Sorting Algorithms

November 15, 2023

Sorting

- Sorting is a process that organizes a collection of data into either *ascending* or *descending* order.
- An internal sort requires that the collection of data fit entirely in the computer's main memory.
- We can use an external sort when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk (or tape).
- We will analyze only internal sorting algorithms.

Why Sorting?

- Significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use sorting somewhere, and in many cases, the sorting cost determines the running time.
- Applications:
  - Closest pair – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them?
  - Frequency distribution – Given a set of n items, which element occurs the largest number of times in the set?
  - Selection – What is the kth largest item in an array?, etc.

Take-Home Lesson

Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.

Preliminaries

- We will discuss the problem of sorting an array of elements.
- We assume all array positions contains data to be sorted.
- To simply matters, we will assume in the examples that the array contains integers. But any comparable type can be used.
- For general sorting algorithms, besides the assignment operator, we will assume existence of the ">" and "<" operators which can be used to place a consistent ordering on the input. Sorting under these conditions is known as "comparison-based sorting".
- A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

Sorting Algorithms

- There are many sorting algorithms, such as:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Shell Sort
  - Heap Sort
  - Merge Sort
  - Quick Sort
- The first three are the foundations for more efficient algorithms.

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - The most common sorting technique used by card players.
- The list is divided into two parts: *sorted* and *unsorted*.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of n elements will take at most $n - 1$ passes to sort the data.

Insertion Sort Example

Sorted **Unsorted**

23	78	45	8	32	56	**Original List**
23	78	45	8	32	56	**After pass 1**
23	45	78	8	32	56	**After pass 2**
8	23	45	78	32	56	**After pass 3**
8	23	32	45	78	56	**After pass 4**
8	23	32	45	56	78	**After pass 5**

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 Algorithm

```
// Simple insertion sort for integer arrays
void insertionSort( int [] a )
{
    for( int p = 1; p < a.length; p++ )
    {
        int tmp = a[p];

        int j;
        for( j = p; j > 0 && tmp < a[j-1]; j-- )
            a[j] = a[j-1];
        a[j] = tmp;
    }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- Best-case: $\rightarrow O(n)$
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: $2 \times (n-1) \rightarrow O(n)$
  - The number of key comparisons: $(n-1) \rightarrow O(n)$
- Worst-case: $\rightarrow O(n^2)$
  - Array is in reverse order:
  - Inner loop is executed $i-1$ times, for $i = 2, 3, \ldots, n$
  - The number of moves:
    $2 \times (n-1) + (1 + 2 + \cdots + n-1) = 2 \times (n-1) + n \times (n-1)/2 \rightarrow O(n^2)$
  - The number of key comparisons:
    $(1 + 2 + \cdots + n-1) = n \times (n-1)/2 \rightarrow O(n^2)$
- Average-case: $\rightarrow O(n^2)$
  - We have to look at all possible initial data organizations.
- So, Insertion Sort is $O(n^2)$

Analysis of insertion sort

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?
- Worst:
  - Longest running time (this is the upper limit for the algorithm)
  - It is guaranteed that the algorithm will not be worse than this.
- Sometimes we are interested in the average case. But there are some problems with the average case.
  - It is difficult to figure out the average case. i.e. what is average input?
  - Are we going to assume all possible inputs are equally likely?
  - In fact, for most sorting algorithms the average case is the same as the worst case.

Other simple sorting algorithms

Selection Sort

- Scans from left to right
- In iteration i, find index min of smallest remaining entry.
- Swap a[i] and a[min].

- arr[] = 64, 25, 12, 22, 11
- **1st pass: 11** 25 12 22 64
- **2nd pass:** 11 **12** 25 22 64
- **3rd pass:** 11 12 **22** 25 64
- **4th pass:** 11 12 22 **25** 64
- **5th pass:** 11 12 22 25 **64**

Other simple sorting algorithms

Bubble Sort

- repeatedly swap the adjacent elements if they are in the wrong order.

First Pass:

- (5 1 4 2 8) –> (1 5 4 2 8), Swaps since 5 > 1
- (1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
- (1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
- (1 4 2 5 8) –> (1 4 2 5 8), since these elements are already in order (8 > 5), algorithm does not swap them.

2nd pass:

- (1 4 2 5 8) -> (1 4 2 5 8)
- (1 4 2 5 8) -> (1 2 4 5 8)
- (1 2 4 5 8) -> (1 2 4 5 8)
- (1 2 4 5 8) -> (1 2 4 5 8)

Will do a third pass since a swap took place in 2nd pass, but the array is sorted!

A lower bound for simple sorting algorithms

- **An inversion :** an ordered pair (A_i, A_j) such that $i < j$ but $A_i > A_j$
- **Example:** 10, 6, 7, 15, 3,1 Inversions are: (10,6), (10,7), (10,3), (10,1), (6,3), (6,1) (7,3), (7,1) (15,3), (15,1), (3,1)

Swapping

- Swapping adjacent elements that are out of order removes one inversion.
- A sorted array has no inversions.
- Sorting an array that contains i inversions requires at least i swaps of adjacent elements.

Theorems

- **Theorem 1:** The average number of inversions in an array of N distinct elements is $N(N-1)/4$
- **Theorem 2:** Any algorithm that sorts by swaping adjacent elements requires $\Omega(N^2)$ time on average.
- For a sorting algorithm to run in less than quadratic time it must do something other than swaping adjacent elements.

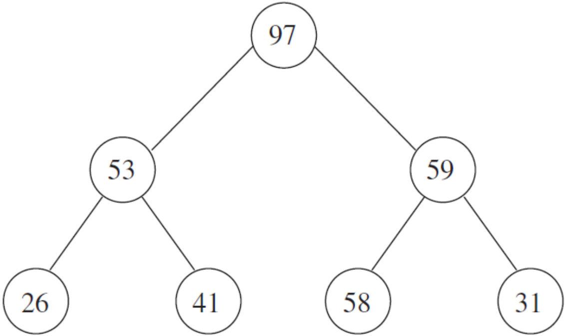
Heapsort

- The priority queue can be used to sort N items by
  1. inserting every item into a binary heap OR calling `buildHeap`
  2. extracting every item by calling `deleteMin` N times
- An algorithm based on this idea is heapsort.
- It is an $O(N \log N)$ worst-case sorting algorithm.

Heapsort

- The main problem with this algorithm is that it uses an extra array for the items exiting the heap.
- We can avoid this problem as follows:
  - After each `deleteMin`, the heap shrinks by 1.
  - Thus the cell that was last in the heap can be used to store the element that was just deleted.
  - Using this strategy, after the last `deleteMin`, the array will contain all elements in decreasing order.
- If we want them in increasing order we must use a max heap.

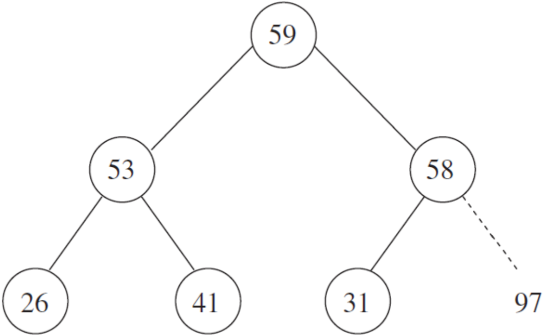
Heapsort Example



	97	53	59	26	41	58	31			
0	1	2	3	4	5	6	7	8	9	10

Max heap after the `buildHeap` phase

Heapsort Example (Cont.)



0	1	2	3	4	5	6	7	8	9	10
	59	53	58	26	41	31	97			

Heap after the first `deleteMax` operation

Implementation

- In the implementation of heapsort, an array is utilized as in Binary Heap implementation.
- There are some minor changes in the code compared to Binary Heap ADT:
  - Since we use max heap, the comparisons logic is changed from > to <.
  - Percolating down needs the current heap size which is lowered by 1 at every deletion.

The Heapsort Sort Algorithm

```
// Standard heapsort.

void heapsort( int [] a )
{
    for( int i = a.length / 2 - 1; i >= 0; i-- ) // buildHeap
        percDown( a, i, a.length );
    for( int j = a.length - 1; j > 0; j-- )
    {
        swapReferences( a, 0, j ); // deleteMax
        percDown( a, 0, j );
    }
}
```

percDown Algorithm

```
// Internal method for heapsort.
// i is the index of an item in the heap.
// Returns the index of the left child.

int leftChild( int i )
{
    return 2 * i + 1;
}
// Internal method for heapsort that is used in
// deleteMax and buildHeap.
// i is the position from which to percolate down.
// n is the logical size of the binary heap.

void percDown( int [] a, int i, int n )
{
    int child;
    int tmp;

    for( tmp =  a[ i ] ; leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child != n-1 && a[ child ] < a[ child+1 ] )
                child++;
        if( tmp < a[ child ] )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}
```

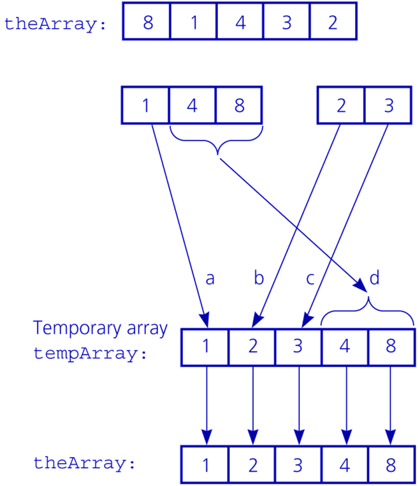
Analysis of Heapsort

- It is an $O(NlogN)$ algorithm.
  - First phase: Build heap $O(N)$
  - Second phase: N deleteMax operations: $O(N \log N)$.
- Detailed analysis shows that, the average case for heapsort is poorer than quick sort.
  - Quicksort's worst case however is far worse.
- An average case analysis of heapsort is very complicated, but empirical studies show that there is little difference between the average and worst cases.
  - Heapsort usually takes about twice as long as quicksort.
  - Heapsort therefore should be regarded as something of an insurance policy.
  - On average, it is more costly, but it avoids the possibility of $O(N^2)$.

Mergesort

- Mergesort algorithm is one of the two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sorts each half separately, and
  - Then merges the sorted halves into one sorted array.

Merge Sort Example



Divide the array in half

Sort the halves

Merge the halves:

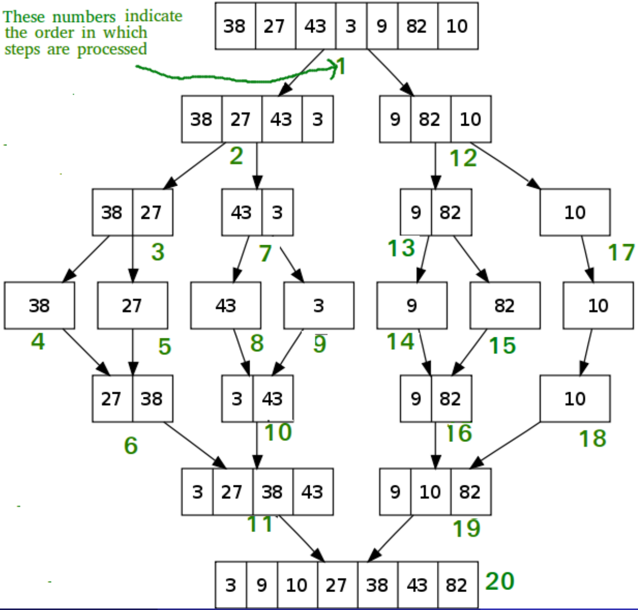
a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

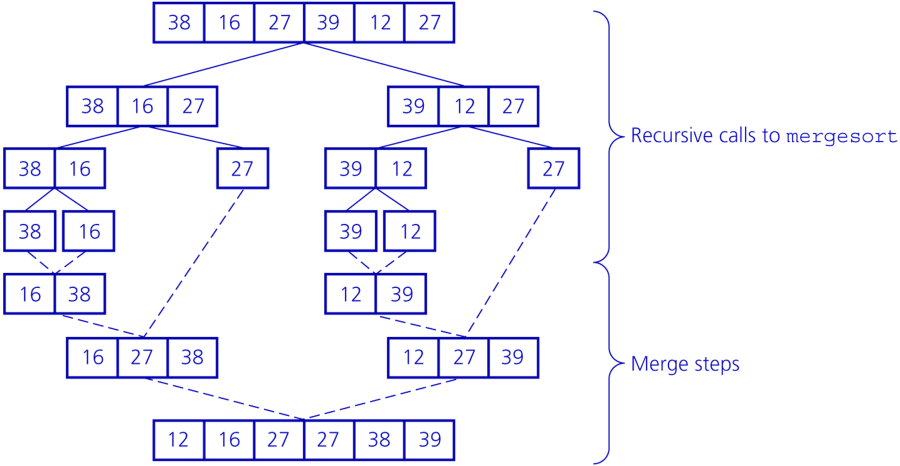
Merge Sort Algorithm

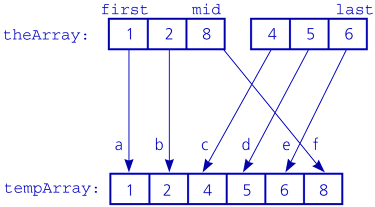
```
mergesort(item_type A[])
{
  n<- length(A), if n<2 return
  middle = n/2
  left: array of size(middle)
  right: array of size (n-middle)
  for i=0:middle-1
        left[i]<-A[i]
  for i=middle:n
        right[i-middle] <- A[i]
  mergesort(left);
  mergesort(right);
  merge(left, right, A);
}

merge(Left, Right, A )
{
  nL <- length(Left)  //nr. of elements on the left
  nR <- length(Right) //nr. of elements on the right
  i,j,k <- 0
  while (i<nL && j<nR) //loop over the left and right
  {
        if(Left[i] < Right[j])
          A[k] <-Left[i], k++, i++ //advance left
        else
          A[k] <-Right[j], k++, j++ //advance right
  }
  while (i<nL) //Copy the remaining
        A[k] <-Left[i], k++, i++
  while (j<nR)
   A[k] <-Right[j], k++, j++
}
```

Merge Sort Example



Merge Sort Example



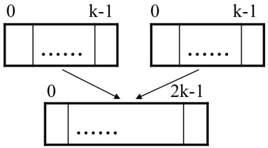


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

A worst-case instance of the merge step in mergesort

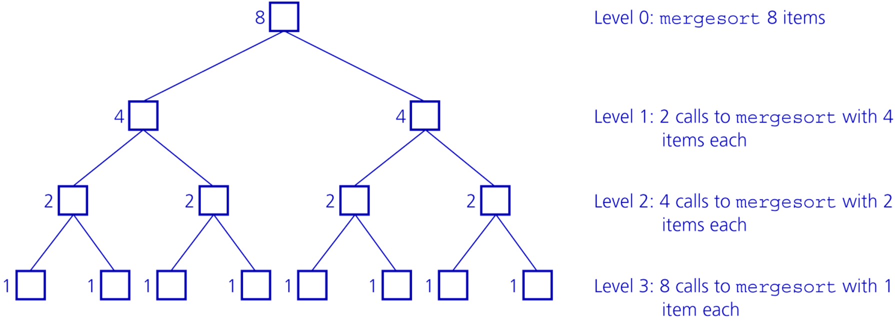
Mergesort - Analysis of Merge (cont.)



Merging two sorted arrays of size k

- Best-case:
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: $2k + 2k$
  - The number of key comparisons: k
- Worst-case:
  - The number of moves: $2k + 2k$
  - The number of key comparisons: $2k - 1$

Mergesort - Analysis



Levels of recursive calls to mergesort, given an array of eight items

Mergesort - Analysis

- Assume that N is a power of 2, split into two even halves, for N=1 mergesort is constant, 1
- Otherwise, the time to mergesort N numbers is equal to the time to do 2 recursive mergesort of size N/2, plus time to merge, which is linear
  $T(1) = 1$, $T(N) = 2T(N/2) + N$
- Let us divide both sides by N
  $T(N)/N = T(N/2)/(N/2) + 1$
- valid for any N that is a power of 2
  $T(N/2)/(N/2) = T(N/4)/(N/4) + 1$ and
  $T(N/4)/(N/4) = T(N/8)/(N/8) + 1$ ...
  $T(2)/(2) = T(1)/(1) + 1$
- Now add up all the equations. Observe that T(N/2)/(N/2) appear on both sides and cancel. In fact all terms appear on both sides and cancel. This is called *telescoping sum*:
  $T(N)/(N) = T(1)/(1) + logN$ (there are logN equations and all 1's add up to logN. Multiplying by N
  $T(N) = NlogN + N = O(NlogN)$

Mergesort - Analysis

- Mergesort is an extremely efficient algorithm with respect to time.
  - Both worst case and average cases are $O(n \times \log n)$
- Running time depends heavily on relative costs of comparing and moving elements in the array and the temporary array. These costs are language dependent.
- Mergesort requires an extra array whose size equals to the size of the original array.
- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half ($O(n)$)

Mergesort for Linked Lists

- Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.
- MergeSort
  1. If head is NULL or there is only one element in the Linked List then return.
  2. Else divide the linked list into two halves.
  3. Sort the two halves a and b.
  4. Merge the two parts of the list into a sorted one.

Quicksort

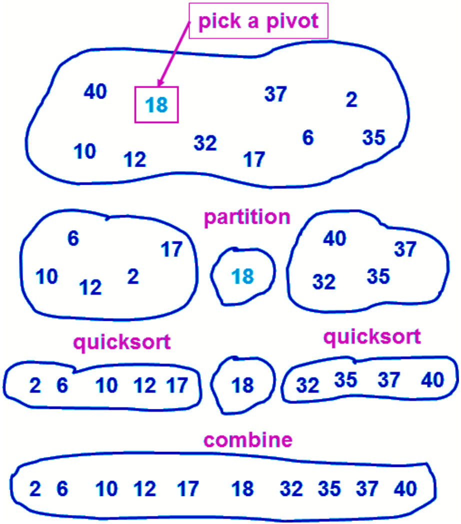
- Like mergesort, Quicksort is also based on the divide-and-conquer paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done before the recursive calls.
- Basic idea:
  1. First, arbitrarily choose an item and partition the array into three groups: those smaller than the chosen item, those equal to the chosen item and those larger than the chosen item,
  2. Then, sort the first and last groups recursively,
  3. Finally, concatenate three groups.

Quicksort

Algorithm 1 Quicksort

```
Let S be the input set.
if | S| = 0 or | S| = 1 then
        return
Pick an element v in S. Call v the pivot.
Partition S − {v} into two disjoint groups:
        S1 = {x ∈ S − {v} |x < v}
        S2 = {x ∈ S − {v} |x ≥ v}
        return { quicksort(S1), v, quicksort(S2) }
```

Quicksort Illustrated



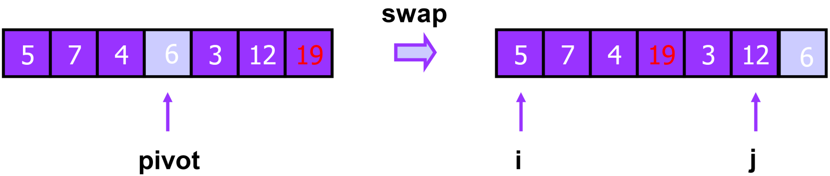
Issues To Consider

- How to pick the pivot?
  - Many methods (discussed later)
- How to partition?
  - Several methods exist.
  - The one we consider is known to give good results and to be easy and efficient.

We discuss the partition strategy first.

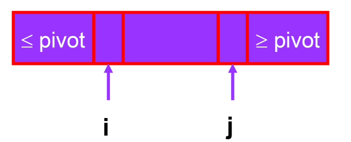
Partitioning Strategy

- For now, assume that pivot = A[(left+right)/2].
- We want to partition array A[left .. right].
- First, get the pivot element out of the way by swapping it with the last element (swap pivot and A[right]).
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)

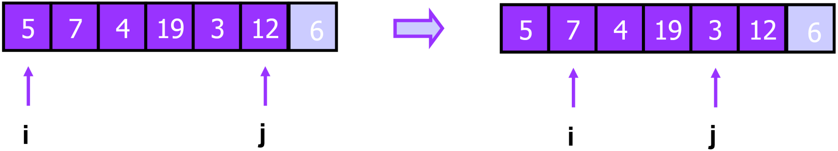


Partitioning Strategy (Cont.)

- Want to have
  - A[k] $\leq$ pivot, for k < i
  - A[k] $\geq$ pivot, for k > j

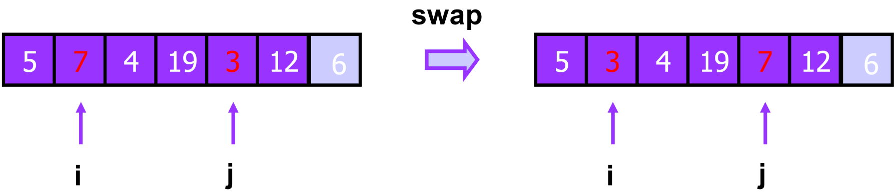


- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] $\geq$ pivot (*i is pointing to a larger element*)
    - A[j] $\leq$ pivot (*j is pointing to a smaller element*)
    - $\Rightarrow$ A[i] and A[j] should now be swapped



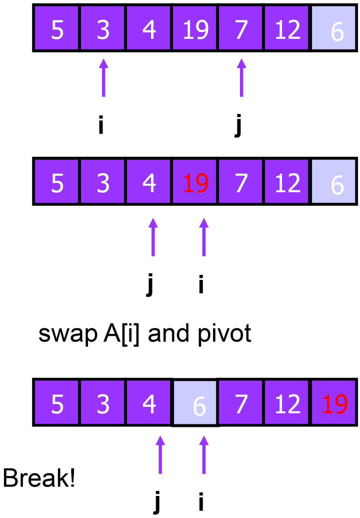
Partitioning Strategy (Cont.)

- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] $\leq$ pivot
    - A[j] $\geq$ pivot
  - Repeat the process until i and j cross



Partitioning Strategy (Cont.)

- When i and j have crossed
  - swap A[i] and pivot
- Result:
  - A[k] $\leq$ pivot, for k < i
  - A[k] $\geq$ pivot, for k > i



Pivot Strategies

- First element:
  - Bad choice if input is sorted or in reverse sorted order
  - Bad choice if input is nearly sorted
- Random:
  - Perfectly safe, unless the random number generator has a flaw
  - Random number generation may be costly
- Median-of-three:
  - The best choice would be the median of the array but not feasible.
  - A good estimate is to choose the median of the left, right, and center elements

Quicksort algorithm

It will be implemented in PS

Analysis of Quicksort

- Worst case: pivot is the smallest (or largest) element all the time.
  $T(N) = T(N-1) + cN$
  $T(N-1) = T(N-2) + c(N-1)$
  $T(N-2) = T(N-3) + c(N-2)$
  $\dots$
  $T(2) = T(1) + c(2)$
  $T(N) = T(1) + c\sum_{i=2}^{N} i \rightarrow O(N^2)$
- Best case: pivot is the median
  $T(N) = 2T(N/2) + cN$
  $T(N) = cN\log N + N \rightarrow O(N \log N)$

Quicksort: Average case

- Assume each of the sizes for S_1 are equally likely.
- $0 \leq |S_1| \leq N - 1$.

$$T(N) = \left(\frac{1}{N} \sum_{i=0}^{N-1} [T(i) + T(N - i - 1)] \right) + cN$$
$$T(N) = \left(\frac{2}{N} \sum_{i=0}^{N-1} T(i) \right) + cN$$
$$NT(N) = \left(2 \sum_{i=0}^{N-1} T(i) \right) + cN^2$$
$$(N-1)T(N-1) = \left(2 \sum_{i=0}^{N-2} T(i) \right) + c(N-1)^2$$
$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$
$$NT(N) = (N+1)T(N-1) + 2cN$$

Quicksort: Average case (Cont.)

$NT(N) = (N+1)T(N-1) + 2cN$
Divide equation by $N(N+1)$

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$

Telescope;

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N}$$
$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1}$$
$$\ldots$$
$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{N+1}\frac{1}{i}$$

Quicksort: Average case (Cont.)

$2c \sum_{i=3}^{N+1} \frac{1}{i} = 2c(H_{N+1} - \frac{3}{2})$

$T(N) = (N+1)(\frac{T(1)}{2} + 2c(H_{N+1} - \frac{3}{2})$

$H_N \approx \log_e(N) + \gamma + \frac{1}{2N}$ ($\gamma = 0.577215664901$(Euler-Mascheroni Constant)

$T(N) \approx (N+1)\left[\frac{T(1)}{2} + 2c\left((\log_e(N+1) + \gamma + \frac{1}{2(N+1)}) - \frac{3}{2}\right)\right]$

$T(N) \rightarrow O(N \log N)$

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?

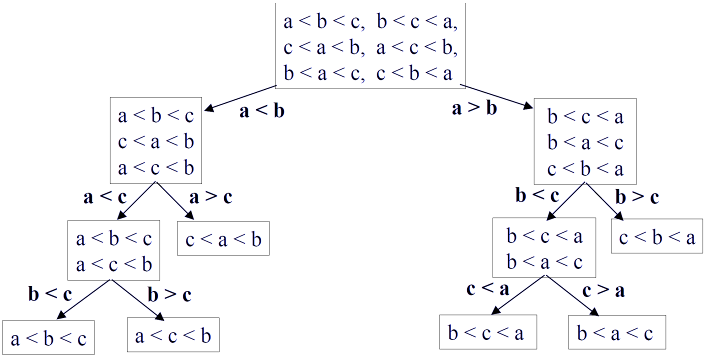
The Answer is No! (if using comparisons only)

- Our basic assumption: we can only compare two elements at a time – how does this limit the run time?
- Suppose you are given N elements
  - Assume no duplicates – any sorting algorithm must also work for this case
- How many possible orderings can you get?

How many possible orderings?

- Example: a, b, c ($N = 3$)
- Orderings:
  1. a b c
  2. b c a
  3. c a b
  4. a c b
  5. b a c
  6. c b a
- 6 orderings = $3 \times 2 \times 1 = 3!$
- For N elements: N! orderings

A Decision Tree



Leaves contain possible orderings of a, b, c

Decision Trees and Sorting

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
- Every algorithm that sorts by using only comparisons can be represented by a decision tree
  - Only 1 leaf has sorted ordering
  - Finds correct leaf by following edges (= comparisons)
- Run time $\geq$ maximum number of comparisons
  - Depends on: depth of decision tree
  - What is the depth of a decision tree for N distinct elements?

Lower Bound on Comparison-Based Sorting

- Suppose you have a binary tree of depth d. How many leaves can the tree have?
  - e.g. depth d = 1 $\rightarrow$ at most 2 leaves,
  - d = 2 $\rightarrow$ at most 4 leaves, etc.
- A binary tree of depth d has at most 2^d leaves
- Number of leaves $L \leq 2^d \rightarrow d \geq logL$
- Decision tree has $L = N!$ leaves $\rightarrow$ its depth $d \geq log(N!)$

Lower Bound on Comparison-Based Sorting

- Stirling's approximation: $N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$
- $log(N!) \approx log\left(\sqrt{2\pi N}\left(\frac{N}{e}\right)^N\right)$
  $= log\left(\sqrt{2\pi N}\right) + log\left(\left(\frac{N}{e}\right)^N\right)$
  $= \frac{1}{2}log(2\pi N) + N(log(N) - 1) \rightarrow \Omega(N \log N)$
- Conclusion: Any sorting algorithm based on comparisons between elements requires $\Omega(NlogN)$ comparisons

Comparison of Sorting Algorithms

Algorithm	Worst case	Average case
Selection sort	$O(N^2)$	$O(N^2)$
Bubble sort	$O(N^2)$	$O(N^2)$
Insertion sort	$O(N^2)$	$O(N^2)$
Heapsort	$O(NlogN)$	$O(NlogN)$
Mergesort	$O(NlogN)$	$O(NlogN)$
Quicksort	$O(N^2)$	$O(NlogN)$

Sorting in linear time

- Comparison sort:
  - Lower bound: $\Omega(n \log n)$.
- Non comparison sort:
  - Bucket sort, radix sort
  - They can sort in linear time (under certain assumptions).

Bucket Sort

- Assumption: The input $A_1, A_2, ..., A_N$ consists of only positive integers smaller than M (obviously extensions are possible)
- Algorithm:
  - Keep an array called *count* (of size M), which is initialized to all 0s. (*count* has M buckets which are all empty)
  - When A_i is read, increment $count[A_i]$ by 1.
  - After all the input is read, scan the *count* array, printing out the representation of the sorted array.
- Running time will be $O(M + N)$.
  If M is $O(N)$ then running time will be $O(N)$.
- also called counting sort

Bucket Sort

- Assumption: Input of size n is uniformly distributed over an interval.
- Above algorithm can be extended as follows:
  - Divide input range into equal-sized subintervals (buckets).
  - Distribute n numbers into buckets.
  - Sort items in each bucket with a generic sorting algorithm.
  - Go through buckets in order to create sorted array.
- Worst case running time will be $O(N^2)$ but if number of buckets is $O(N)$ and the numbers are not distributed non-uniformly to the buckets, then average running time will be $O(N)$.

Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census
- Digit-by-digit sort.
- Hollerith's original (not optimal) idea: sort on most-significant digit first.
- Good idea: Sort on least-significant digit first with auxiliary stable sort.
- **Stable Sort Property:**The relative order of any two items with the same key is preserved after the execution of the algorithm.

Radix Sort Algorithm

Algorithm 2 RadixSort(A,d)

1: **for** $i \leftarrow 1$ *to* d **do**
  use stable BucketSort to sort array A on digit i.

- **Lemma:** Given n d-digit numbers in which each digit can take on up to k possible values, RadixSort correctly sorts these numbers in $\Theta(d(n+k))$ time.
  - If d is constant and $k = O(n)$, then time is $\Theta(n)$.

Radix Sort Example

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729