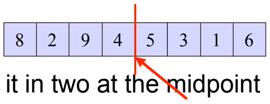
# “Divide and Conquer”

- 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 → known as Mergesort
- **Idea 2 :** Partition array into small items and large items, then recursively sort the two sets → known as Quicksort

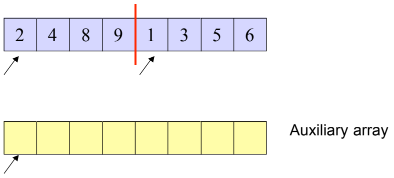
# Mergesort



- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

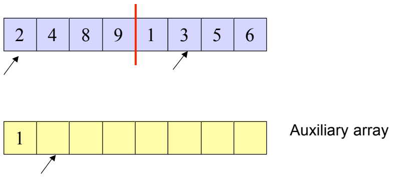
# Auxiliary Array

- The merging requires an auxiliary array.



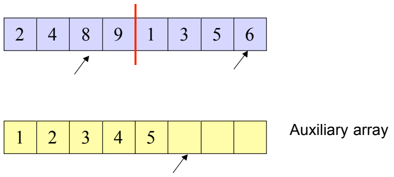
# Auxiliary Array

- The merging requires an auxiliary array.



# Auxiliary Array

- The merging requires an auxiliary array.



# Recursive Mergesort

```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```

## Recurring Student Activity

Merge Sort 31 16 54 4 2 17 6

## Merge Sort: Complexity

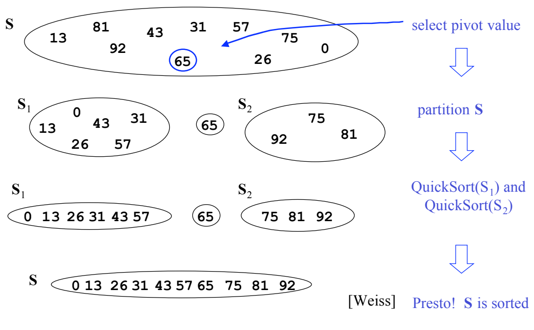
## Quicksort

- 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
        - 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
    - › Concatenate left and right sub-arrays in O(1) time

## "Four easy steps"

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

## The steps of QuickSort



select pivot value

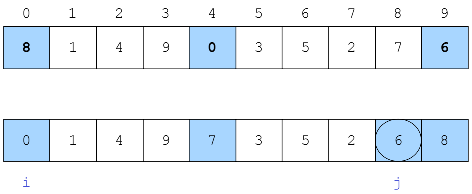
partition S

QuickSort($S_1$) and QuickSort($S_2$)

Presto! S is sorted

[Weiss]

## QuickSort Example



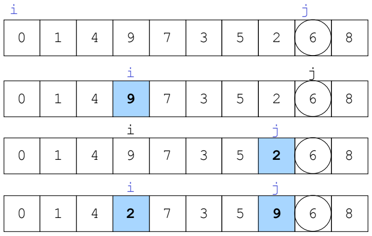
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **8** | 1 | 4 | 9 | **0** | 3 | 5 | 2 | 7 | **6** |

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

•Choose the pivot as the median of three.

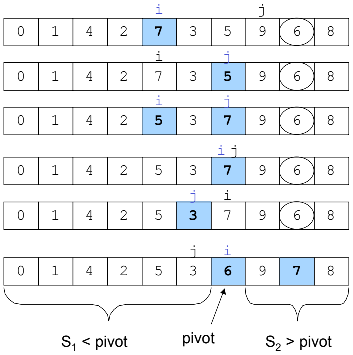
•Place the pivot and the largest at the right and the smallest at the left

## QuickSort Example



- •Move i to the right to be larger than pivot.
- •Move j to the left to be smaller than pivot.
- •Swap

## QuickSort Example



## 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.

## Recurring Student Activity

Quick Sort 31 16 54 4 2 17 6

## QuickSort:
## Best case complexity

## QuickSort:
## Worst case complexity

## QuickSort:
## Average case complexity

Turns out to be O($n$ log $n$)

See Section 7.7.5 for an idea of the proof.
*Don't need to know proof details for this course.*

## Features of Sorting Algorithms

- In-place
  - Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)
- Stable
  - Items in input with the same value end up in the same order as when they began.

Student Activity

## Sort Properties

| Are the following: | **stable?** | | | **in-place?** | |
|---|---|---|---|---|---|
| Insertion Sort? | No | Yes | Can Be | No | Yes |
| Selection Sort? | No | Yes | Can Be | No | Yes |
| MergeSort? | No | Yes | Can Be | No | Yes |
| QuickSort? | No | Yes | Can Be | No | Yes |

## How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if the basic action is a comparison.

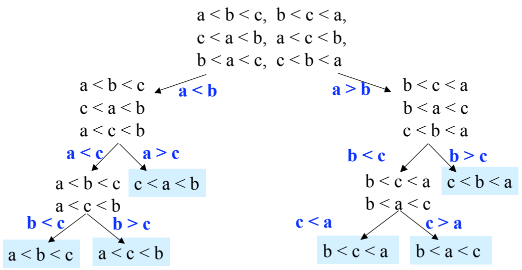
## Sorting Model

- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)

## Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = 3·2·1 = 3! (ie, "3 factorial")
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - N(N-1)(N-2)…2)(1)= N! possible orderings

# Decision Tree



a < b < c, b < c < a,
c < a < b, a < c < b,
b < a < c, c < b < a

**a < b**

a < b < c
c < a < b
a < c < b

**a < c**

a < b < c
a < c < b

**b < c**

a < b < c

**b > c**

a < c < b

**a > c**

c < a < b

**a > b**

b < c < a
b < a < c
c < b < a

**b < c**

b < c < a
b < a < c

**c < a**

b < c < a

**c > a**

b < a < c

**b > c**

c < b < a

The leaves contain all the possible orderings of a, b, c

# Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - ie, the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

# Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - ie, by making comparisons
  - Each decision reduces the possible solution space by one half
- Run time is ≥ maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

Student Activity

# Lower bound on Height

- A binary tree of height h has **at most** *how many* leaves?

L

- A binary tree with L leaves has height **at least**:

h

- The decision tree has how many leaves:
- So the decision tree has height:

h

# log(*N*!) is Ω(*N*log*N*)

$$\log(N!) = \log\big(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\big)$$

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$

select just the first N/2 terms

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2}$$

each of the selected terms is ≥ logN/2

$$\geq \frac{N}{2} \log \frac{N}{2}$$

$$\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2}$$

$$= \Omega(N \log N)$$

# Ω(N log N)

- Run time of any comparison-based sorting algorithm is **Ω(N log N)**
- Can we do better if we don’t use comparisons?

# BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and *K*, create an array `count` of size *K*, **increment** counts while traversing the input, and finally output the result.

**Example** *K*=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

| `count` array | |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

**Running time to sort n items?**

# BucketSort Complexity: O(*n*+*K*)

- Case 1: *K* is a constant
  - BinSort is linear time
- Case 2: *K* is variable
  - Not simply linear time
- Case 3: *K* is constant but large (e.g. $2^{32}$)
  - ???

# Fixing impracticality: RadixSort

- Radix = "The base of a number system"
  - We'll use 10 for convenience, but could be anything
- Idea: BucketSort on each **digit**, least significant to most significant (lsd to msd)

# Radix Sort Example (1st pass)

Input data

478
537
9
721
3
38
123
67

Bucket sort by 1's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 721 | | 3<br>123 | | | | 537<br>67 | 478<br>38 | 9 |

After 1st pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

# Radix Sort Example (2nd pass)

After 1st pass

721
3
123
537
67
478
38
9

Bucket sort by 10's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 03<br>09 | | 721<br>123 | 537<br>38 | | | 67 | 478 | | |

After 2nd pass

3
9
721
123
537
38
67
478

# Radix Sort Example (3rd pass)

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 003<br>009<br>038<br>067 | 123 | | | 478 | 537 | | 721 | | |

**Invariant**: after k passes the low order k digits are sorted.

After 3rd pass

3
9
38
67
123
478
537
721

Student Activity

# RadixSort

- Input:126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

BucketSort on next-higher digit:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

BucketSort on msd:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

---

# Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?
- In practice
  - RadixSort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort

---

# Internal versus External Sorting

- So far assumed that accessing A[i] is fast – Array A is stored in internal memory (RAM)
  - Algorithms so far are good for internal sorting
- What if A is so large that it doesn't fit in internal memory?
  - Data on disk or tape
  - Delay in accessing A[i] – e.g. need to spin disk and move head

---

# Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - Load chunk of data into RAM, sort, store this "run" on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
  - Text gives some examples

---

# Summary of Sorting

- Sorting choices:
  - O($N^2$) – Bubblesort, Insertion Sort
  - O(N log N) average case running time:
    - Heapsort: In-place, not stable
    - Mergesort: O(N) extra space, stable.
    - Quicksort: claimed fastest in practice but, O($N^2$) worst case. Needs extra storage for recursion. Not stable.
  - O(N) – Radix Sort: fast and stable. Not comparison based. Not in-place.