# Object-Oriented Programming and Data Structures

COMP2012: Sorting

# Sorting Algorithms

- Problem: Given a list of N items, sort them in non-descending order.

- Comparison-based, exchanging adjacent elements
  - Bubble sort
  - Selection sort
  - Insertion sort

- Comparison-based, divide-and-conquer
  - Merge sort
  - Quicksort

- Non-comparison-based, linear time
  - Radix sort

# Some Terminologies

- Stable algorithms:
  - do not change the relative order of elements with equal values.
  - Importance: e.g., sort names first on the first name and then on the last name.
    - if the last names are sorted by stable algorithm, names with the same last name will maintain the relative sorted order of their first name.
- In-place algorithms:
  - no extra space is required, or require only a constant amount O(1) of extra memory space.
- Online algorithms:
  - sort a list as it receives it.

# Comparison-based Sorting Algorithms by Exchanging Adjacent Elements

# Bubble Sort

- Procedure
  1. Starting from the first item, compare each pair of adjacent items.
  2. If they are out of order, swap them.
  3. Repeat steps 1 – 2 with only the first (*N-j*) items until no swapping occurs in an iteration, where *j* is the iteration index (starting from 1).
- Idea: At the end of each iteration, the maximum item should "bubble" to the end.
- Worst-case: $(N-1) + (N-2) + \ldots + 1 = N(N+1)/2 \Rightarrow O(N^2)$
- Average-case: $O(N^2)$
- Best-case: O(N) when the input is already sorted.

# Bubble Sort: Example

- To sort: 2, 9, 7, 4, 1
  - first pass:
    - 2, 9, 7, 4, 1
    - 2, 7, 9, 4, 1
    - 2, 7, 4, 9, 1
    - 2, 7, 4, 1, 9
  - second pass:
    - 2, 7, 4, 1, 9
    - 2, 4, 7, 1, 9
    - 2, 4, 1, 7, 9
  - etc.

# Bubble Sort Code

```
#include <algorithm>

void bubble_sort(int* data, int N)
{
    bool swapped = true;

    while (swapped) // If swapping occurs in the previous iteration
    {
        swapped = false;

        for (int j = 1; j < N; ++j)
        {
            if (data[j] < data[j-1]) // Max item bubbles to the end
            {
                std::swap(data[j], data[j-1]);
                swapped = true;      // Swapping occurs
            }
        }

        --N;
    }
}
```

# Selection Sort

- Procedure
  1. Find the minimum item in the list
  2. Swap it with the item in the first position
  3. Repeat the steps 1 and 2 for the rest of the list (starting at the second position)

- Worst/Average/Best-case analysis:
  - finding the minimum item is O(N)
  - the procedure is repeated N times => $O(N^2)$

# Selection Sort: Example

- To sort: 2, 9, 7, 4, 1
  - 1st iteration: min_index = 4
    - 1, 9, 7, 4, 2
  - 2nd iteration: min_index = 3 for the sub-list
    - 1, 2, 7, 4, 9
  - 3rd iteration: min_index = 1 for the sub-list
    - 1, 2, 4, 7, 9
  - 4th iteration: min_index = 0 for the sub-list
    - 1, 2, 4, 7, 9
  - 5th iteration: return immediately
    - 1, 2, 4, 7, 9

# Selection Sort Code

```
#include <algorithm>

void selection_sort(int* data, int N)
{
    if (N <= 1) // Base case
        return;

    // Find the index of the minimum value
    int min_index = 0;

    for (int j = 1; j < N; ++j)
    {
        if (data[min_index] > data[j])
            min_index = j;
    }

    if (min_index != 0)
        std::swap(data[0], data[min_index]); // Swap min with the first
    selection_sort(data+1, N-1); // Sort the remaining list
}
```

# Insertion Sort

- Procedure
  1. It takes (N – 1) passes.
  2. For pass *p* = 1 to (N – 1), items in position 0 to (*p-1*) are sorted.
  3. In pass *p*, continuously move the item in position *p* left until it finds its correct position in the preceding *p* items.

# Insertion Sort: Example

| Original | 34 | 8 | 64 | 51 | 32 | 21 | Positions Moved |
|---|---|---|---|---|---|---|---|
| After $p = 1$ | 8 | 34 | 64 | 51 | 32 | 21 | 1 |
| After $p = 2$ | 8 | 34 | 64 | 51 | 32 | 21 | 0 |
| After $p = 3$ | 8 | 34 | 51 | 64 | 32 | 21 | 1 |
| After $p = 4$ | 8 | 32 | 34 | 51 | 64 | 21 | 3 |
| After $p = 5$ | 8 | 21 | 32 | 34 | 51 | 64 | 4 |

# Insertion Sort Code

```
void insertion_sort(int* data, int N)
{
    for (int p = 1; p < N; ++p)
    {
        int item_to_insert = data[p];

        int j;
        for (j = p; j > 0 && item_to_insert < data[j-1]; --j)
            data[j] = data[j-1];

        data[j] = item_to_insert;
    }
}
```

# Insertion Sort: Extended Example

- 6 items to sort: 34 8 64 51 32 21

p = 1; item_to_insert = 8;

34 > item_to_insert, so 2nd element a[1] is set to 34: {8, 34}…

We have reached the front of the list.
Thus, 1st position a[0] = item_to_insert=8

After 1st pass: 8 34 64 51 32 21

(first 2 elements are sorted)

p = 2; item_to_insert = 64;

34 < 64, so stop at 3rd position and set 3rd position = 64

After 2nd pass: 8 34 64 51 32 21

(first 3 elements are sorted)

# Insertion Sort: Extended Example ..

p = 3;  item_to_insert = 51;

51 < 64, so we have  8  34  64  64  32  21,

34 < 51, so stop at 2nd position, set 3rd position = item_to_insert,

After 3rd pass: 8  34  51  64  32  21

(first 4 elements are sorted)

p = 4; item_to_insert = 32;

32 < 64, so 8  34  51  64  64  21,

32 < 51, so 8  34  51  51  64  21,

next 32 < 34,  so 8  34  34  51  64  21,

next 32 > 8, so stop at 1st position and set 2nd position = 32,

After 4th pass: 8  32  34  51  64  21

p = 5; item_to_insert = 21,  . . .

After 5th pass:  8  21  32  34  51  64

# Insertion Sort: Worst-case Analysis

```
for (j = p; j > 0 && item_to_insert < data[j-1]; --j)
    data[j] = data[j-1];
```

- Inner loop is executed p times, for each *p* = 1..N

  ⟹ Overall: 1 + 2 + 3 + . . . + (N-1) = $O(N^2)$

- Space (memory) requirement is O(N)

# Insertion Sort: The Bound is Tight

- Tight bound: $\Theta(N^2)$
- That is, there exists some input which actually uses (the lower bound) $\Omega(N^2)$ time.
- Consider the worst case with a reversed sorted list
  - When *A[p]* is inserted into the sorted *A[0..p-1]*, we need to compare *A[p]* with all elements in A*[0 .. p-1]* and move each element one position to the right

    $\Rightarrow$ $\Omega(p)$ steps
  - the total number of steps is

    $\Omega(\Sigma_1^{N-1} p) = \Omega(N(N-1)/2) = \Omega(N^2)$

# Insertion Sort: Best-case Analysis

- The input is already sorted in increasing order
  - When inserting *a[p]* into the sorted *a[0..p-1]*, only need to compare *a[p]* with *a[p-1]* and there is no data movement
  - For each iteration of the outer for-loop, the inner for-loop terminates after checking the loop condition once
    => O(N) time
- If input is nearly sorted, insertion sort runs fast.

# Insertion Sort: Summary

- Simple to implement.
- Efficient on (quite) small data sets.
- Efficient on data sets which are already substantially sorted.
- More efficient in practice than most other simple $O(N^2)$ algorithms such as selection sort or bubble sort.
  - it is linear in the best case.
- Properties:
  - stable
  - in-place
  - online algorithm

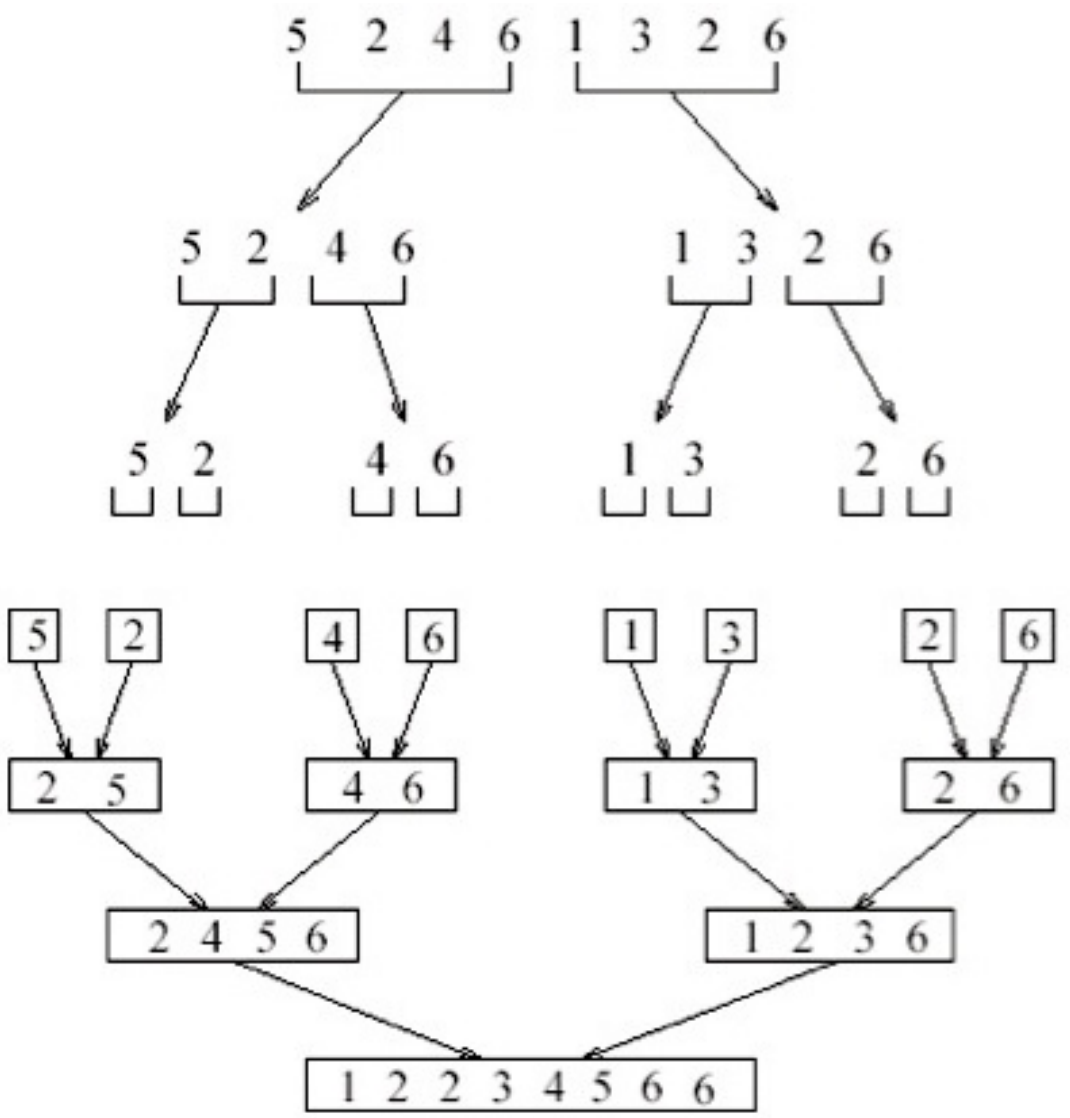
# Comparison-based Sorting Algorithms by Divide-and-Conquer

# Mergesort

- Procedure:
  1. Split data into roughly 2 equal halves.
  2. Recursively mergesort each half.
  3. Merge the two sorted halves together.

- 2 issues
  - #1: How to divide?
  - #2: How to merge?

# Mergesort: Example



# Mergesort: How to Divide?

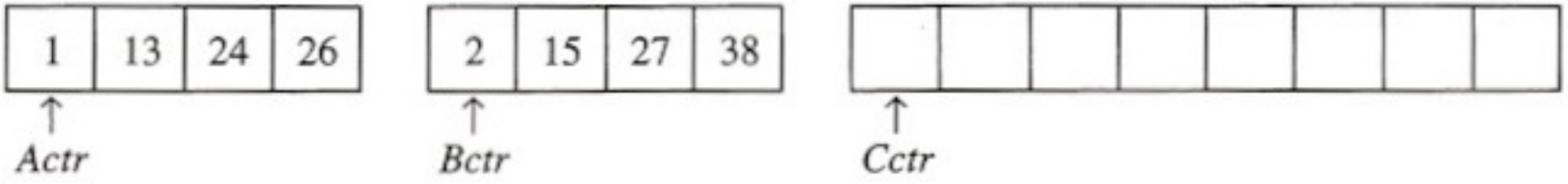
- Divide into 2 halves as equal as possible.

```
// Actual recursive routine requires an temporary array to store results
void mergesort(int* data, int* temp, int begin, int end)
{
    if (begin < end)
    {
        int center = (begin + end)/2;
        mergesort(data, temp, begin, center);
        mergesort(data, temp, center+1, end);
        merge(data, temp, begin, center, end);
    }
}

void mergesort(int* data, int begin, int end) // Driver function
{
    int* temp = new int [end - begin + 1]; // Store merged results
    mergesort(data, temp, begin, end);     // Actual recursive routine
    delete temp;
}
```

# Mergesort: How to Merge?

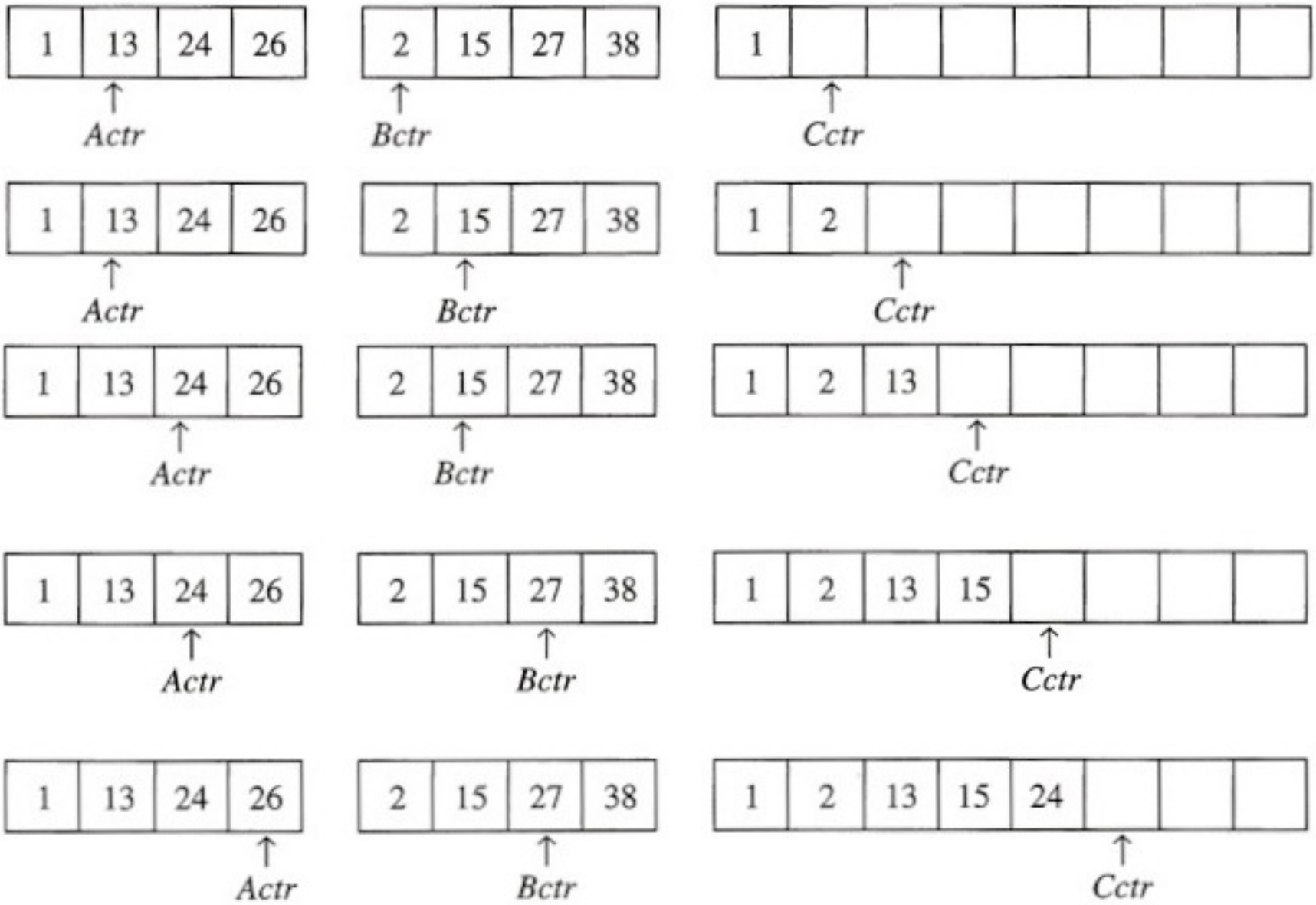
- Input: 2 sorted arrays, A and B
- Output: an output sorted array C
- Procedure:

1. 3 counters: Actr, Bctr, and Cctr, initially set to the beginning of their respective arrays.

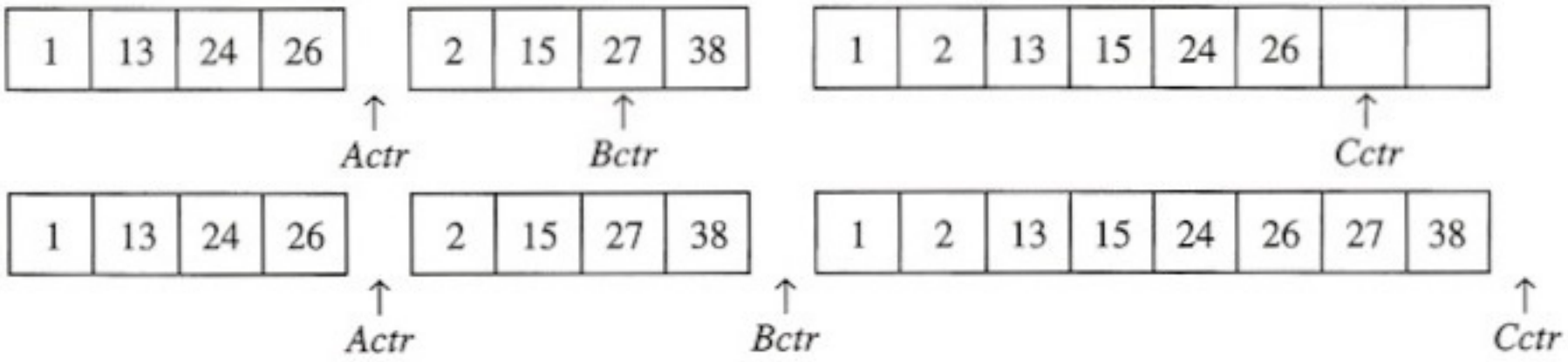



2. Compare A[Actr] and B[Bctr]: the smaller one is copied to the C[Cctr].
3. Advance Cctr and Actr/Bctr if its item is copied; repeat 2.
4. When either input list is exhausted, the remainder of the other list is copied to C.

# Mergesort: How to Merge? ..



# Mergesort: How to Merge? …




- Running time analysis of merging 2 lists of lengths N1 and N2 :
  - $O(N_1 + N_2) = O(N)$ as $N = N_1 + N_2$.
- Space requirement:
  - need the space of an additional list

# Merge Code

```
void merge(int* data, int* temp, int l_begin, int l_end, int r_end)
{
    int left = l_begin;     // Indexing the left subset
    int right = l_end + 1;  // Indexing the right subset
    int j = l_begin;        // Indexing the temp array for intermediate results

    // Merge the 2 sorted sublists
    while (left <= l_end && right <= r_end)
        temp[j++] = (data[left] <= data[right]) ? data[left++] : data[right++];

    // Copy any remaining items in the left sublist
    while (left <= l_end)
        temp[j++] = data[left++];

    // Copy any remaining items in the right sublist
    while (right <= r_end)
        temp[j++] = data[right++];

    // Copy results from the temp array back to the data array
    for (int k = l_begin; k <= r_end; ++k)
        data[k] = temp[k];
}
```

# Mergesort: Analysis

- Let T(N) be the worst-case running time of mergesort.
- Assume there are $N = 2^k$ numbers.
- Divide step: O(1) to find the center; result in 2 ~equal halves.
- Conquer step: 2 T(N/2)
- Merge step: O(N)
- Recurrence equation:
  - T(1) = 1
  - T(N) = 2T(N/2) + N

# Mergesort: Analysis ..

$$T(N) = 2T(\frac{N}{2}) + N$$

$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$

$$= 4T(\frac{N}{4}) + 2N$$

$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$

$$= 8T(\frac{N}{8}) + 3N = \cdots$$

$$= 2^k T(\frac{N}{2^k}) + kN$$

Since $N = 2^k$, we have $k = \log_2 N$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$

$$= N + N \log N$$

$$= O(N \log N)$$

# Mergesort vs. Insertion Sort: An Experiment

- Code from textbook (using template)
- Unix `time` utility

| $n$ | Isort (secs) | Msort (secs) | Ratio |
|---|---|---|---|
| 100 | 0.01 | 0.01 | 1 |
| 1000 | 0.18 | 0.01 | 18 |
| 2000 | 0.76 | 0.04 | 19 |
| 3000 | 1.67 | 0.05 | 33.4 |
| 4000 | 2.90 | 0.07 | 41 |
| 5000 | 4.66 | 0.09 | 52 |
| 6000 | 6.75 | 0.10 | 67.5 |
| 7000 | 9.39 | 0.14 | 67 |
| 8000 | 11.93 | 0.14 | 85 |

# Mergesort vs. Insertion Sort: An Experiment ..

## Comparing $n \log_{10} n$ and $n^2$

| $n$ | $n \log_{10} n$ | $n^2$ | Ratio |
|---|---|---|---|
| 100 | 0.2K | 10K | 50 |
| 1000 | 3K | 1M | 333.33 |
| 2000 | 6.6K | 4M | 606 |
| 3000 | 10.4K | 9M | 863 |
| 4000 | 14.4K | 16M | 1110 |
| 5000 | 18.5K | 25M | 1352 |
| 6000 | 22.7K | 36M | 1588 |
| 7000 | 26.9K | 49M | 1820 |
| 8000 | 31.2K | 64M | 2050 |

# Quicksort: Generally the Best

- Fastest known sorting algorithm in practice.
- Best case:
  - O(N) [ 3-way partitioning (smaller, equal, larger subsets) + all values being equal ]
  - O(N log N) [bad partition]
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.

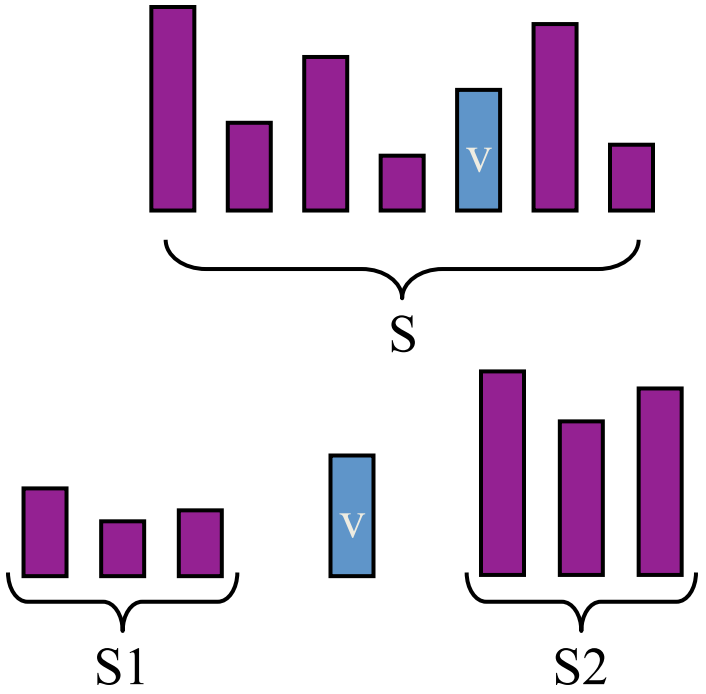
# Quicksort: Idea

- Divide step:
  - Pick any element (***pivot***) v in S
  - Partition S – {v} into 2 disjoint groups:

    S1 = {x ∈ S – {v} | x <= v}

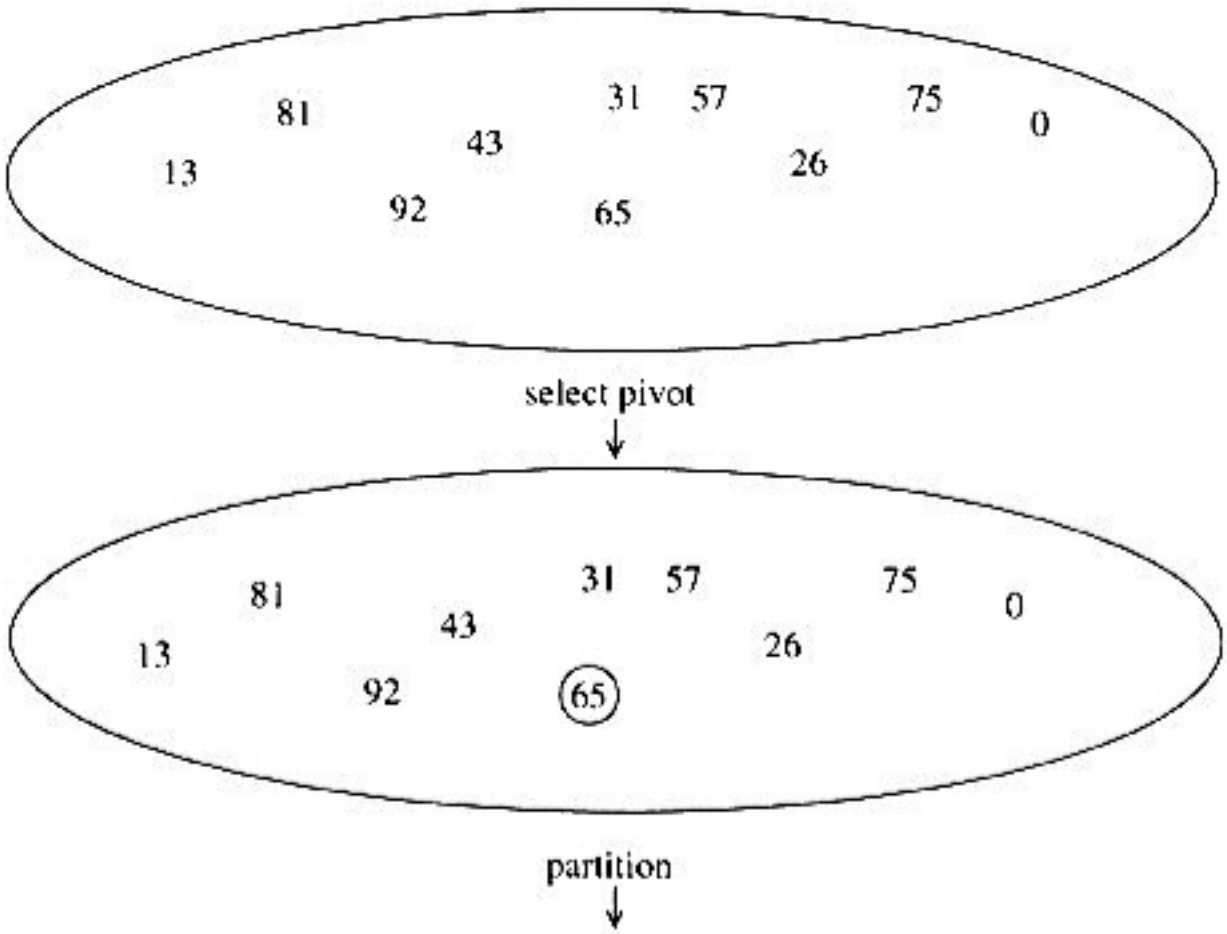
    S2 = {x ∈ S – {v} | x ≥ v}
- Conquer step:
  - recursively sort S1 and S2
- Combine step:
  - just join together the sorted S1, followed by v, followed by the sorted S2

    (nothing extra needs to be done!!)

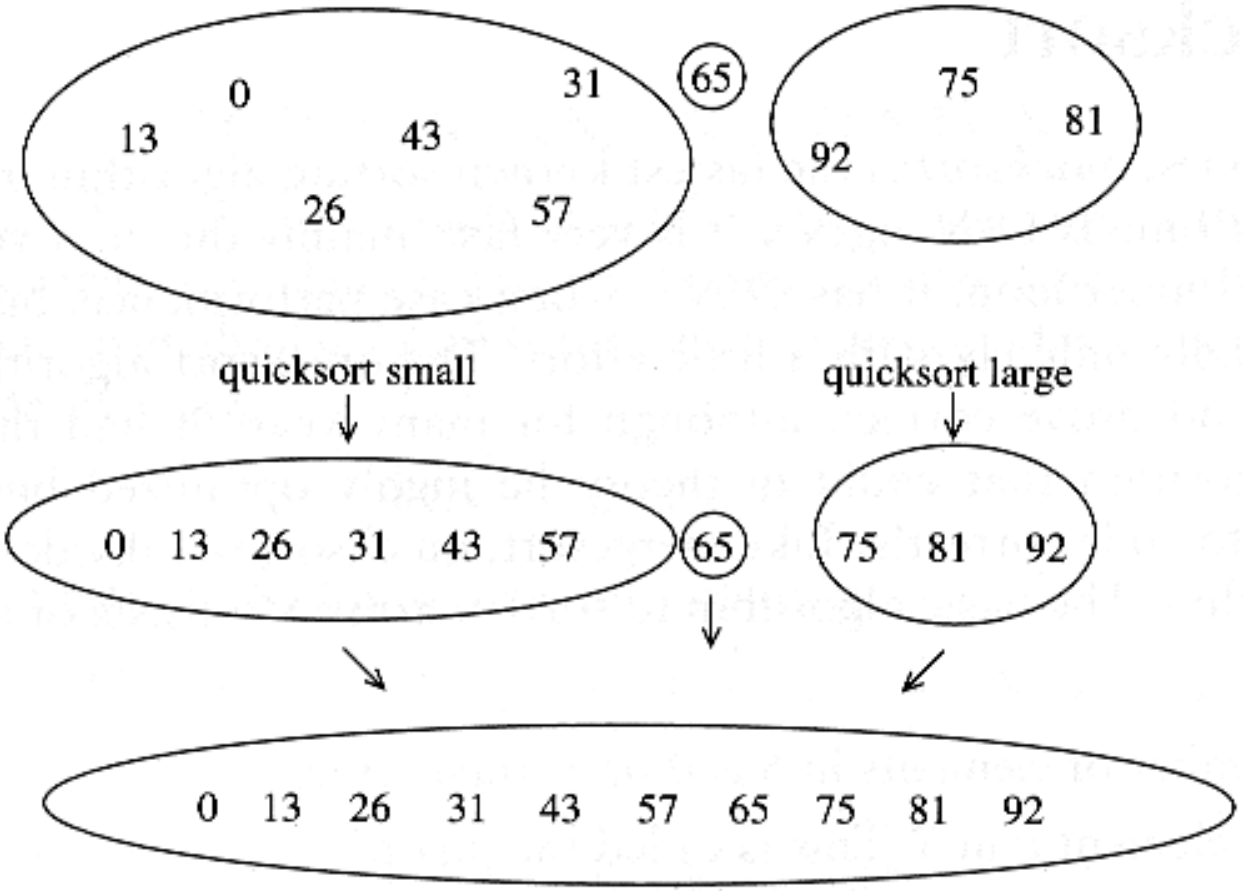



*To simplify, we may assume that we don't have repetitive elements. So to ignore the 'equality' case.*

# Quicksort: Example



# Quicksort: Example ..



# Basic Quicksort Code

```
void quicksort(int* data, int begin, int end)
{
    int pivot_index = partition(data, begin, end); // Partition to 2 subsets
    quicksort(data, begin, pivot_index - 1);       // Sort the smaller items
    quicksort(data, pivot_index + 1, end);         // Sort the greater items
}
```

- Many ways to implement, but even the slightest deviations may cause surprisingly bad results.
  - it turns out it is hard to write it correctly ☹
  - many efficient implementation are not stable: the partition step does not preserve the ordering of elements with the same value.
- Desirable: sizes of the 2 subsets are as equal as possible
- Major issue: How to determine a good pivot?

# Using the 1st Element as the Pivot?

- Use the first element as pivot
  - if the inputs are really random => ok
  - if the input is pre-sorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls => results in $O(n^2)$ behavior
- Choose the pivot randomly
  - generally safe
  - but random number generation itself can be expensive

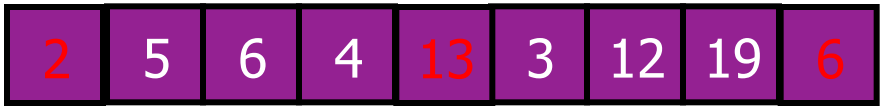
# Exact Median as the Pivot?

- If the median of the data array is used
  - Partitioning always cuts the array into 2 equal subsets
- An optimal quicksort: O(N log N)
- However, it is hard to find the exact median
  - chicken-egg problem
  - e.g., need to sort an array to pick the value in the middle!
- Need to find an approximation to the exact median.

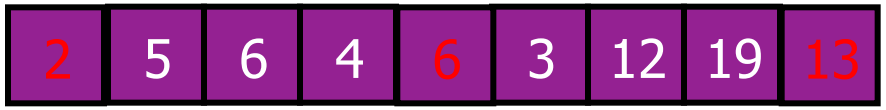
# Quicksort Pivot: Median of 3 --- Algorithm

- Compare just three elements: the leftmost, rightmost and the center elements.
- Swap these elements if necessary so that
  - A[begin] = the smallest
  - A[center] = median of the three elements
  - A[end] = the largest
- Pick A[center] as the pivot
- Swap A[center] and A[end– 1] so that pivot is at second last position (why?)

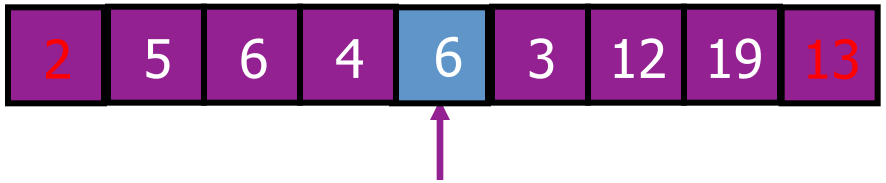
# Quicksort Pivot: Median of 3 --- Example



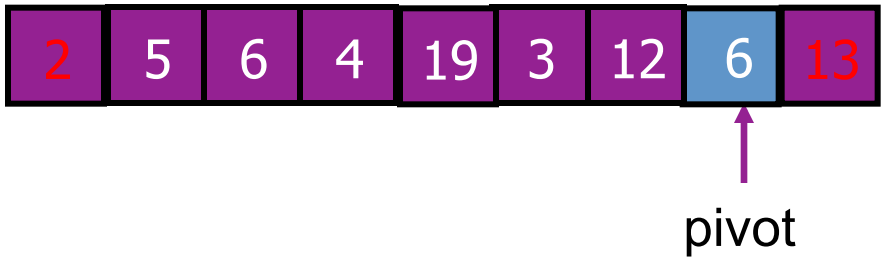

A[begin] = 2,
A[center] = 13, A[end] = 6




Swap A[center] and A[end]




Choose A[center] as pivot




Swap pivot and A[end– 1]

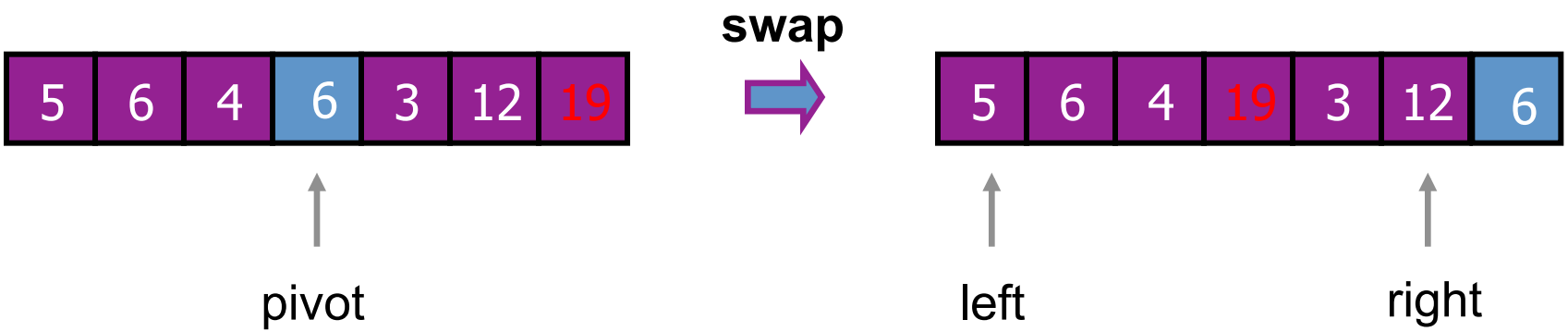
Note we only need to partition A[begin+1 … end–2]. Why?

# Quicksort Pivot: Median of 3 --- Code

```
#include <algorithm>
const int& median3(int* data, int begin, int end)
{
    int center = (begin + end)/2;

    if (data[begin] > data[center])
        std::swap(data[begin], data[center]);
    if (data[begin] > data[end])
        std::swap(data[begin], data[end]);

    // Now data[begin] is the smallest among the 3 items
    if (data[center] > data[end])
        std::swap(data[center], data[end]);

    // Place pivot as the last but one item
    std::swap(data[center], data[end - 1]);
    return data[end - 1];
}
```

# Quicksort: Partitioning

- Want to partition an array A[begin ... end] in place.
- First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[end])
- Let *left* start at the first element and *right* start at the next-to-last element: *left = begin*, *right = end-1*

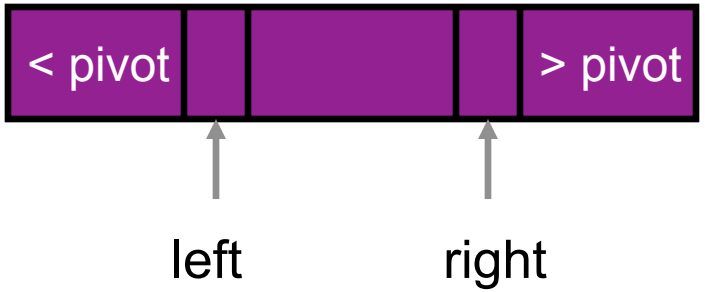


# Quicksort: Partitioning ..

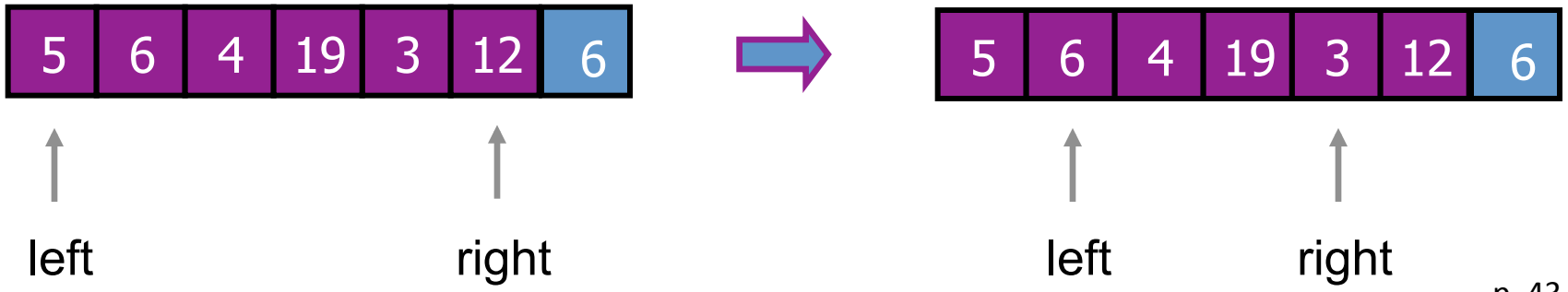
- Want to have

  *A[x]* <= pivot, for *x* < *left*

  *A[x]* >= pivot, for *x* > *right*

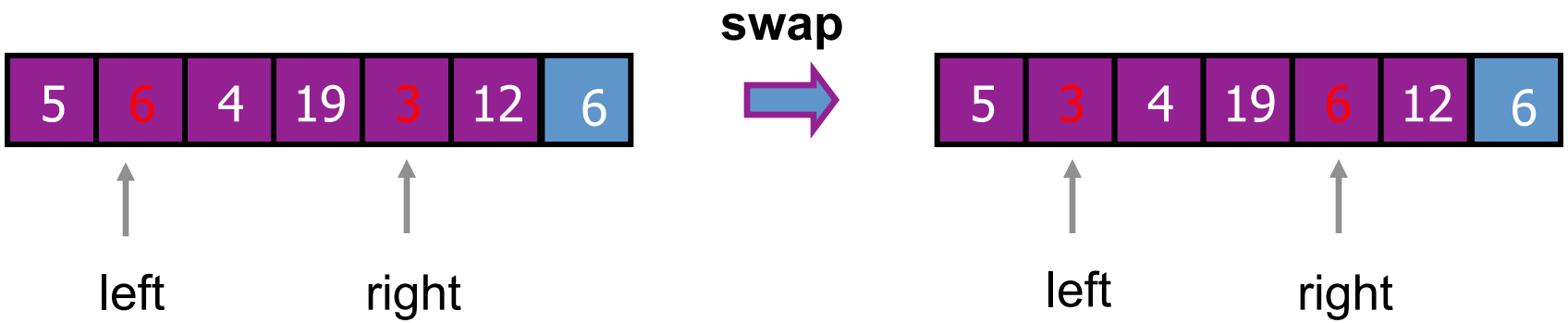



- As long as *left* < *right*
  - Move *left* to the right, skipping over elements < pivot
  - Move *right* to the left, skipping over elements > pivot
  - When both *left* and *right* have stopped
    - *A[left]* >= pivot and *A[right]* <= pivot



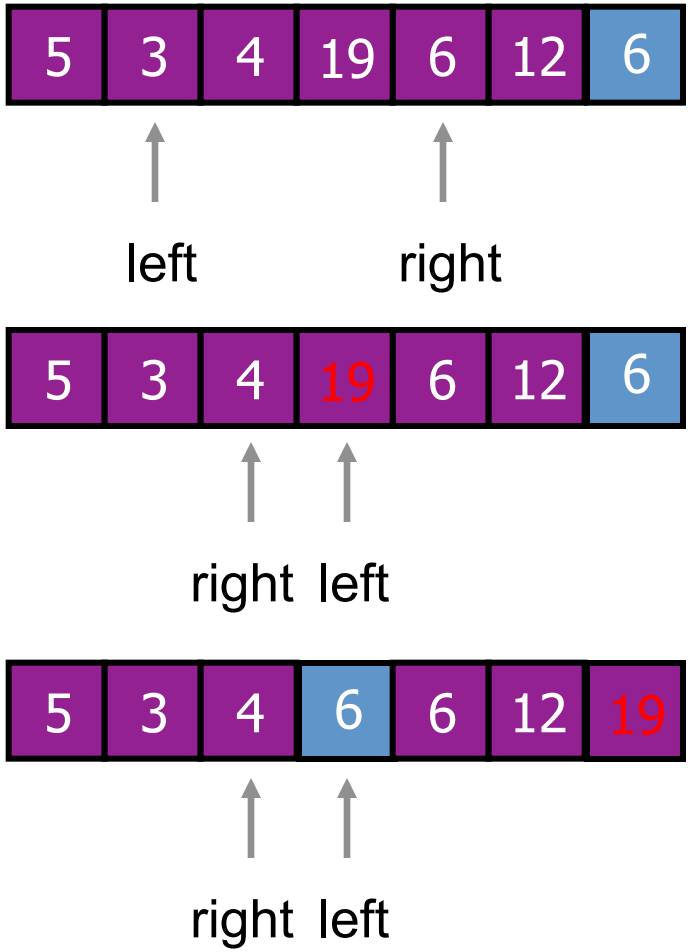
# Quicksort: Partitioning ...

- When *left* and *right* stop, and *left* is to the left of *right*
  - Swap *A[left]* and *A[right]*
    - larger element is pushed to the right and smaller element is pushed to the left
  - After swapping
    - *A[left]* <= pivot and *A[right]* >= pivot
  - Repeat the process until *left* and *right* cross each other.



# Quicksort: Partitioning ....

- When *left* and *right* cross
  - Swap *A[left]* and pivot
- *right* will not run past the beginning
  - because A[begin] <= pivot
- *left* will not run past the end
  - because A[end-1] = pivot
- Result:
  - *A[x]* <= pivot, for x < *left*
  - *A[x]* >= pivot, for x > *right*



# Quicksort: Partitioning Code

```
int partition(int* data, int begin, int end) // Return the pivot index
{
    const int& pivot = median3(data, begin, end);
    int left = begin;
    int right = end - 1;

    while (true)
    {
        while (data[++left] < pivot) { }  // Skip all smaller items
        while (data[--right] > pivot) { } // Skip all bigger items

        if (left < right)
            std::swap(data[left], data[right]);
        else
            break;                        // The 2 indices cross over
    }

    std::swap(data[left], data[end-1]);   // Put pivot in its place
    return left;
}
```

# Partitioning and Duplicates

- Partitioning so far defined is ambiguous for duplicate elements (the equality is included for both sets).
- Its 'randomness' makes a 'balanced' distribution of duplicate elements in the 2 subsets.
- When all elements are identical:
  - both left and right stop right away in each iteration

    => many swaps
  - but they cross in the middle

    => partition is balanced

    => O(N log N)

# Quicksort is not Good for Short arrays

- For very small arrays, quicksort does not perform as well as insertion sort.
  - How small depends on many factors, such as the time spent making a recursive call, the compiler, etc.
- Do not use quicksort recursively for small arrays.
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort.

# Practical Implementation of Quicksort

```
void quicksort(int* data, int begin, int end)
{
    int insertion_sort(int* data, int N); // Forward declaration

    // >= 2 for median-3 partitioning, but 10-20 for efficiency
    const int QSORT_MIN_SIZE = 10;

    if (end - begin > QSORT_MIN_SIZE)
    {
        // Partition to 2 subsets of similar sizes
        int pivot_index = partition(data, begin, end);

        // Recursively sort the smaller items
        quicksort(data, begin, pivot_index - 1);

        // Recursively sort the greater items
        quicksort(data, pivot_index + 1, end);
    }
    else // Insertion sort is better for short list of data
        insertion_sort(data + begin, end - begin + 1);
}
```

# Quicksort Analysis

- Assumptions
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time = O(1)
  - partitioning: linear time = O(N)
  - running time of the 2 recursive calls
- T(N) = T(M) + T(N-M-1) + cN
  - c is a constant
  - M is the number of elements in S1

# Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

# Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= N + cN \log N = O(N \log N)
\end{aligned}
$$

divide the whole equation by N

add up all equations; remove identical terms on both sides

# Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N)

# Quicksort vs. Mergesort

- Both quicksort and mergesort take O(N log N) in the average case.
- Quicksort: the inner loop consists of an increment/ decrement (by 1, which is fast), a test and a jump.
- Partitioning in quicksort and merging in mergesort both take O(N) time.
- But, mergesort requires copying results back to the original data array => may be slower.
- Exact performance depends on the speed of data copying data, data comparison, etc.
  - Java sorting lib: mergesort
  - C++ STL sort: quicksort

# Lower Bound for Sorting
# Linear-Time Sorting Algorithms

# Lower Bound for Sorting

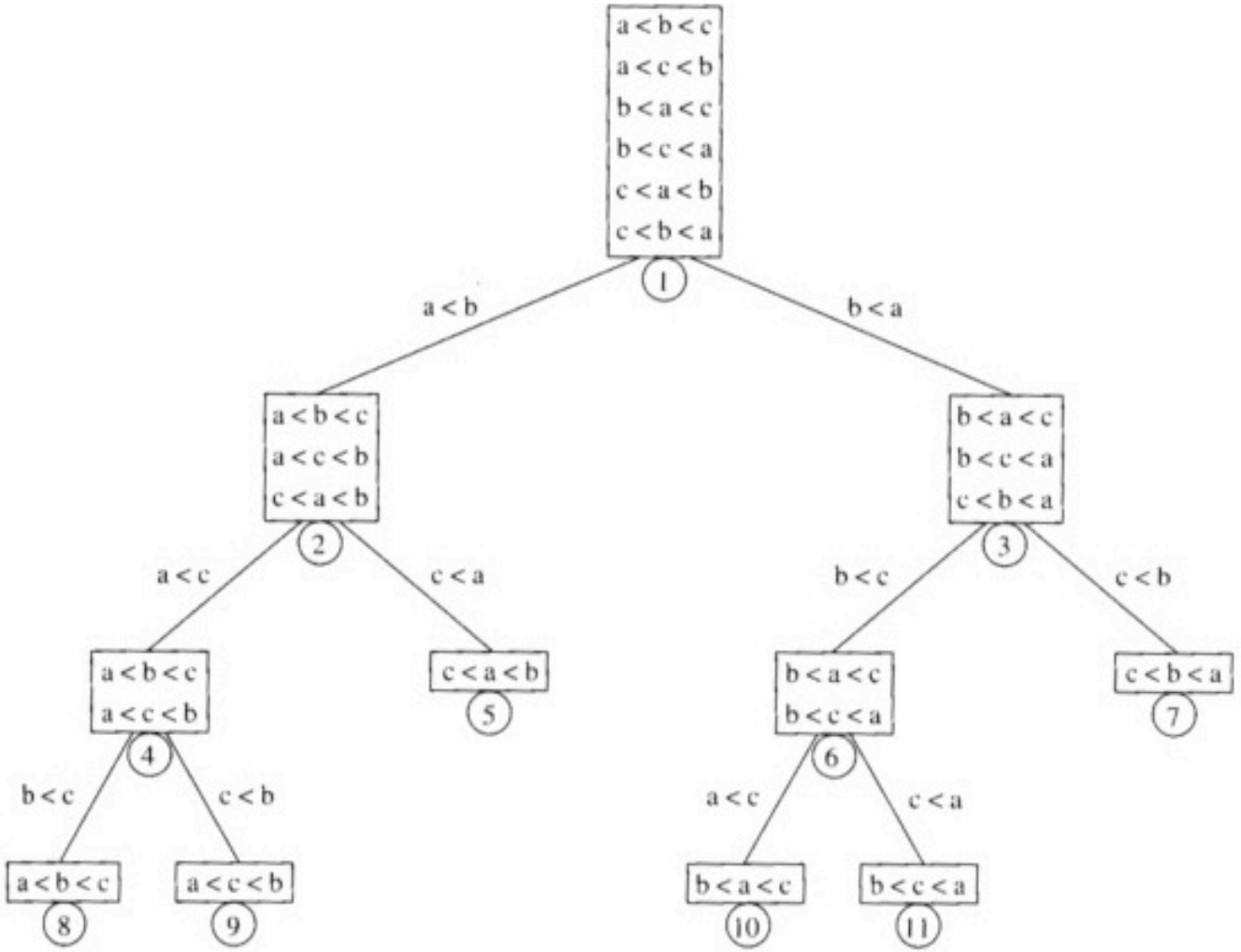
- Mergesort
  - worst-case running time is O(N log N)
- Are there better algorithms?
- Theorem:

  Any sorting algorithm based only on comparisons to sort N elements takes $\Omega$(N log N) comparisons in the worst case (worse-case input).

# Decision Trees for Sorting

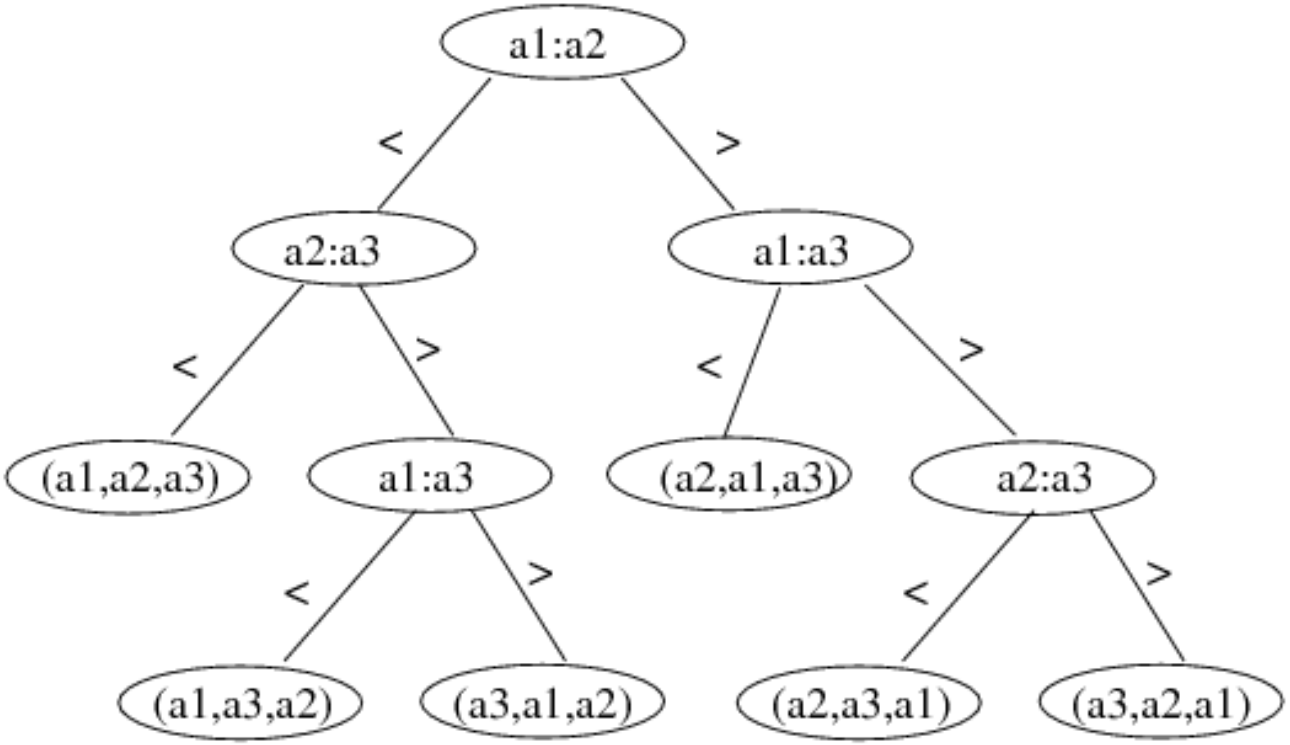
- Suppose we want to sort N distinct elements.
- How many possible orderings do we have for N elements?
  - There are N! possible orderings
  - e.g., the sorted output for { a, b, c } can be one of
    - a b c, b a c, a c b, c a b, c b a, b c a.
- Any comparison-based sorting process can be represented as a binary decision tree.
  - Each node represents a set of possible orderings, consistent with all the comparisons that have been made.
  - Tree edges are results of intermediate comparisons.
  - Leaves are the results.

# Decision Tree for Sorting 3 Elements



# Decision Tree for Insertion Sort

- Different algorithms have different decision trees.
- There exists a root-to-leaf path that arrives at a leaf containing inputs in sorted order.
- For insertion sort, the longest path in such a tree is $O(N^2)$.



# Lower Bound for Sorting

- Worst-case: number of comparisons used by a sorting algorithm is equal to the depth of the deepest leaf.
- Average case: number of comparisons is equal to the average depth of the leaves
- A decision tree to sort N elements must have N! leaves.
  - a binary tree of depth d has at most $2^d$ leaves

    $\Rightarrow$ a binary tree with $2^d$ leaves must have depth at least d

    $\Rightarrow$ the decision tree with N! leaves must have depth at least $\lceil \log_2 (N!) \rceil$
- Thus, any sorting algorithm based only on comparisons between elements requires at least $\lceil \log_2(N!) \rceil$ comparisons in the worst case.

# Lower Bound for Sorting ..

$$
\begin{aligned}
\log(N!) &= \log(N(N-1)(N-2)\cdots 2\cdot 1) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
&\geq \frac{N}{2}\log\frac{N}{2} \\
&= \frac{N}{2}\log N - \frac{N}{2} \\
&= \Omega(N\log N)
\end{aligned}
$$

- Therefore, any sorting algorithm based on comparisons between elements requires $\Omega(N \log N)$ comparisons.

# Linear Time Sorting Algorithms

- Can we do better?
- Yes, we may even obtain O(N) sorting algorithm if the input has special structure or extra information, e.g.,
  - uniformly distributed
  - every number can be represented by r digits
- Linear time sorting algorithms
  - bucket sort
  - radix sort

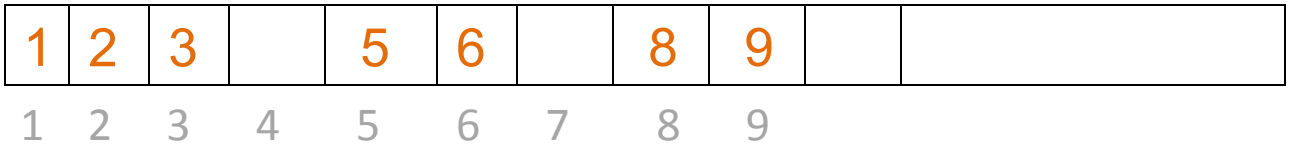
# Bucket Sort with No Duplicates

- To sort N +ve integers *A[1 .. N]* in the range of 1 to *M >= N*. These integers may be keys of actual storing objects.

1. Define an array *B[1 .. M]*; initialize all to 0 $\Rightarrow$ O(M)
2. Scan through the input list *A[i]*; insert *A[i]* into *B[A[i]]* $\Rightarrow$ O(N)
3. Scan B once and read out the non-zero integers $\Rightarrow$ O(M)

- Total time: O(M + N)
  - if M is O(N), then total time is O(N).
  - Can be bad if range is very big, e.g. M = $O(N^2)$.

- Example: To sort { 8 1 9 5 2 6 3 }. N = 7, M = 9.




- Output: 1 2 3 5 6 8 9

# Bucket Sort with Duplicates

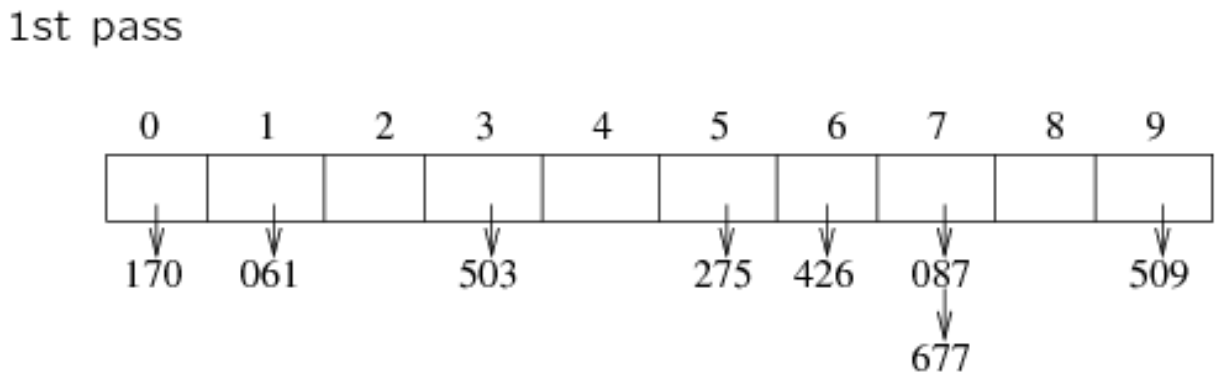
- *B* is an array of linked lists.
- Each cell in B has 2 pointers:
  - head points to the beginning of a linked list.
  - tail points to the end of the linked list.
- *A[j]* is inserted at the end of the list headed by *B[A[j]]*
  => stable sort
- After all elements of array *A* are inserted, array *B* is sequentially traversed and each non-empty list is printed out from head to tail.
- Time: O(M + N)

# Radix Sort

- Given N integers in the range 0 to ($r^k - 1$)
  - i.e., they can be represented by at most k digits
  - $d_1 d_2 \ldots d_k$ where $d_i$ are digits in base $r$
  - $d_1$: most significant digit
  - $d_k$: least significant digit
- Algorithm
  1. Bucket sort by the least significant digit first.
     => numbers with the same $d_k$ digit go to same bin
  2. Re-order the numbers according to their current sorted order from bucket 0 to bucket ($r$-1).
  3. Sort by the next least significant digit.
  4. Continue this process until the numbers are sorted on all $k$ digits.

# Radix Sort: Example 1

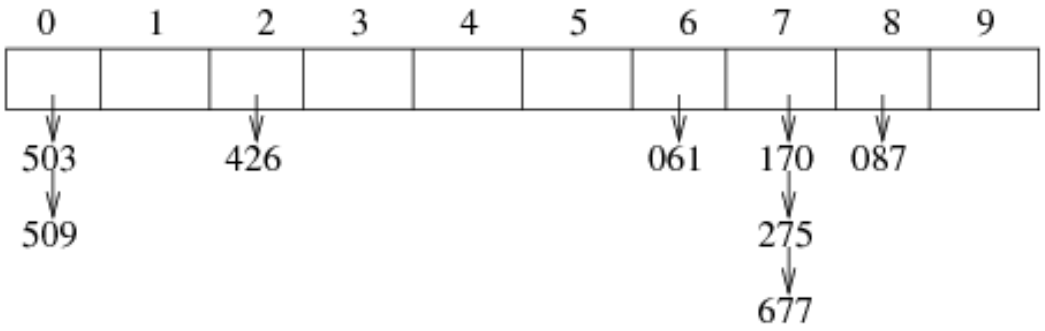
- Example: 275, 087, 426, 061, 509, 170, 677, 503
- 1st pass: Least-significant-digit first.



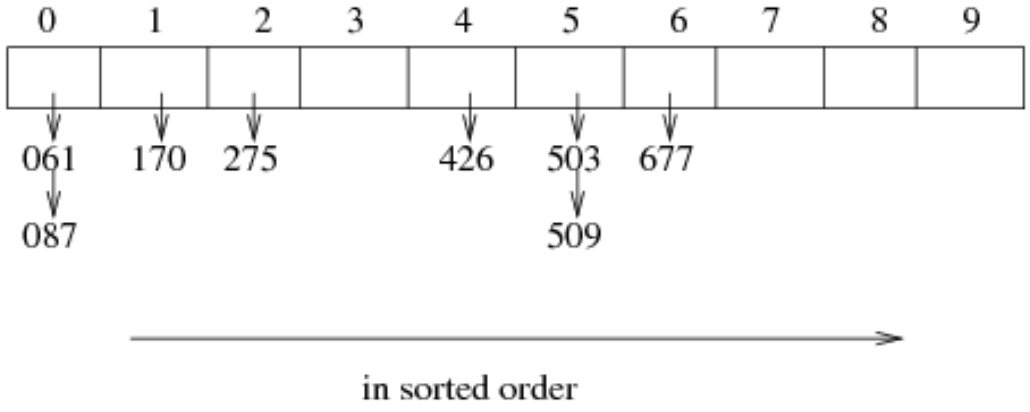

  - Read the currently sorted numbers from the buckets:
    - 170 061 503 275 426 087 677 509

# Radix Sort: Example 1 ..

2nd pass Before: 170 061 503 275 426 087 677 509




3rd pass Before: 503 509 426 061 170 275 677 087




After: 061 087 170 275 426 503 509 677

# Radix Sort: Example 2

- One commonly uses radix sort to sort cards.
- Use 2 digits to represent each card: $d_1 d_2$
  - $d_1$ = ♦ ♣ ♥ ♠: base 4
    - ♦ ≤ ♣ ≤ ♥ ≤ ♠
  - $d_2$ = A, 2, 3, ...J, Q, K: base 13
    - A ≤ 2 ≤ 3 ≤ ... ≤ J ≤ Q ≤ K
  - e.g., ♦2 ≤ ♣2 ≤ ♥5 ≤ ♠K

# Radix Sort: Pseudo-Code

- A = input array, n = number of data, d=# of digits

```
Algorithm RadixSort(A, n, d)
1.    for 0 ≤ p ≤ 9                                  ← assume base = 10
2.        do Q[p] := empty queue;
3.    D := 1;
4.    for 1 ≤ k ≤ d                                  ← Consider the k-th digit
5.        do
6.            D := 10 * D;
7.            for 0 ≤ i < n                          ← Scan A[i], put into correct bucket
8.                do t := (A[i] mod D) div (D/10);
9.                   enqueue(A[i], Q[t]);
10.           j := 0;
11.           for 0 ≤ p ≤ 9                          ← Put numbers back to the original array
12.               do while Q[p] is not empty
13.                      do A[j] := dequeue(Q[p]);
14.                         j := j + 1;
```

# Radix Sort: Summary

- Increasing the base $r$ decreases the number of digits $k$.
- Running time analysis:
  - $k$ passes over the numbers: 1 bucket sort for each digit
  - each pass takes $2N$
  - total: $O(2Nk) = O(Nk)$
- Radix sort is not based on comparisons; the values are used as array indices.
- If all $N$ input values are distinct, then $k = \Omega(\log N)$.
  - e.g., in binary digits, to represent 8 different numbers, we need at least 3 digits.
  - The running time of radix sort also becomes $\Omega(N \log N)$.