Sorting

CSE 373
Data Structures

Reading

- Reading Chapter 11
  - Sections 11.1 (a review)
  - Sections 11.2 to 11.5

Sorting

- Input: an array A of data records with a key value in each data record
  - Some sorting algorithms, e.g. Mergesort work also on linked lists
- The values must be “comparable”
  - For example: integers, strings
- Output: reorganize the elements of A such that
    - For any i and j, if $i < j$ then $A[i] \leq A[j]$

Consistent Ordering

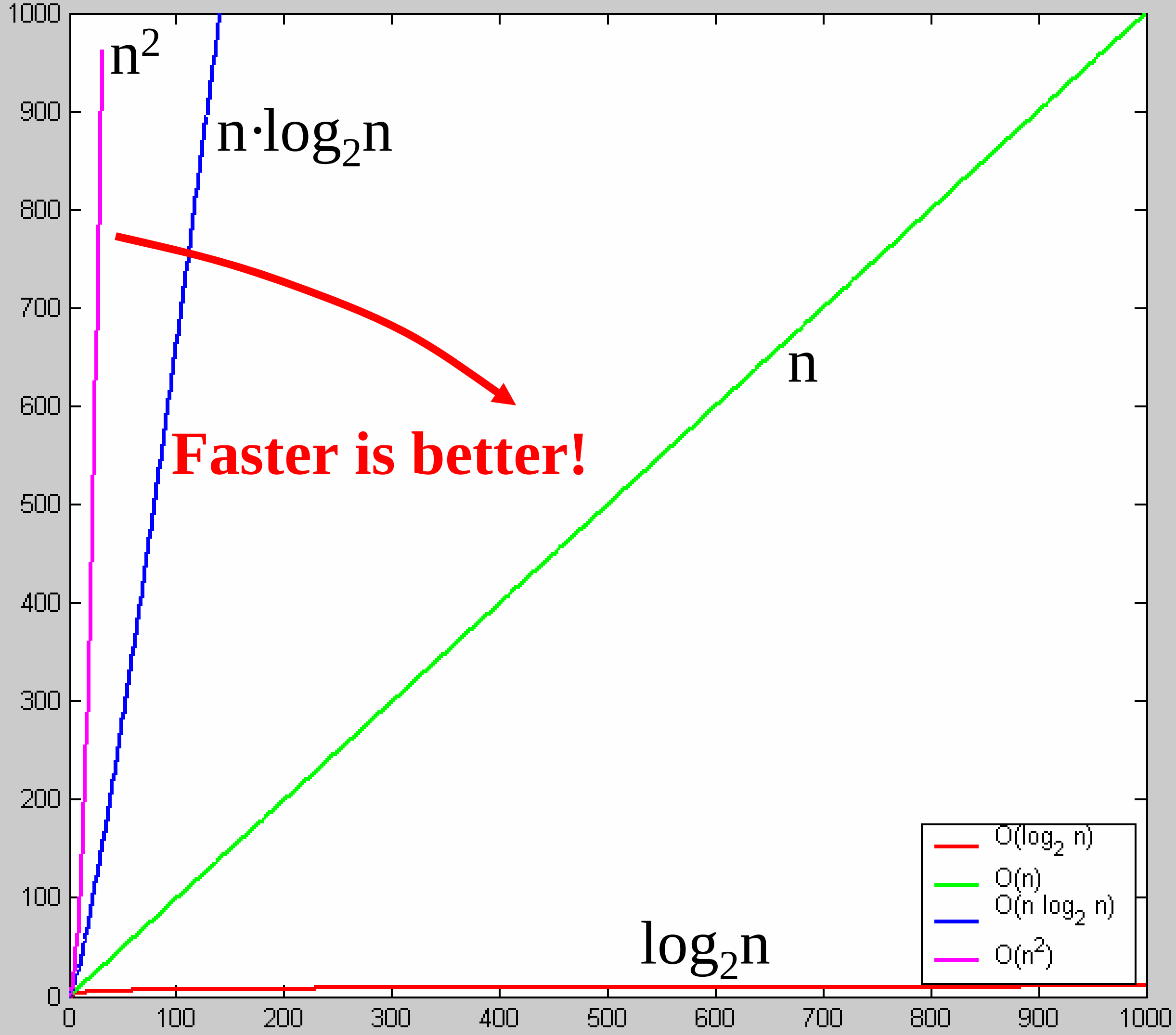
- The comparison function must provide a consistent *ordering* on the set of possible keys
  - You can compare any two keys and get back an indication of a < b, a > b, or a = b
  - The comparison functions must be consistent
    - If **`compare(a,b)`** says a<b, then **`compare(b,a)`** must say b>a
    - If **`compare(a,b)`** says a=b, then **`compare(b,a)`** must say b=a
    - If **`compare(a,b)`** says a=b, then **`equals(a,b) and equals(b,a)`** must say a=b

Why Sort?

- Sorting algorithms are among the most frequently used algorithms in computer science
- Allows binary search of an N-element array in O(log N) time
- Allows O(1) time access to *k*th largest element in the array for any *k*
- Allows easy detection of any duplicates

Time

- How fast is the algorithm?
  - The definition of a sorted array A says that for any i<j, A[i] < A[j]
  - This means that you need to at least check on each element at the very minimum, I.e., at least O(N)
  - And you could end up checking each element against every other element, which is $O(N^2)$
  - The big question is: How close to O(N) can you get?


n^2
$n \cdot \log_2 n$
Faster is better!
n
$\log_2 n$
$O(\log_2 n)$
$O(n)$
$O(n \log_2 n)$
$O(n^2)$
1000
900
800
700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000

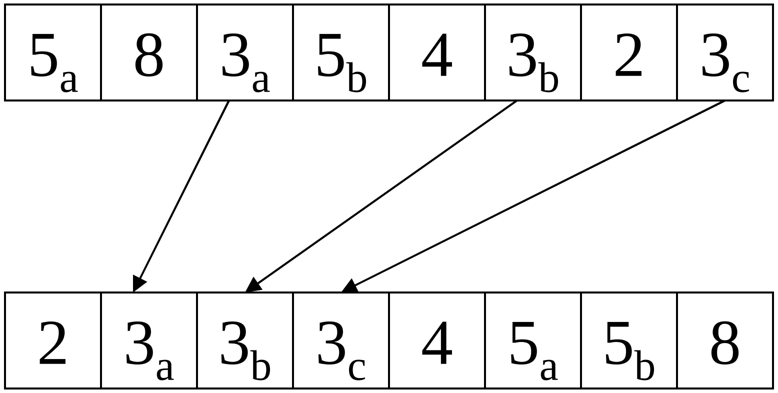
Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed? O(n) additional space
  - In-place sorting – no copying – O(1) additional space
  - Somewhere in between for “temporary”, e.g. O(logn) space
  - External memory sorting – data so large that does not fit in memory

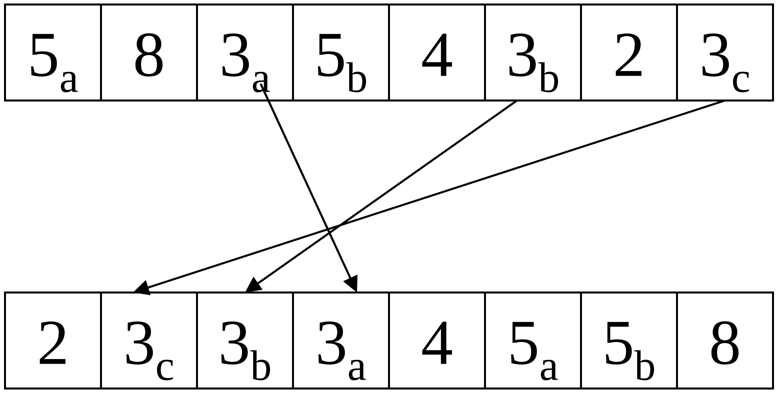
Stability

- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
    - › E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
    - › Extremely important property for databases
    - › A stable sorting algorithm is one which does not rearrange the order of duplicate keys

Example



Stable Sort



Unstable Sort

Bubble Sort

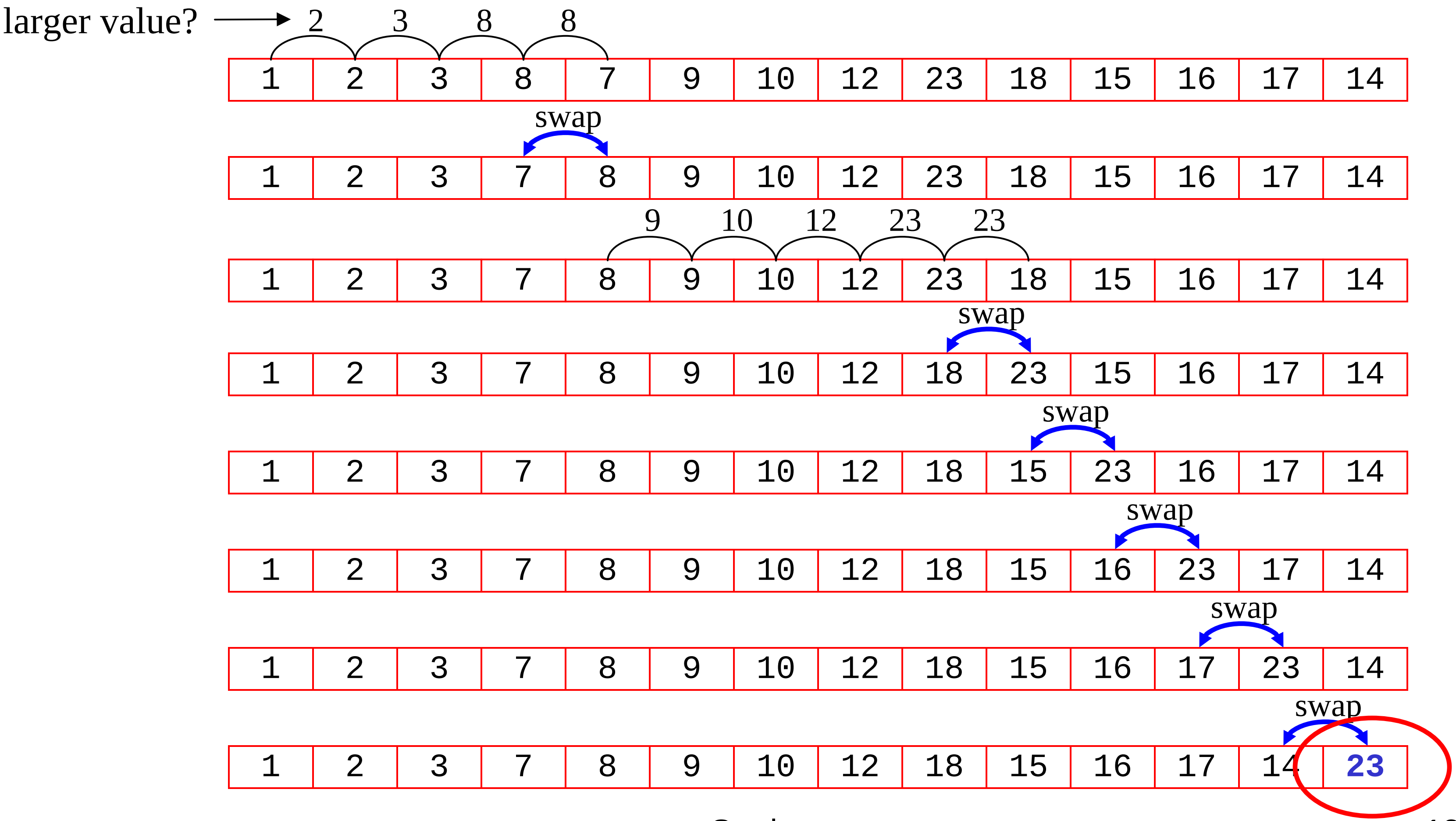
- “Bubble” elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
  - Bubble every element towards its correct position
    - last position has the largest element
    - then bubble every element except the last one towards its correct position
    - then repeat until done or until the end of the quarter, whichever comes first ...

Bubblesort

```
bubble(A[1..n]: integer array, n : integer): {
  i, j : integer;
  for i = 1 to n-1 do
    for j = 2 to n–i+1 do
       if A[j-1] > A[j] then SWAP(A[j-1],A[j]);
  }

SWAP(a,b) :  {
  t :integer;
  t:=a; a:=b; b:=t;
}
```

Put the largest element in its place



larger value? → 2 3 8 8

1 2 3 8 7 9 10 12 23 18 15 16 17 14

swap

1 2 3 7 8 9 10 12 23 18 15 16 17 14

9 10 12 23 23

1 2 3 7 8 9 10 12 23 18 15 16 17 14

swap

1 2 3 7 8 9 10 12 18 23 15 16 17 14

swap

1 2 3 7 8 9 10 12 18 15 23 16 17 14

swap

1 2 3 7 8 9 10 12 18 15 16 23 17 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 23 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 14 23

Bubble Sort: **Just Say No**

- “Bubble” elements to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
- We bubblize for i=1 to n (i.e, n times)
- Each bubblization is a loop that makes n-i comparisons
- This is $O(n^2)$

Insertion Sort

- What if first *k* elements of array are already sorted?
  - 4, 7, 12, 5, 19, 16
- We can shift the tail of the sorted elements list down and then *insert* next element into proper position and we get k+1 sorted elements
  - 4, 5, 7, 12, 19, 16

Insertion Sort

```
InsertionSort(A[1..N]: integer array, N: integer) {
  i, j, temp: integer ;
  for i = 2 to N {
    temp := A[i];
    j := i-1;
    while j > 1 and A[j-1] > temp {
          A[j] := A[j-1]; j := j–1;}
    A[j] = temp;
  }
}
```

- Is Insertion sort in place? Stable? Running time = ?

Insertion Sort Characteristics

- In place and Stable
- Running time
  - Worst case is $O(N^2)$
    - reverse order input
    - must copy every element every time
- Good sorting algorithm for almost sorted data
  - Each item is close to where it belongs in sorted order.

Inversions

- An inversion is a pair of elements in wrong order
  - › i < j but A[i] > A[j]
- By definition, a sorted array has no inversions
- So you can think of sorting as the process of removing inversions in the order of the elements

Inversions

- Our simple sorting algorithms so far swap adjacent elements (explicitly or implicitly) and remove just 1 inversion at a time
  - Their running time is proportional to number of inversions in array
- Given N distinct keys, the maximum possible number of inversions is

$$(n-1)+(n-2)+\ldots+1=\sum_{i=1}^{n-1} i=\frac{(n-1)n}{2}$$

Inversions and Adjacent Swap Sorts

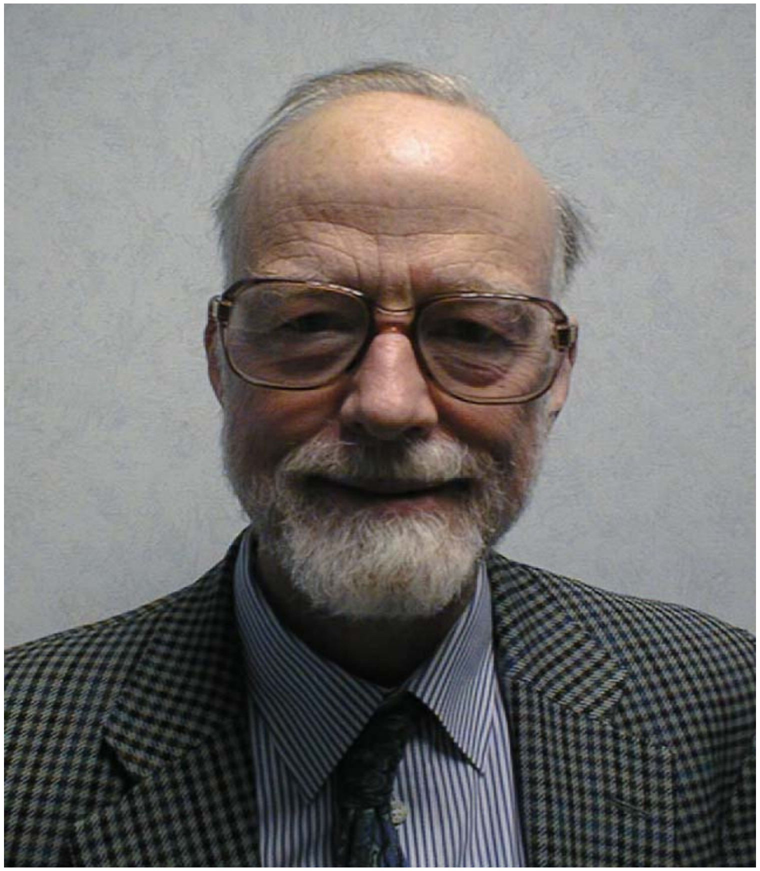
- "Average" list will contain half the max number of inversions = $\frac{(n-1)n}{4}$
  - So the average running time of Insertion sort is $O(N^2)$
- Any sorting algorithm that only swaps adjacent elements requires $O(N^2)$ time because each swap removes only one inversion (lower bound)

“Divide and Conquer” Sorting algorithms

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
- **Idea 2 :** Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Quicksort (1962)

- Due to Sir Tony Hoare (1934-)



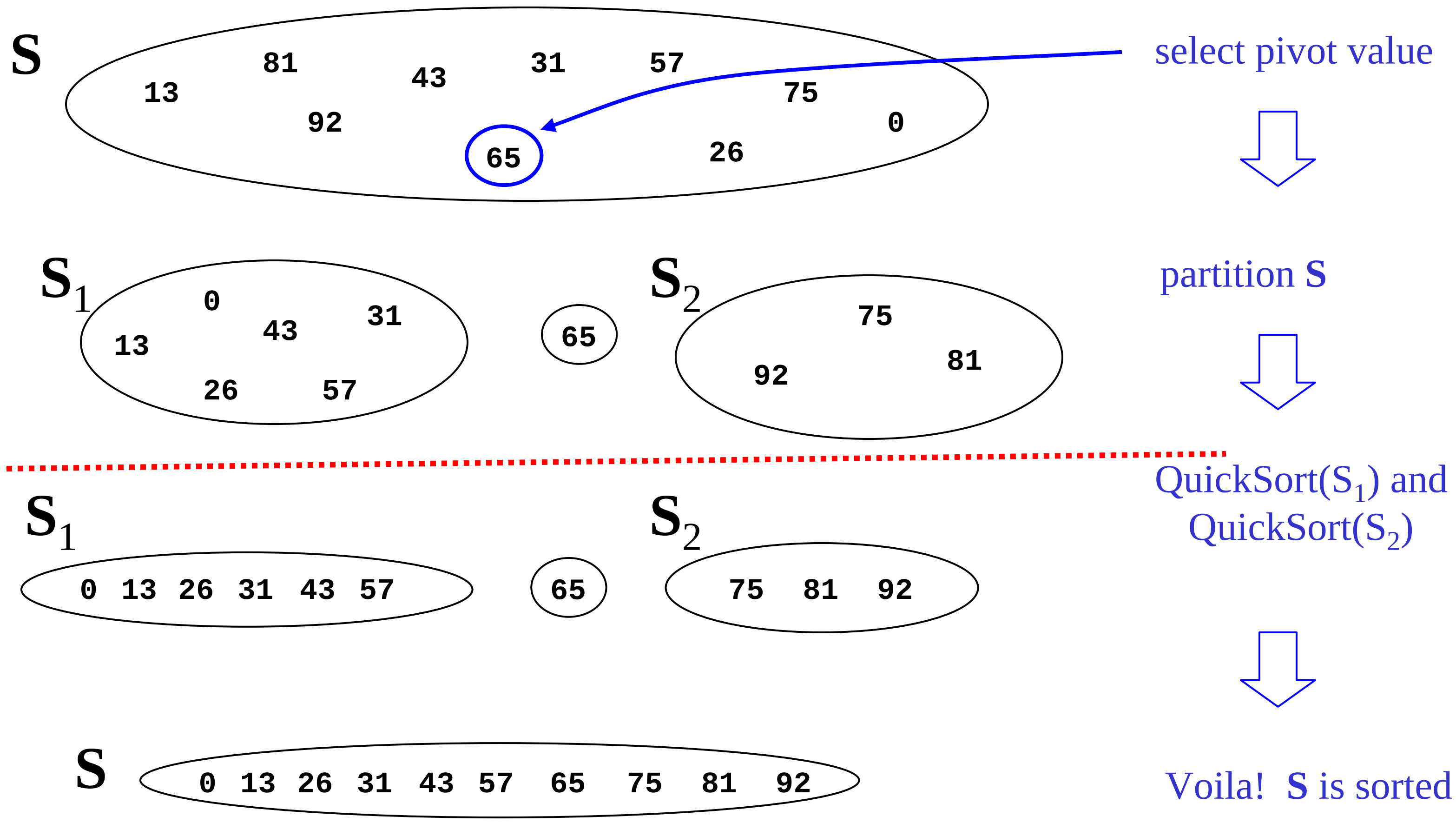
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - › Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - › Recursively sort left and right sub-arrays
  - › The elements “remain” in the array

“Four easy steps”

- To sort an array **S**
    1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
    2. Pick an element *v* in **S**. This is the *pivot* value.
    3. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1 = \{\text{all values } x \leq v\}$, and $\mathbf{S}_2 = \{\text{all values } x \geq v\}$.
    4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



S

13 81 92 43 65 31 57 75 26 0

select pivot value

S_1

0 13 43 31 26 57

65

S_2

75 92 81

partition **S**

QuickSort(S_1) and
QuickSort(S_2)

S_1

0 13 26 31 43 57

65

S_2

75 81 92

S

0 13 26 31 43 57 65 75 81 92

Voila! **S** is sorted

Details, details

- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Choosing the order of the recursive calls

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are ≤pivot
  - elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning:Choosing the pivot

- One implementation (there are others) median3
  - Median3 takes the median of leftmost, middle, and rightmost elements
  - An alternative is to choose the pivot randomly
  - Another alternative is to choose the first element (but can be very bad. Why?)

Median 3

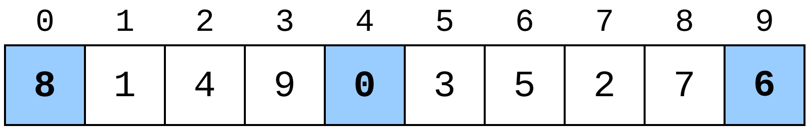
- Find median, min and max of A[left], A[right] and A[(left+right)/2]
- A[left] = min
- A[right] = max
- A[right-1] = median (called pivot)

Partitioning in-place

- Set pointers i and j to start and end of array except for pivot and last element
- Increment i until you hit element A[i] > pivot
  - “while A[i] < pivot then i++”
- Decrement j until you hit element A[j] < pivot
  - “while A[j] > pivot then j- -”
- Swap A[i] and A[j]
  - “if i< j then swap(A,i,j)”
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]

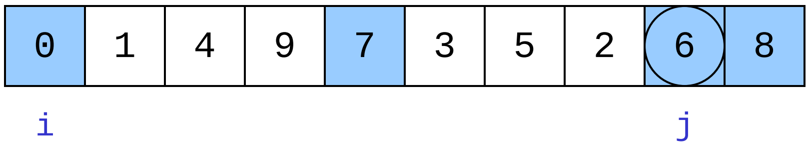
Example

Choose the pivot as the median of three



0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

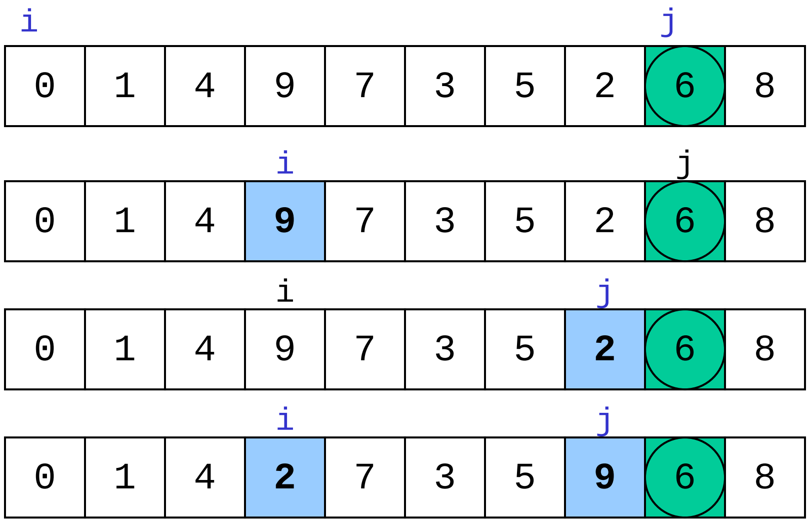
Median of 0, 6, 8 is 6. Pivot is 6



0	1	4	9	7	3	5	2	6	8
i								j	

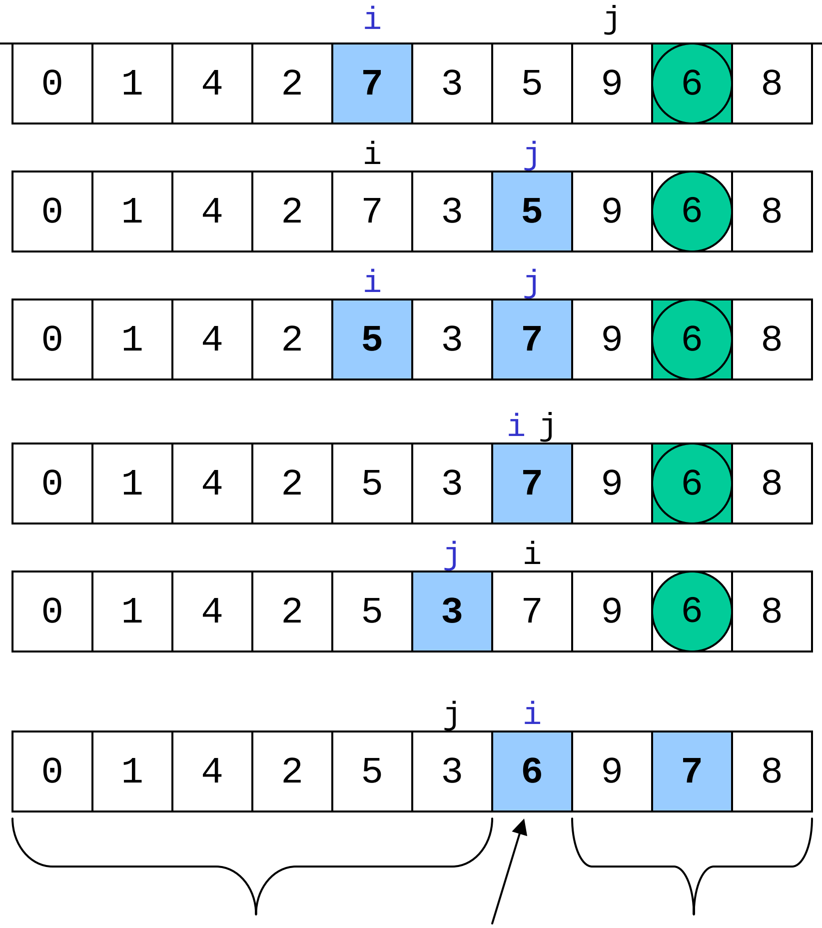
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example



Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example



Cross-over i > j

S_1 < pivot

pivot

S_2 > pivot

Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don’t use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - › T(0) = T(1) = O(1)
    - constant time if 0 or 1 element
  - › For N > 1, 2 recursive calls plus linear time for partitioning
  - › T(N) = 2T(N/2) + O(N)
    - Same recurrence relation as Mergesort
  - › T(N) = O(N log N)

Analysis Upper Bound

$$
\begin{aligned}
T(n) &\le 2T(n/2) + dn \qquad \text{Assuming } n \text{ is a power of } 2 \\
&\le 2(2T(n/4) + dn/2) + dn \\
&= 4T(n/4) + 2dn \\
&\le 4(2T(n/8) + dn/4) + 2dn \\
&= 8T(n/8) + 3dn \\
&\vdots \\
&\le 2^k T(n/2^k) + kdn \\
&= nT(1) + kdn \qquad \text{if } n = 2^k \\
&\le cn + dn \log_2 n \\
&= O(n \log n)
\end{aligned}
$$

$n = 2^k$, $k = \log n$

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) \leq T(N-1) + bN$
  - $\leq T(N-2) + b(N-1) + bN$
  - $\leq T(2) + b(3) + \ldots + bN$
  - $\leq T(1) + b(2 + 3 + \ldots + N)$
  - $T(N) = O(N^2)$
- Fortunately, *average case performance* is O(N log N) (not a simple analysis)

Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call (O(logn) space).
  - Choose smallest partition first in the recursion
- O(n log n) average case performance, but O(n^2) worst case performance.

Folklore

- “Quicksort is the best in-memory sorting algorithm.”
- Truth
  - › Quicksort uses very few comparisons on average.
  - › Quicksort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality