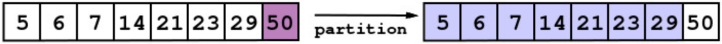


- How will partition behave? What cases are there?

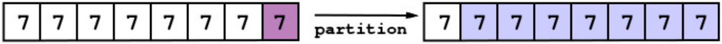
Quicksort Performance

Using our Java `partition` method:

- will only remove one element from sorted or reverse-sorted data



- will only remove one element from data with many duplicates



These types of input will produce $O(N^2)$ performance

Quicksort Performance

With non-worst-case inputs Quicksort usually outperforms Merge Sort

- The constant factors are smaller
- Quicksort common choice for library sort functions

Strategies for avoiding $O(N^2)$ performance include:

- Choose the pivot at random
- Choose the pivot as the median of three random elements [Sedgewick]
  - Both give expected $\Theta(N \log_2 N)$ performance
  - The worst case is still possible, but unlikely
- Modify `partition` to generate three partitions containing elements that are strictly less than, equal to and greater than the pivot