CHAPTER 7
SORTING

Iris Hui-Ru Jiang

Fall 2008

Sorting

- **Contents**
  - Comparison sort
    - Bubble sort
    - Selection sort
    - Insertion sort
    - Merge sort
    - Quick sort
    - Heap sort
    - Introspective sort (Introsort)
- **Readings**
  - Chapter 7

The Sorting Problem (1/2)

- **Input:**
  - A list of n records (R_1, R_2, …, R_n)
    - Each record R_i has a key value K_i
- **Output:**
  - A permutation σ, ($R_{\sigma(1)}$, $R_{\sigma(2)}$, …, $R_{\sigma(n)}$), s.t.
    - Key values are in nondecreasing order: $K_{\sigma(1)} \leq K_{\sigma(2)} \leq \ldots \leq K_{\sigma(n)}$

The Sorting Problem (2/2)

- **Problem: Sorting**
  - Sort n integers in nondecreasing order
  - Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$, $n \geq 1$
  - Output: a permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ s.t. $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- **Example:**
  - Input: (5, 1, 4, 2, 8)
  - Output: (1, 2, 4, 5, 8)

In-Place and Stable Sorting

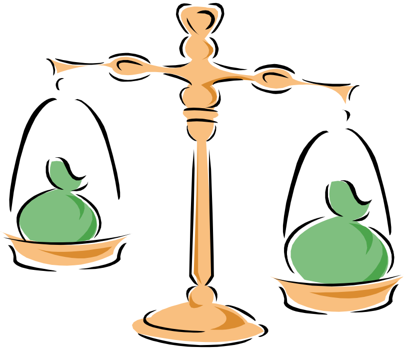
- **In-place sorting:**
  - Only a constant # of variables are stored outside the working array
- **The permutation (output) is not unique if several key values are identical**
- **Stable sorting:**
  - Numbers with the same value appear in the output array in the same order as they do in the input array

Internal/External Sorting

- **Internal sorting:**
  - The input list is small enough so that the entire sort can be carried out in main memory
- **External sorting:**
  - The input list is too large $\Rightarrow$ disk or tape

Comparison Sort

- **A comparison sort:**
  - A type of sorting algorithm that determines which of two elements should occur first in the final sorted list by a single abstract comparison operation
    - Often a “less than or equal to” operator
  - Examples
    - Bubble sort
    - Selection sort
    - Insertion sort
    - Quick sort
    - Merge sort
    - Heap sort
    - Introspective sort (Introsort)
  - Animated sorting algorithms
    - http://www.sorting-algorithms.com/



Bubble Sort

- **Repeat:**
  - Compare two items at a time
  - Swap them if they are in the wrong order ⇒ bubble up
- **e.g., (5, 4, 1, 2, 8), first pass:**

  5 4 1 2 8

  4 5 1 2 8

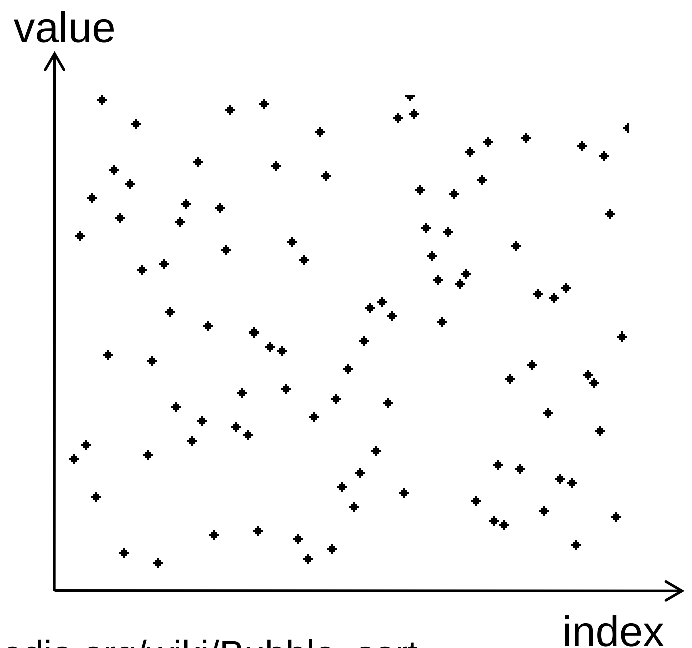
  4 1 5 2 8

  4 1 2 5 8

  4 1 2 5 8 ⇒ 8 is the largest

```
template <class T>
void BubbleSort(T *a, const int n) {
// input array a with n keys: a[1:n]
    for (int j = n; j > 1; j--)
        for (int k = 1; k < j; k++)
            if (a[k+1] < a[k]) swap(a[k+1], a[k]);
}
```

$O(n^2)$

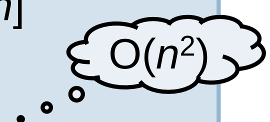


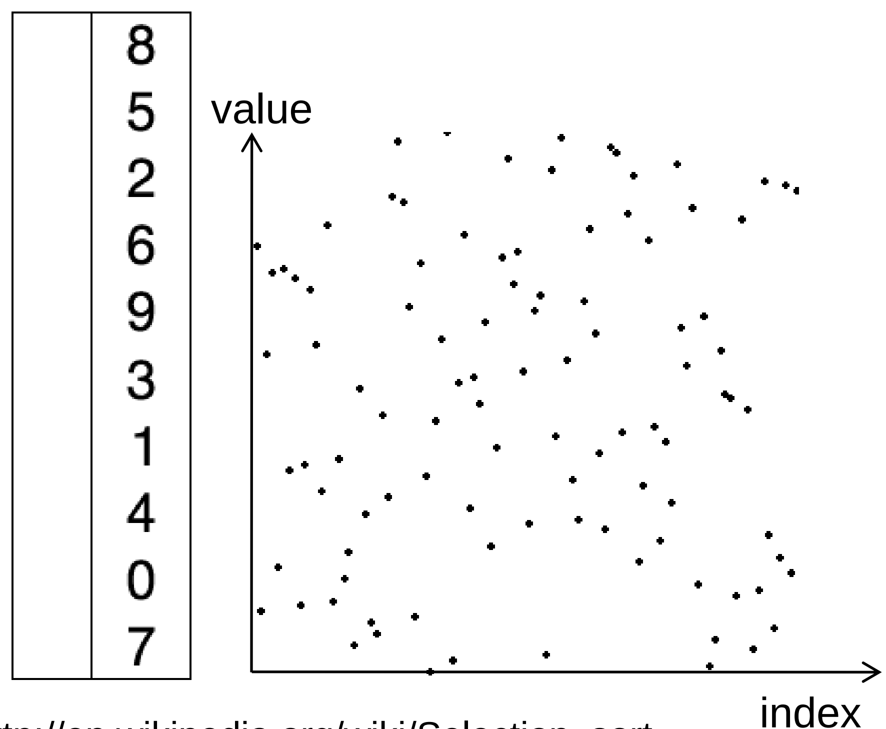
http://en.wikipedia.org/wiki/Bubble_sort

Selection Sort

- **Repeat:**
  - Find the smallest value in the list
  - Swap it with the value in the first position of the unsorted list

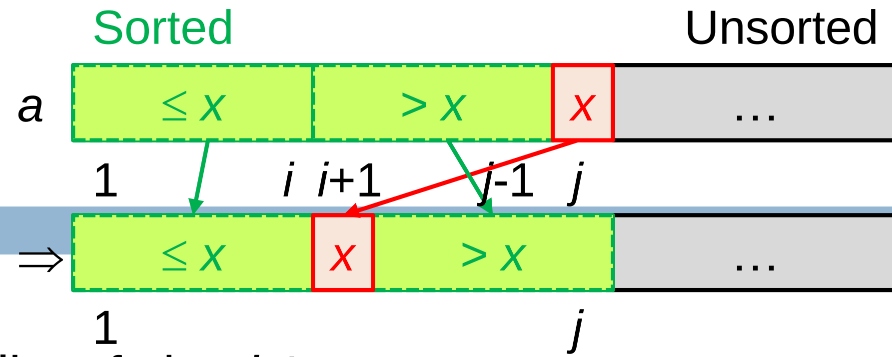
```
template <class T>
void SelectionSort(T *a, const int n) {
// input array a with n keys: a[1:n]
    for (int i = 1; i < n; i++) {
        min = i;
        for (int j = i+1; j <= n; j++)
            if (a[j] < a[min]) min = j;
        swap(a[i], a[min]);
    }
}
```





http://en.wikipedia.org/wiki/Selection_sort

Insertion Sort (1/2)



- **Repeat:**
  - Insert the j^{th} item into a sorted list of size j-1
  - Result in a sorted list of size j

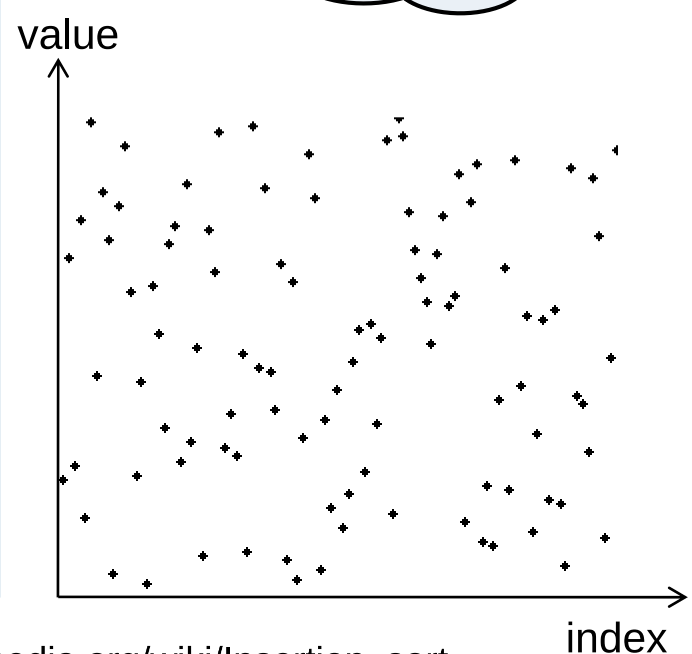
Stable?
In-place?

Best case?
Worst case?

```
template <class T>
void InsertionSort(T *a, const int n) {
// input array a with n keys: a[1:n]
  for (int j = 2; j <= n; j++) { // skip j=1
     // Insert a[j] into the sorted array a[1..j-1]
     a[0] = a[j]; // a[0]: temp keeps a[j]
     int i = j-1;
     while (a[0] < a[i]) { // a[1:i]: sorted
        a[i+1] = a[i];
        i--;
     }
     a[i+1] = a[0];
  }
}
```

O(n^2)

O(j)



http://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort (2/2)

- **Worst case:**

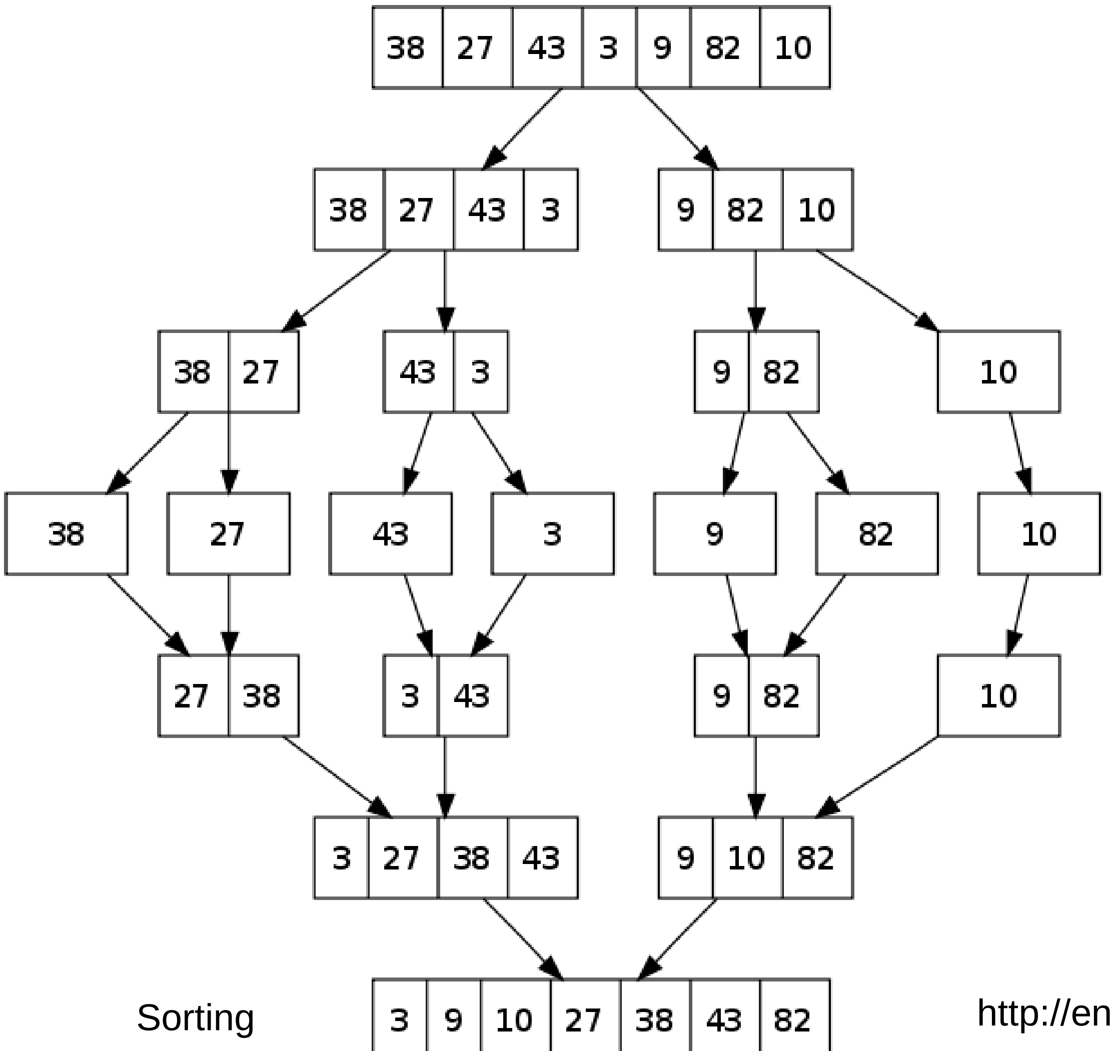
j	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

- **Best case:**

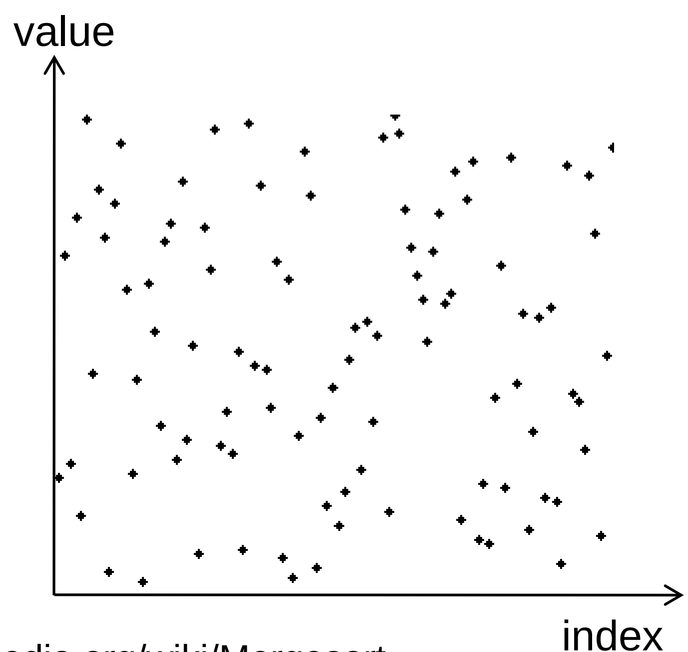
j	[1]	[2]	[3]	[4]	[5]	
-	**1**	2	3	4	5	
2	**1**	**2**	3	4	5	O(1)
3	**1**	**2**	**3**	4	5	O(1)
4	**1**	**2**	**3**	**4**	5	O(1)
5	**1**	**2**	**3**	**4**	**5**	O(1)

Recursive Merge Sort (1/4)

- **Divide-and-conquer:**



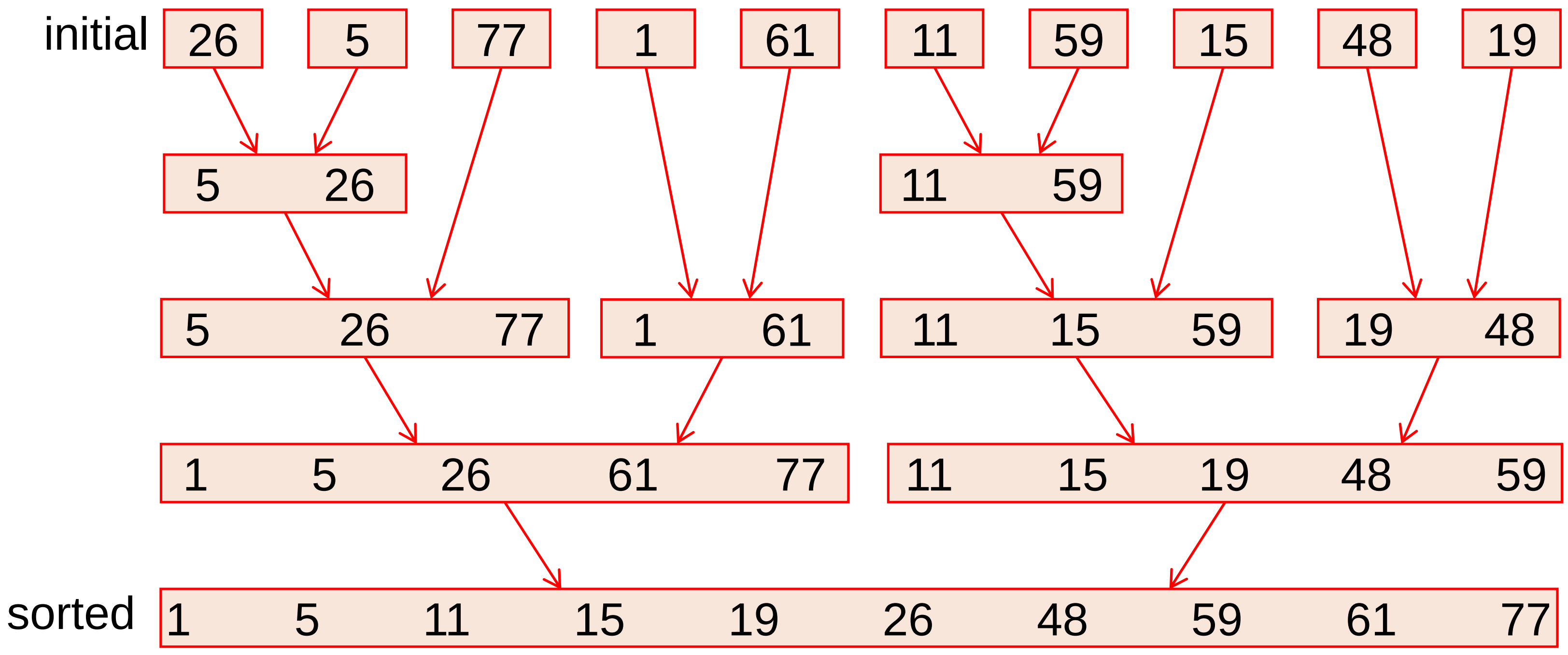
Sorting



http://en.wikipedia.org/wiki/Mergesort

Recursive Merge Sort (2/4)

- **Idea: divide-and-conquer**
  - **Divide**: divide a list into 2 equally-sized sublists
  - **Conquer**: recursively and individually sort the 2 sublists
  - **Combine**: merge 2 sorted sublists into 1



Recursive Merge Sort (3/4)


```
template <class T>
int rMergeSort(T *a, int* link, const int left, const int right) {
// sort a[left..right] into nondecreasing order
// link[i] is initially 0 for all i
// return the index of the first element in the sorted chain

    if (left >= right) return left; // termination condition

    int mid = (left+right)/2; // divide w.r.t. mid
    return ListMerge(a, link,
                     rMergeSort(a, link, left, mid), // recursively sort left half
                     rMergeSort(a, link, mid+1, right)); // recursively sort right half
}
```

Stable?
In-place?

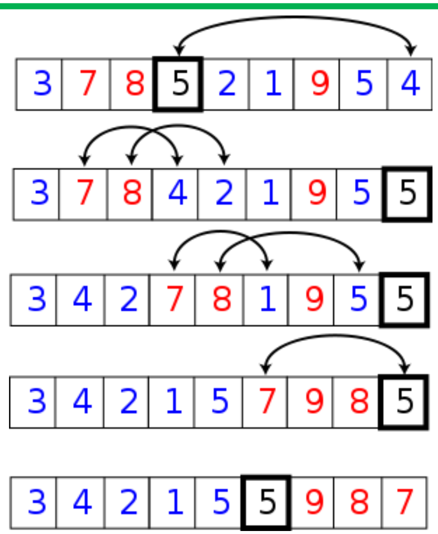
- ***ListMerge* merges 2 sublists starting at *start*1 and *start*2 and returns the 1st position of the resulting list**
  - int *ListMerge* (*T* * *a*, int* *link*, const int *start*1, const int *start*2)
  - You may implement *ListMerge* in your way
- **Time complexity: $T(n) = 2T(n/2)+O(n)$, $T(1)=O(1) \Rightarrow O(n\lg n)$**

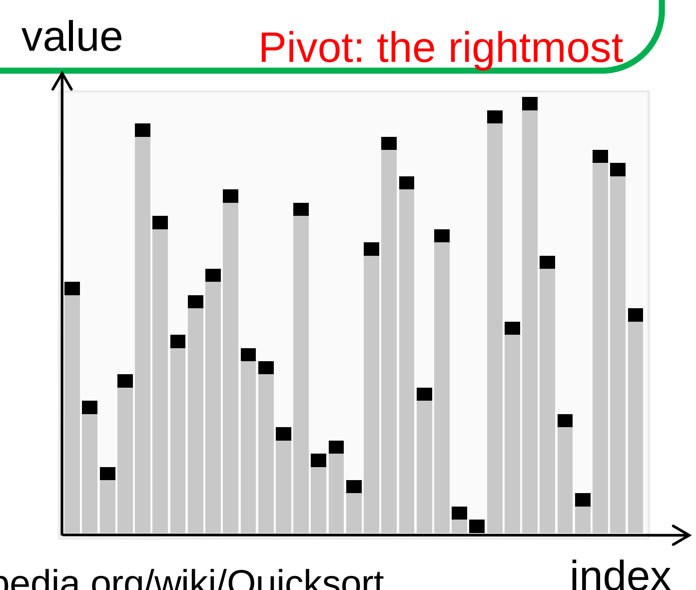
Recursive Merge Sort (4/4)


```
template <class T>
int ListMerge(T *a, int* link, const int start1, const int start2) {
// The sorted chains beginning at start1 and start2, respectively, are merged.
// link [0] is used as a temporary header. Return start of merged chain.

    int iResult = 0; // last record of result chain
    for (int i1 = start1, i2 = start2; i1&& i2;)
        if (a[i1] <= a[i2]) {
            link [iResult ] = i1;
            iResult = i1; i1 = link [i1];
        }
        else {
            link [iResult ] = i2;
            iResult = i2; i2 = link [i2];
        }

    // attach remaining records to result chain
    if (i1 == 0) link [iResult ] = i2;
    else link [iResult ] = i1;
    return link [0];
}
```

Quick Sort (1/3)

- **Idea: divide-and-conquer**
  - Select a pivot and put it on the correct position
  - Partition into left sublist (≤) and right sublist (≥)
  - **Divide:** Select a pivot and find its correct position s.t. all records to its left/right with keys ≤/≥ key[pivot]
  - **Conquer:** Recursively and individually sort the left/right sublists
    - No more comparison between the left and right sublists
  - **Combine:** Do nothing





http://en.wikipedia.org/wiki/Quicksort

Quick Sort (2/3)

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

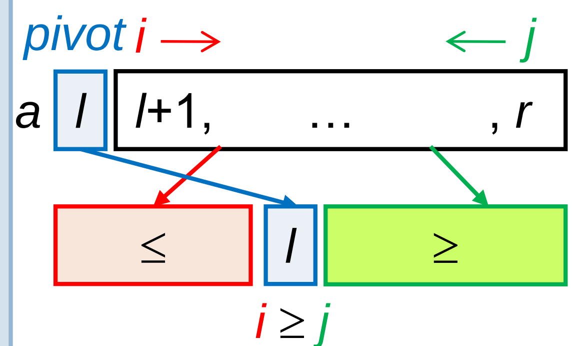
Pivot: the leftmost

Quick Sort (3/3)


```
template <class T>
void QuickSort(T *a, const int left, const int right) {
// sort a[left..right] into nondecreasing order
// pivot: a[left] (or arbitrarily choose one and swap with a[left])
// i: a[m] ≤ a[left] for m < i; j: a[m] ≥ a[left] for m > j

   if (left < right) { // termination condition check
      int i = left,
         j = right + 1,
         pivot = a[left];
      // find the correct position of pivot; partition the array
      do {
         do i++; while (a[i] < pivot); // search from left
         do j--; while (a[j] > pivot); // search from right
         if (i < j) swap(a[i], a[j]);
      } while (i < j); // until i≥j
      swap(a[left], a[j]); // j is the correct position
      QuickSort(a, left, j-1);
      QuickSort(a, j+1, right);
   }
}
```

Stable?
In-place?

QuickSort(*a*, 1, *n*)

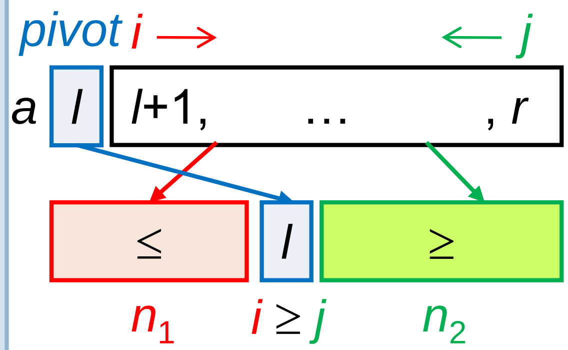


Analysis on Quick Sort (1/2)


```
template <class T>
void QuickSort(T *a, const int left, const int right) {
    if (left < right) {
        int i = left,
            j = right + 1,
            pivot = a[left];
        do {
            do i++; while (a[i] < pivot); // search from left
            do j--; while (a[j] > pivot); // search from right
            if (i < j) swap(a[i], a[j]);
        } while (i < j); // until i≥j
        swap(a[left], a[j]); // j is the correct position
        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
}
```

O(n)

$T(n_1)+T(n_2)$

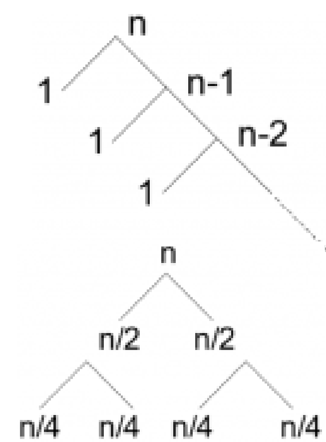


Best case?
Worst case?

- **Time complexity: $T(n) = T(n_1)+T(n_2)+O(n)$, $T(1)=O(1)$**

Analysis on Quick Sort (2/2)

- **Time complexity: $T(n) = T(n_1)+T(n_2)+O(n)$, $T(1)=O(1)$**
  - Worst-case: $O(n^2)$
    - n_1, n_2: one is 0, the other is n-1
    - The input is already sorted
  - Best-case: $O(n\lg n)$
    - $n_1==n_2==(n-1)/2$
    - Pivot is always the median
      - Median-of-3: *median*(*key*[*left*], *key*[*middle*], *key*[*right*])
      - Randomized: randomly choose one as pivot
      - Make the worst-case unlikely to occur
  - Average-case: $O(n\lg n)$



By substitution,

$T(n) \le cn + 2T(n/2)$, c: constant

$\le cn + 2(cn/2 + 2T(n/4))$

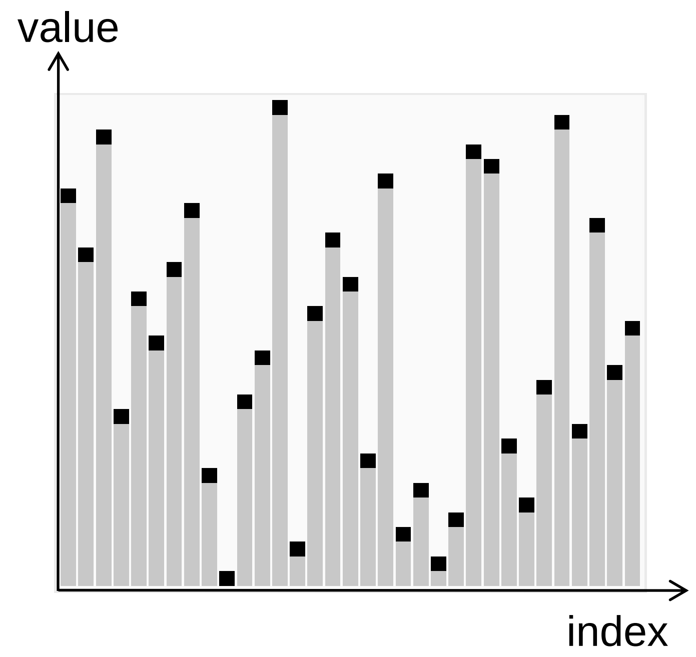
$\le 2cn + 4T(n/4)$

…

$\le cn \lg n + nT(1) = O(n\lg n)$

Heapsort

- **Ref lecture 5**
- **Idea:**
  - Build a max heap
  - Sort: repeat…
    - Swap the largest item with the last unsorted item
    - Reconstruct the heap (heapify)



Introspective Sort (Introsort) (1/2)

- **David Musser, "Introspective Sorting and Selection Algorithms," *Software: Practice and Experience*, Wiley, 27 (8): 983–993, 1997.**
- **Idea: Quicksort with "introspection"**
  - Add introspective element: monitor the recursion depth
  - Begin with quicksort
    - Median-of-3 pivot selection
  - Switch to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted
- **Time complexity: O(*n*lg*n*)**

Introspective Sort (Introsort) (2/2)

	Length (in 1000)	algorithm	assignments	comparisons	total
Random arrays	256	Introsort	9629.7	7666.5	17296.3
		Quicksort (opt.)	9603.1	7666.5	17269.6
		Quicksort	9790.2	9499.1	19289.4
		Heapsort	16423.7	17498.0	33921.7
	1024	Introsort	42482.9	33676.1	76159.0
		Quicksort (opt.)	42376.6	33676.1	76052.7
		Quicksort	43347.4	41711.0	85058.4
		Heapsort	72873.9	78192.9	151066.7

	Length (in 1000)	algorithm	assignments	comparisons	total
Median-of-3 killer sequence	64	Introsort	5806.9	5931.1	11737.9
		Quicksort (opt.)	386043.4	385804.8	771848.2
		Quicksort	770320.7	770420.7	1540741.4
		Heapsort	3636.1	3824.5	7460.6
	256	Introsort	26052.9	26799.7	52852.6
		Quicksort (opt.)	6153857.4	6152777.5	ca. 1.23*10^7
		Quicksort	-	-	-
		Heapsort	16322.3	17329.7	33652.0

http://ralphunden.net/content/tutorials/a-guide-to-introsort/

Summary of Internal Sorting

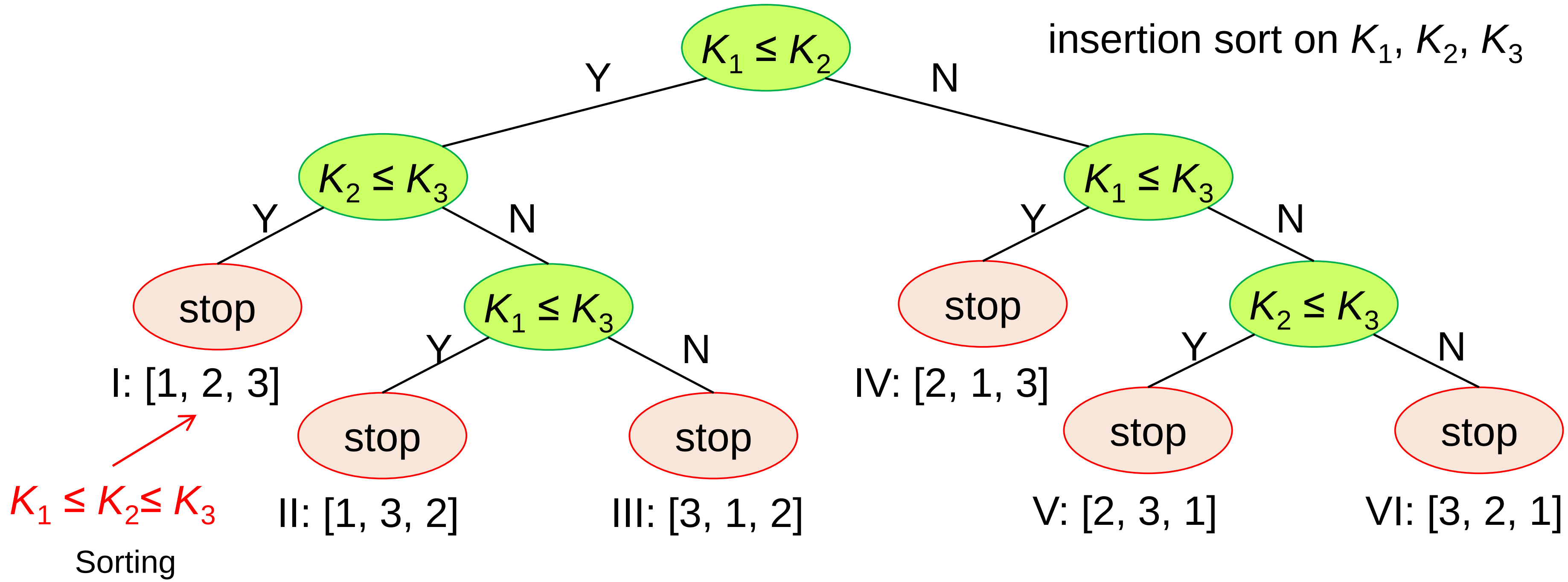
Comparison sort

How Fast Can We Sort?

- **Use only "comparisons" + "interchanges" to sort**
  - Insertion sort, quick sort, merge sort, heap sort
- **Decision tree of *n* keys**
  - # of leaves: $n!$; binary tree
  - Tree height h: at least $\lg(n!)+1$
  - Fastest: $\Omega(\lg(n!)+1) \Rightarrow \Omega(n\lg n)$

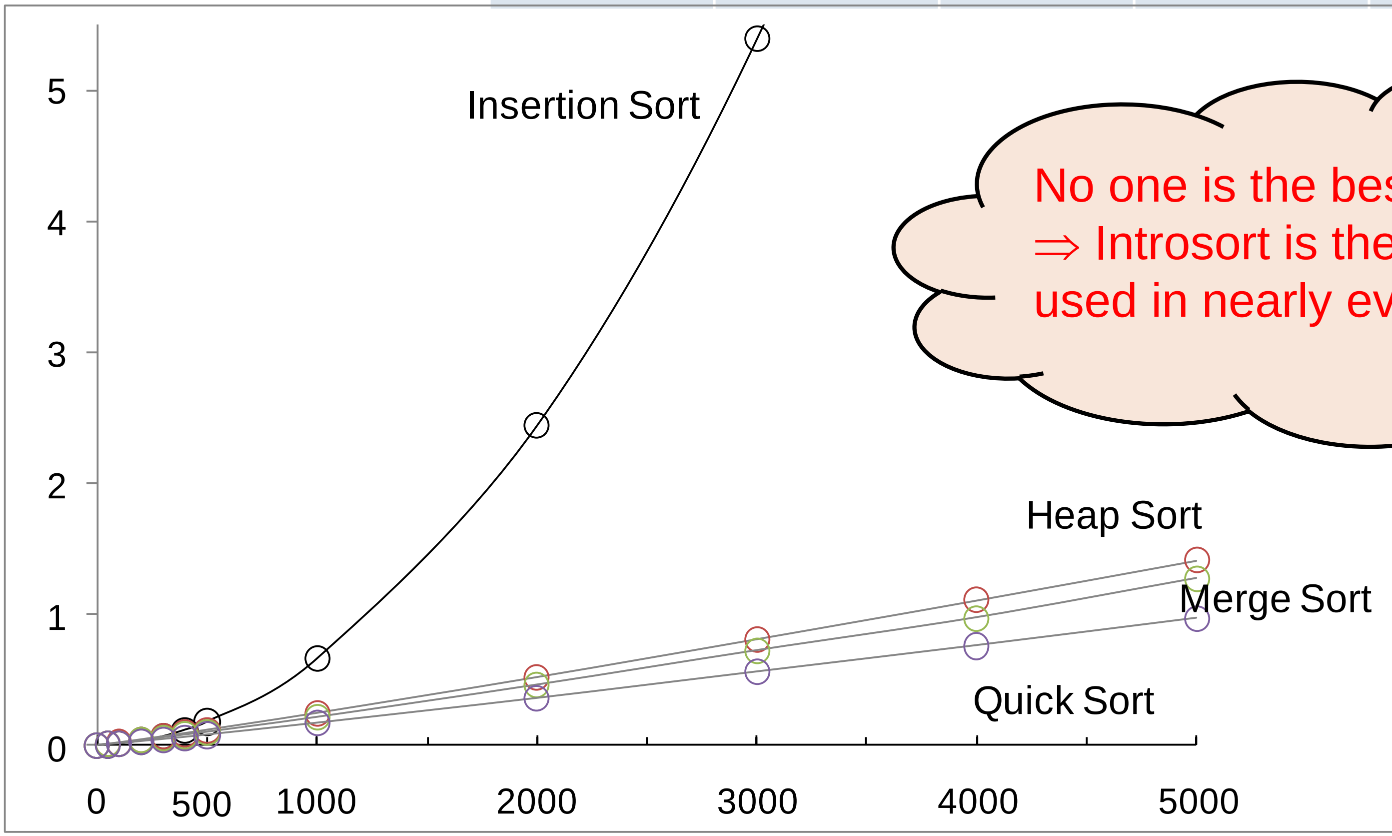
Why? $n! \leq 2^{h-1}$

insertion sort on K_1, K_2, K_3



Summary

Method	In-place	Stable	Time complexity		
			Best	Average	Worst
Insertion	Y	Y	$O(n)$	$O(n^2)$	$O(n^2)$
Quick	Y	N	$O(n\lg n)$	$O(n\lg n)$	$O(n^2)$
Merge	N	Y	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$
Heap	Y	N	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$



No one is the best for all cases
⇒ Introsort is the one that can be used in nearly every situation

Sorting