Algorithms and Lab (CSE130)

Sorting

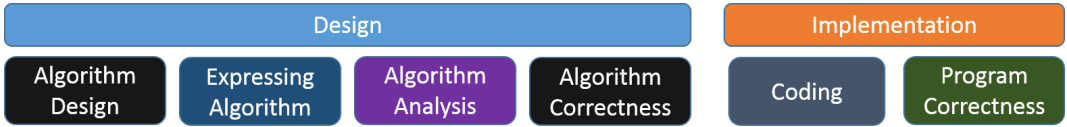
Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Last Week : Week04



Algorithm design techniques

- There are general approaches to construct efficient solutions to problems. They provide templates suited to solving a broad range of diverse problems.
- Well-known design techniques
  - Brute Force
  - Divide and conquer
  - Decrease and conquer
  - Transform and conquer
  - Dynamic programming
  - Greedy approach
  - State space search techniques

Contents

Sorting

- The sorting problem is to rearrange the items of a given list in nondecreasing order.
- Symbolically, the problem of sorting can be viewed as
  - Input: A sequence of n objects (numbers, character, strings, etc) $\langle a_1, a_2, a_3, ..., a_n \rangle$
  - Output: A permutation (reordering) of the input sequence $\langle a'_1, a'_2, a'_3, ..., a'_n \rangle$ such that each and every pair of the output has the order property i.e $a'_1 \leq a'_2 \leq a'_3 \leq ... \leq a'_n$
  - An order relation is a binary relation that ranks elements against one another. i.e. $a_i \leq a_j$ or $a_i \geq a_j$
  - Example: $[5, 7, 2, 4, 1, 3, 7] \Rightarrow [1, 2, 3, 4, 5, 6, 7]$
- The number of all permutations of an input sequence $[a_1, a_2, a_3, ..., a_n]$, is $n!$. Sorting can be done by finding all permutations and selecting the sorted one, but it is very expensive
- A pair of elements is inverted if $a_i > a_j$ *for* $i < j$. Therefore a non-sorted list has at least one inverted pair.
- Now we can define the act of sorting as removing all inverted pairs from the list.

Sorting Algorithms (cont...)

- In computing, sorting is the most fundamental problem in the study of algorithms and is used as an auxiliary step in solving several important problems, For examples

1. Data searching problems
2. Commercial computing
3. Operations research
4. Event-driven simulation
5. Numerical problems
6. Combinatorial search
7. Data compression problems
8. Minimum spanning tree problems
9. Shortest pat problems
10. String processing problems
11. load-balancing problems
12. Scheduling problems
13. Graph problems
14. Geometric problems

- Scientists have discovered dozens of different sorting algorithms

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort
7. ShellSort
8. TimSort
9. Comb Sort
10. Pigeonhole Sort
11. Cycle Sort
12. Cocktail Sort
13. Strand Sort
14. Counting Sort
15. Radix Sort
16. Bucket Sort
17. Bitonic Sort
18. Pancake sorting
19. Binary Insertion Sort
20. Permutation Sort
21. Stooge Sort
22. Tree Sort
23. Gnome Sort
24. Sleep Sort

Classification of Sorting Algorithms

Classification and Properties of Sorting Algorithms :

- Comparisons-based sorting algorithms : A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. For example heap sort, bubble sort
- Non-comparisons-based sorting algorithms: These algorithms perform sorting without comparing the elements. For example counting sort, radix sort, bucket sort
- Swaps based sorting algorithms: These algorithms swap elements to sort the input. For example exchange sort, selection Sort requires the minimum number of swaps.
- Recursive sorting algorithms: Some sorting algorithms, such as quick Sort, merge sort , use recursion to sort the input.
- Iterative sorting algorithms: Sorting algorithms, such as selection Sort or insertion Sort , use nested for-loops or while-loop to sort elements iteratively.

Classification of Sorting Algorithms (cont...)

- Stability: A sorting algorithm is **stable** if duplicate elements remain in the same relative position after sorting. Insertion sort , Merge Sort , and Bubble Sort are stable whereas Heap Sort and Quick Sort are unstable. Generally,a given sorting algorithm which is not stable can be modified to be stable.

Input	5	$3'$	2	9	$3''$	8	4	7	$3'''$	6
Sorted (stable)	2	$3'$	$3''$	$3'''$	4	5	6	7	8	9
Sorted (unstable)	2	$3'''$	$3'$	$3''$	4	5	6	7	8	9

- In-place sorting algorithms : An in-place algorithm transforms (sort) the input without using any extra memory. As the algorithm executes, the input is usually overwritten by the output, and no additional space is needed for this operation. I nsertion sort, selection sort, quick sort, bubble sort are in-place sorting algorithms

- Out-of-place sorting algorithms: An algorithm that is not in-place is called a not-in-place or out-of-place algorithm. The extra space used by an out-of-place algorithm depends on the input size. The standard merge sort algorithm is an example of out-of-place algorithm as it requires $O(n)$ extra space for merging.

Bubble Sort

- The Bubble sort algorithm compares each pair of elements in an array and swaps them if they are out of order until the entire array is sorted.
- Algorithm

```
1: procedure BUBBLESORT(a[])
2:     for (i = n − 1; i > 0; i++) do
3:         for (j = 0; j < i; j++) do
4:             if (a[j] > a[j + 1]) then
5:                 swap(a[j], a[j + 1])
6:             end if
7:         end for
8:     end for
9: end procedure
```

- Complexity Analysis

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

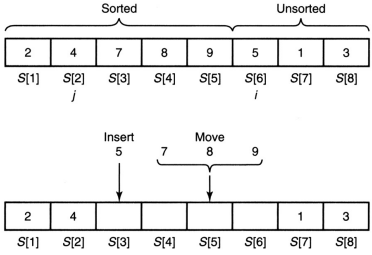
	a[1]	a[2]	a[3]	a[4]	a[5]
i=5	7	3	1	4	2
j=1	3	7	1	4	2
j=2	3	1	7	4	2
j=3	3	1	4	7	2
j=4	3	1	4	2	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=4	7	3	1	4	2
j=1	1	3	4	2	7
j=2	1	3	4	2	7
j=3	1	3	2	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=3	1	3	2	4	7
j=1	1	3	2	4	7
j=2	1	2	3	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=2	1	2	3	4	7
j=1	1	2	3	4	7
	a[1]	a[2]	a[3]	a[4]	a[5]
i=1	1	2	3	4	7

Insertion Sort

- Insertion sort is a comparison based, in-place and stable sorting algorithm

- algorithm

```
1:  procedure INSERTIONSORT(S[])
2:      for i = 1; i < n; i++ do
3:          key = S[i]
4:          j = i − 1;
5:          while j >= 0 && S[j] > key do
6:              S[j + 1] = S[j]
7:              j = j − 1
8:          end while
9:          S[j + 1] = key
10:     end for
11: end procedure
```



- Complexity Analysis:
  - Best-case: The largest number of inversions for $A[i]$ $0 \leq i \leq n-1$ is 0. This happens if $A[i]$ is smaller than all the elements to the right of it. (input is sorted already) $T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$

Insertion Sort (cont...)

- Worst-case: The largest number of inversions for $A[i]$ $0 \leq i \leq n-1$ is $n-1-i$. This happens if $A[i]$ is greater than all the elements to the right of it.

$$T_w(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} (i-1) = \sum_{i=2}^{n} i - \sum_{i=2}^{n} 1 = \frac{n(n-1)}{2} - (n-1) = \frac{n^2-3n+2}{2} \in \Theta(n^2)$$

- Average-case: If the probability of insertion of j_{th} element in the i_{th} slot is $p_j = \frac{1}{i}$ slot, the average number of comparisons needed are

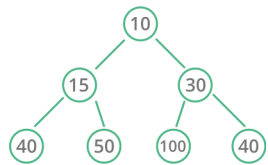
$$\sum_{j=1}^{i-1} j.p_j = \sum_{j=1}^{i-1} j.\frac{1}{i} = \frac{1}{i} \sum_{j=1}^{i-1} j = \frac{1}{i}.\frac{i(i-1)}{2} = \frac{i-1}{2}$$

- For the average number of key comparisons to sort the array of length n

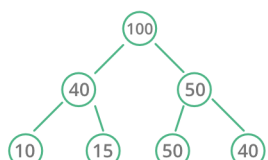
$$T_a(n) = \sum_{i=2}^{n} \frac{i-1}{2} = \sum_{i=2}^{n} \frac{i-1}{2} = \sum_{i=2}^{n} \frac{i}{2} - \sum_{i=2}^{n} \frac{1}{2} = \sum_{i=2}^{n} \frac{i}{2} = \frac{n(n-1)}{4} - \frac{n-1}{2} \in \Theta(n^2)$$

Heap (Priority Queue)

- A heap is a complete binary tree that conforms to the heap order.
- The heap order property: in a (min) heap, for every node X, the key in the parent is smaller than (or equal to) the key in X.
- The heap order property: in a (max) heap, for every node X, the key in the parent is larger than (or equal to) the key in X.
- Heap data structures are used when we wish to remove the elements based on (maximum/minimum) priorities
- For example: among the vehicles on the road, an ambulance or a fire truck are given priority



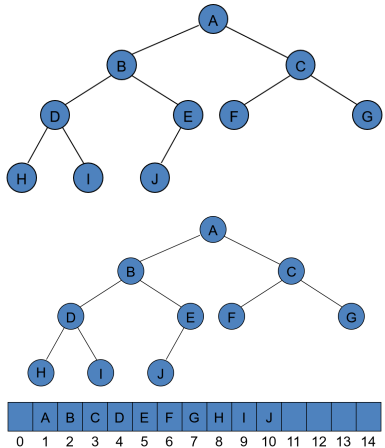
Min Heap



Max Heap

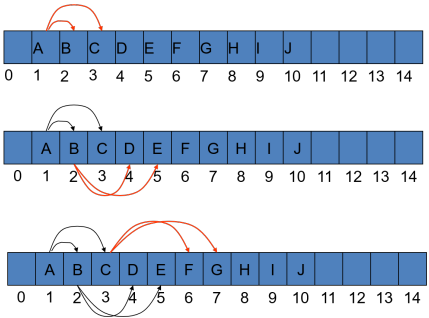
Complete Binary Tree

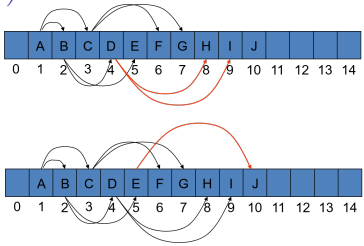
- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level.
- The bottom level is filled from left to right.
- Recall that such a tree of height h has nodes between 2^h *to* $2^{h+1} - 1$.
- The height of such a tree is $\lfloor \log_2 (N) \rfloor$ where N is the number of nodes in the tree
- Because the tree is so regular, it can be stored in an array; no pointers are necessary.



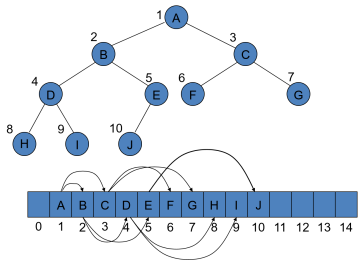
HeapSort (cont...)

- For any array element at position i, the left child is at 2i, the right child is at $(2i + 1)$ and the parent is at $\lfloor \frac{i}{2} \rfloor$





- Level-order numbers array index

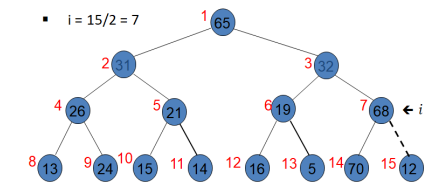


HeapSort (cont...)

How can we construct a heap for a given list of keys?

- Suppose we are given as input N keys (or items) and we want to build a max-heap of the keys.
- Obviously, this can be done with N successive inserts. But we have a better approach
- The idea is to heapify the complete binary tree formed from the array in reverse level order following a bottom-up approach.
- Suppose we have a method heapify() or percolateDown() which moves down the key in node p downwards.
- The general algorithm is to place the N keys in an array and consider it to be an unordered binary tree.
- bottom-up heap construction algorithm

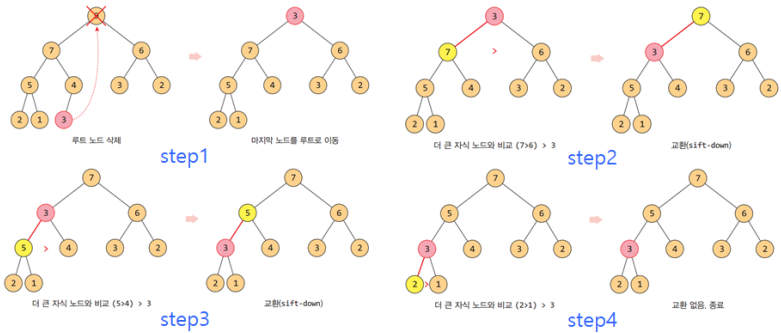
```
procedure BUILTMAXHEAP(A[])
    for i = N/2; i > 0; i– do
        heapify(A,i)
    end for
end procedure
```



HeapSort (cont...)

Deleting an element from a heap

- Step-1: Remove the element from the root and replace the last element with root
- Step-2 heapify: Shift the element(node) down by comparing it with its parent.



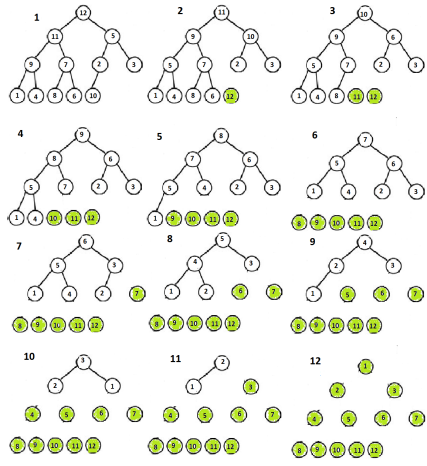
- The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1.
- The time efficiency of deletion is in $\Theta(logn)$

HeapSort Algorithm

- heapsort is an interesting sorting algorithm. This is a two-stage algorithm that works as follows

  1. (heap construction): Construct a heap for a given array of n elements .
  2. (maximum deletions): Apply the root-deletion operation $n-1$ times to the remaining heap.

```
1: procedure HEAPSORT(A[])
2:     builtMaxHeap(A)
3:     for (i = n; i > 1; i − −) do
4:         swap A[1] with A[i]
5:         size[A]= size[A]-1
6:         heapify(A, 1)
7:     end for
8: end procedure
```



HeapSort (cont...)

Worse-Case Time Complexity Analysis

- Note that:
  - Height of a binary tree is $h = logn$
  - Total number of nodes are $n = 2^{h+1} - 1$
  - level 0 of the tree has 1 node (root node), and up to level h has leaves(2^h)
  - Heapify put the element down from level $j = 0$ to $j = h - 1$
  - at level j, from the bottom there are 2^{h-j} nodes, and each might sift down j levels.
- Time for build heap function

$$T_{buildheap}(n) = \Theta(n)$$

- Time for heapify function

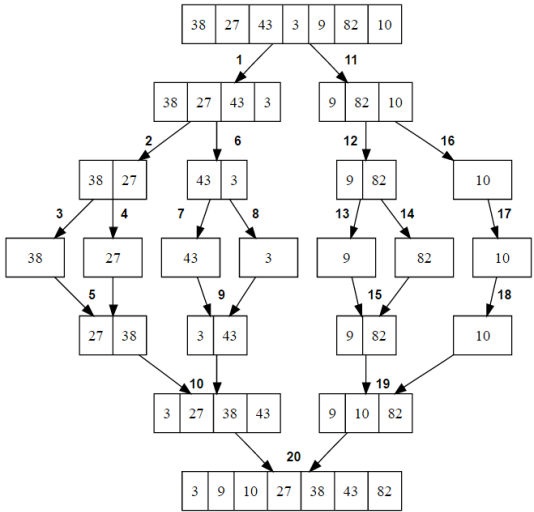
$$T_{heapify}(n) = \Theta(h) = \Theta(logn)$$

- Time for heapSort

$$T_{heapSort}(n) = \Theta(nlogn)$$

Merge Sort

- Merge sort is an efficient sorting algorithm that uses a divide-and-conquer approach to order elements in an array.
- By repeatedly applying the merging procedure, an array can be sorted.
- Merge sort involves the following steps:
  1. Divide the array into two subarrays each with $n/2$ items.
  2. Conquer (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this. The terminal condition occurs when an array of size 1 is reached; at that time, the merging begins.
  3. Merge/combine the solutions to the subarrays by merging them into a single sorted array.



Merge Sort (cont...)

Mergesort Algorithm

- Problem: Sort n keys in nondecreasing sequence.
- Inputs: positive integer n, array of keys S indexed from 1 to n.
- Outputs: the array S containing the keys in nondecreasing order.

- Pseudo-code

```
1:  procedure MERGESORT(n, S[])
2:      if (n > 1) then
3:          int h = [n/2]
4:          int m = n - h
5:          copy S[1] through S[h] to U[1] through U[h]
6:          copy S[h + 1] through S[n] to V[1] through
    V[m]
7:          mergesort(h, U)
8:          mergesort(m, V)
9:          merge(h, m, U, V, S)
10:     end if
11: end procedure
```

- Steps in the recursive mergesort algorithm:
  1. If the list has only one element, return the list and terminate. (Base case)
  2. Split the list into two halves that are as equal in length as possible. (Divide)
  3. Using recursion, sort both lists using mergesort. (Conquer)
  4. Merge the two sorted lists and return the result. (Combine).

$$T(h,m) = \begin{cases} 0 & n = 1 \\ \underbrace{T(h)}_{\text{sort } U} + \underbrace{T(m)}_{\text{sort } V} + \underbrace{T(h,m)}_{\text{merge } U,\, V} & n > 1 \end{cases}$$

Merge Sort (cont...)

Merge Algorithm

- **Problem:** Merge two sorted arrays into one sorted array.
- **Inputs:** positive integers h and m, array of sorted keys U indexed from 1 to h, array of sorted keys V indexed from 1 to m.
- **Outputs:** an array S indexed from 1 to $h + m$ containing the keys in U and V in a single sorted array.

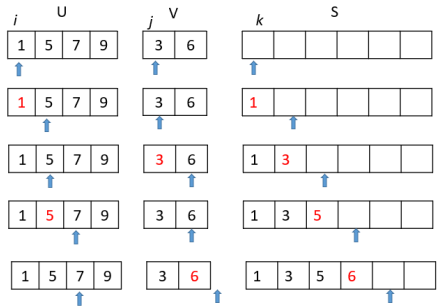
```
1: procedure MERGE((h, m, U[], V[]))
2:     S[], i = j == k = 1
3:     while (i <= h and j <= m) do
4:         if (U[i] < V[j]) then S[k] = U[i];i = i + 1
5:         else                  S[k] = V[j]; j = j + 1
6:         end if
7:         k=k+1
8:     end while
9:     if (i > h) then copy V[j] : V[m] to S[k] : S[h + m]
10:    else            copy U[i] through U[h] to S[k] through S[h + m]
11:    end if
12:    return S
13: end procedure
```

$$T(h, m) = \sum_{i=1}^{h} 1 + \sum_{j=1}^{m} 1 = h + m$$

Merge Sort (cont...)

Merging two sorted arrays

- The most important part of the merge sort algorithm is the merge step.
- The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).
- The algorithm maintains three pointers i, j, k, one for each of the two arrays and one for maintaining the current index of the final sorted array.



Merge Sort (cont...)

Worse-Case Time Complexity Analysis

- Complexity Function

$$T(h,m) = \begin{cases} 0 & n = 1 \\ \underbrace{T(h)}_{\text{sort } U} + \underbrace{T(m)}_{\text{sort } V} + \underbrace{T(h,m)}_{\text{merge } U,V} & n > 1 \end{cases}$$

- Suppose n is a power of 2. Then we have

$$h = \frac{n}{2}$$
$$m = n - h \Rightarrow n - \frac{n}{2} = \frac{n}{2}$$
$$n = h + m \Rightarrow \frac{n}{2} + \frac{n}{2} = n$$

- The Complexity function is reduced to

$$T(n) = \begin{cases} 0 & n = 1 \\ \underbrace{T(\frac{n}{2})}_{\text{sort } U} + \underbrace{T(\frac{n}{2})}_{\text{sort } V} + \underbrace{h+m}_{\text{merge } U,V} & n > 1 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

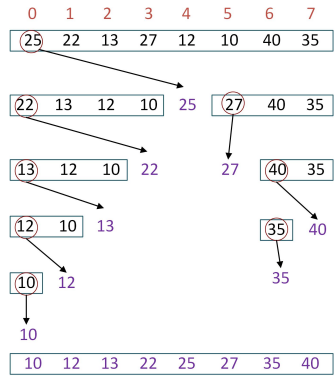
- Master Theorem:

$$\begin{cases} T(n) \in \Theta\left(n^k\right) & a < b^k \quad (case-1) \\ T(n) \in \Theta\left(n^k \lg n\right) & a = b^k \quad (case-2) \\ T(n) \in \Theta\left(n^{\log_b a}\right) & a > b^k \quad (case-3) \end{cases}$$

Since, $a = 2, b = 2, k = 1$ and
$a = b^k \Rightarrow\ 2 = 2^1\ \ (case-2)$
$T(n) \in \Theta(n\, log_2 n)$

Quicksort (Partition Exchange Sort)

- Quicksort was developed by Hoare (1962).
- Quicksort is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively.
- On the basis of Divide/Decrease and conquer approaches, quicksort algorithm can be explained as:
  - Divide/Decrease: The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.
  - Conquer The left and the right subparts are again partitioned recursively and the solution for the smallest problem is obtained.
  - Combine: Do not need to merge/combine as the array is already sorted at the end of the conquer step.



Quicksort Algorithm

- Problem: Sort n keys in nondecreasing sequence.
- Inputs: positive integer n, array of keys S indexed from 1 to n.
- Outputs: the array S containing the keys in nondecreasing order.

- Pseudo-code

```
1: procedure QuickSort(A[], low , high)
2:     if (high > low) then
3:         pi = partition(A, low, high)
4:         quicksort(A, low, pi − 1))
5:         quicksort(A, pi + 1, high)
6:     end if
7: end procedure
```

- Choosing a Pivot :The pivot item can be any item from the input array however, it is difficult to determine what a good pivot might be.
- The choice of pivot item may affect the complexity of the algorithm.
- Appropriate choices may include
  1. Select a random pivot.
  2. Select the first or the middle or the last element as the pivot.
  3. Take the first, middle, and last value of the array, and choose the median of those three numbers as the pivot (median-of-three method).

Quicksort (Partition Exchange Sort) (cont...)

Partition Algorithm

- **Problem:** Partition the array S for Quicksort.
- **Inputs:** two indices, low and high, and the subarray of S indexed from low to high.
- **Outputs:** pivotpoint, the pivot point for the subarray indexed from low to high.

- Pseudo-code

```
1:  procedure PARTITION(A, low, high)
2:      i = 0  j = low,  pp = 0
3:      pivot = A[low]
4:      for (i = low + 1; i <= high; i++) do
5:          if (A[i] < pivot) then
6:              j = j + 1
7:              exchange(A[i] and A[j])
8:          end if
9:      end for
10:     exchange(A[low] and A[j])
11:     return j
12: end procedure
```

- Partition method is the most important function in quick sort.
- It rearranges the elements of the array so that all the elements to the left of the pivot are smaller than the pivot and all the elements to the right are greater than the pivot.
- For convenience we simply take the first element in the array as pivot item.

$$T(n) = \sum_{i=2}^{n} 1 = n - 1$$

Tracing Partition Algorithm

- Let the input array $S = [7 \quad 3 \quad 2 \quad 8 \quad 6 \quad 4 \quad 5 \quad 9 \quad 1]$
- At each iteration, values for different variables inside the for loop

$pivot = S[1] = 7, i = 2, \ j = 1$

1. $S[i = 2] < pivot, \ 3 < 7 \ \ true$
   $\Rightarrow j++, \ excange \ \ S[i = 2] \ and \ S[j = 2] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 8 \quad 6 \quad 4 \quad 5 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 3, \ j = 2$

2. $S[i = 3] < pivot, \quad 2 < 7 \ \ true$
   $\Rightarrow \ j++, \ \ excange \ \ S[i = 3] \ and \ S[j = 3] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 8 \quad 6 \quad 4 \quad 5 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 4, \ j = 3$

4. $S[i = 4] < pivot, \quad 8 < 7 \ \ False$
   $\Rightarrow$, *Nothing to do* $\quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 8 \quad 6 \quad 4 \quad 5 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 5, \ j = 3$

4. $S[i = 5] < pivot, \quad 6 < 7 \ \ true$
   $\Rightarrow \ j++, \ \ excange \ \ S[i = 5] \ and \ S[j = 4] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 6 \quad 8 \quad 4 \quad 5 \quad 9 \quad 1]$

Quicksort (Partition Exchange Sort) (cont...)

$pivot = S[1] = 7, i = 6,\ j = 4$

6. $S[i = 6] < pivot,\quad 4 < 7\ \ true$
   $\Rightarrow j++,\ \ excange\ \ S[i = 6]\ and\ S[j = 5] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 6 \quad 4 \quad 8 \quad 5 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 7,\ j = 5$

7. $S[i = 7] < pivot\ ,\quad 5 < 7\ \ true$
   $\Rightarrow j++;\ \ excange\ \ S[i = 7]\ and\ S[j = 6] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 6 \quad 4 \quad 5 \quad 8 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 8,\ j = 6$

8. $S[i = 8] < pivot,\quad 9 < 7\ \ False$
   $\Rightarrow nothing\ todo \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 6 \quad 4 \quad 5 \quad 8 \quad 9 \quad 1]$

$pivot = S[1] = 7, i = 9,\ j = 6$

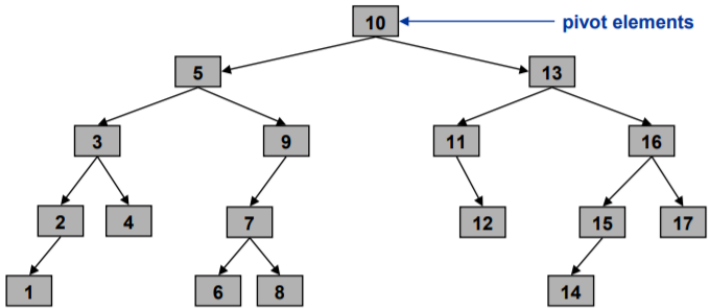
9. $S[i = 9] < pivot\ \ 1 < 7\ \ true$
   $\Rightarrow j++,\ \ excange\ \ S[i = 9]\ and\ S[j = 7] \quad \Rightarrow S = [7 \quad 3 \quad 2 \quad 6 \quad 4 \quad 5 \quad 1 \quad 9 \quad 8]$

- *End of For Loop*
  $excange\ \ S[1]\ and\ S[j = 7] \quad \Rightarrow S = [1 \quad 3 \quad 2 \quad 6 \quad 4 \quad 5 \quad 7 \quad 9 \quad 8] \quad \Rightarrow return\ j = 7$

Quicksort (Partition Exchange Sort) (cont...)

BST Representation of Pivots

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16
1	2	3	4	5	8	6	7	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17



Quicksort (Partition Exchange Sort) (cont...)

Worst-Case Complexity Analysis

- The worst case occurs if the array is already sorted in nondecreasing order. As when partition is called at the top level, no items are placed to the left of the pivot item, and the value of pivot position assigned by partition is 1.
- Thus, the array is repeatedly partitioned into an empty subarray on the left and a subarray with one less item on the right. Total time for the quick sort algorithm
- Time complexity function

$$T(n) = \underbrace{T(0)}_{\text{left subarray}} + \underbrace{T(n-1)}_{\text{right subarray}} + \underbrace{(n-1)}_{\text{partition}}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + n - 1 & n > 0 \end{cases}$$

- Complexity in terms of asymptomatic notations (Solution on next slide)

$$T(n) = \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

Quicksort (Partition Exchange Sort) (cont...)

- Consider the following non-homogeneous recurrence relation

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + n - 1 & n > 0 \end{cases}$$

- The characteristic equation with root $r = 1$

$$(r-1)(r-1)^2 = 0$$
$$(r-1)^3 = 0$$

- The general solution to the recurrence:

$$T(n) = c_1(1^n) + c_2 n(1^n) + c_3 n^2(1^n)$$
$$= c_1 + c_2 n + c_3 n^2$$

- We need three initial conditions to get three values

$$T(0) = 0$$
$$T(1) = T(0) + 1 - 1 = 0 + 0 = 0$$
$$T(2) = T(1) + 2 - 1 = 0 + 1 = 1$$

- Set of questions after applying initial conditions

$$c_1 = 0$$
$$c_1 + c_2 + c_3 = 0$$
$$c_1 + 2c_2 + 4c_3 = 1$$

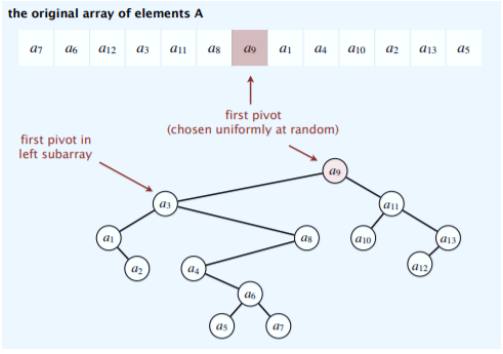
- Getting values for constants
  $c_1 = 0, c_2 = -1/2, c_3 = 1/2$, the final solution

$$T(n) = \frac{n(n-1)}{2}$$

Quicksort (Partition Exchange Sort) (cont...)

Average-Case Complexity Analysis

- Proposition: The expected number of compares to quicksort an array of n distinct elements $a_1 < a_2 < ... < a_n$ is $O(nlogn)$.
- Consider BST representation of pivot elements.
- a_i and a_j are compared once iff one is an ancestor of the other.
- Pr [ai and aj are compared] $= \frac{2}{j-i+1}$, where $i < j$
- Let pivot is selected as a random element



$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j} \leq 2n \sum_{j=2}^{n} \frac{1}{j} = 2nlogn \in O(nlogn)$$

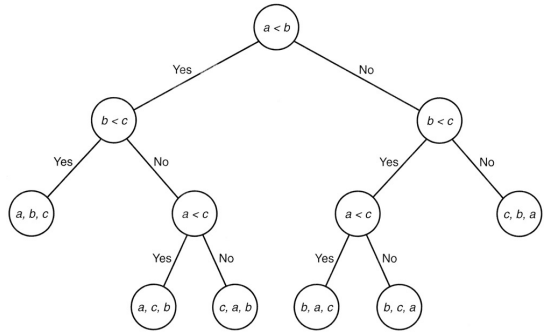
Lower Bounds for Sorting Only by Comparison of Keys

Lower Bounds for Sorting Only by Comparison of Keys

- Comparison sorts can be viewed abstractly in terms of decision trees.
- A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.
- A decision tree is called valid for sorting n keys if, for each permutation of the n keys, there is a path from the root to a leaf that sorts that permutation
- A decision tree is pruned if every leaf can be reached from the root by making a consistent sequence of decisions.
- Lemma : To every deterministic algorithm for sorting n distinct keys there corresponds a pruned, valid, binary decision tree containing exactly $n!$ leaves.
- Any deterministic comparison-based sorting algorithm must perform $\Omega(nlogn)$ comparisons to sort n elements in the worst case. Specifically, for any deterministic comparison-based sorting algorithm A, for all n 2 there exists an input I of size n such that A makes at least $log_2(n!) = \Omega(nlogn)$ comparisons to sort I.
- Consider the following two algorithms for sorting three keys $S[a, b, c]$

Lower Bounds for Sorting Only by Comparison of Keys (cont...)

```
1:  procedure sortThree(S[a, b, c])
2:      if (a < b) then
3:          if (b < c) then
4:              S[a, b, c]
5:          else
6:              if (a < c) then
7:                  S[a, c, b]
8:              else
9:                  S[c, a, b]
10:             end if
11:         end if
12:     else
13:         if (b < c) then
14:             if (a < c) then
15:                 S[b, a, c]
16:             else
17:                 S[b, c, a]
18:             end if
19:         else
20:             S[c, b, a]
21:         end if
22:     end if
23: end procedure
```



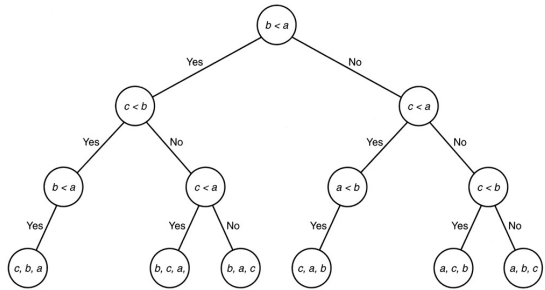
- The decision tree corresponding to procedure sortthree.
- There is a leaf in the tree for every permutation of three keys, because the algorithm can sort every possible input of size 3.

Lower Bounds for Sorting Only by Comparison of Keys (cont...)

- Exchange Sort algorithm

```
1: procedure EXCHANGESORT(S[])
2:     integer i, j
3:     for (i = 1; i <= n − 1; i++) do
4:         for (j = i + 1; j <= n; j++) do
5:             if (S[j] < S[i]) then
6:                 exchange S[i] and S[j]
7:             end if
8:         end for
9:     end for
10: end procedure
```

- The decision tree corresponding to Exchange Sort when sorting three keys.



- Complexity analysis

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

$$= n^2 - n - \frac{n^2 - n}{2} = \frac{2n^2 - 2n - n^2 + n}{2} = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Lower Bounds for Sorting Only by Comparison of Keys (cont...)

- Any comparison based sorting algorithm must make at least $\Omega(nlogn)$ comparisons to sort the input array

$$\underbrace{n!}_{permutations} \leq \underbrace{2^x}_{maximum\ leaves}$$
$$log_2(n!) \leq log_2(2^x)$$
$$\leq x \in \Omega(nlog_2 n)$$

- Consider the following

$$\begin{aligned} n! &= n(n-1)(n-2)...1 \\ &= (n-0)(n-1)...(n-(n-1)) \\ &= n(1-\frac{0}{n}) \times n(1-\frac{1}{n})...n(1-\frac{n-1}{n}) \\ &= n^n \left[(1-\frac{0}{n}) \times (1-\frac{1}{n})...(1-\frac{n-1}{n})\right] \\ &= n^n \Pi_{i=0}^{n-1}(1-\frac{i}{n}) \end{aligned}$$

- Taking log on both sides

$$\begin{aligned} log(n!) &= log(n^n \Pi_{i=0}^{n-1}(1-\frac{i}{n})) \\ &= log(n^n) + log(\Pi_{i=0}^{n-1}(1-\frac{i}{n})) \\ &= nlogn + log(\Pi_{i=0}^{n-1}(1-\frac{i}{n})) \end{aligned}$$

- Hence

$$nlogn \leq nlogn + log(\Pi_{i=0}^{n-1}(1-\frac{i}{n}))$$
$$nlogn \in \Omega(nlogn)$$

Counting sort

Non-comparisons-based sorting algorithms:

- Algorithm that sorts only by comparisons of keys can be no better than $\Theta(nlgn)$.
- There are some sorting algorithms that perform sorting without comparing the elements such sorting algorithms are known as the non-comparison based sorting algorithms.
- In these algorithms, the keys are distributed into piles, so some times are called sorting by distribution
- non-comparison based sorting algorithm with following assumptions about input.
  1. All the entries in the array are integers.
  2. The difference between the maximum value and the minimum value in the array (range of element) is not too high.
- Generally, the non-comparison based sorting algorithms like Radix Sort, Counting Sort, and Bucket Sort are faster than comparison based sorting algorithms such as Selection Sort, QuickSort,

Counting sort (cont...)

Counting sort

- Counting sort is an efficient algorithm for sorting that sorts the elements of an array by counting the number of occurrences of each unique element in the array.
- The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.
- Counting Sort algorithm works on the keys that are small integer and lies between a specific range.
- It is not an in-place sorting algorithm as it requires extra additional space
- Counting Sort is stable sort as relative order of elements with equal values is maintained.
- Counting Sort is inefficient if the range of key value (k) is very large.
- Counting Sort has a disadvantage that it can only sort discrete values like integer

int input[] = { 2, 1, 4, 5, 7, 1, 7, 11, 8, 9, 10 };

Index	0	1	2	3	4	5	6	7	8	9	10	11
Count[]	0	2	1	0	1	1	0	2	1	1	1	1

Index	0	1	2	3	4	5	6	7	8	9	10	11
Modified Count[]	0	2	3	3	4	5	5	7	8	9	10	11

Count[i]=Count[i] + Count[i-1]

Result[]	0	1	1	2	4	5	7	7	8	9	10	11

Count[input[i]] will tell you the index position of input[i] in Result[]

Counting sort (cont...)

- Algorithm

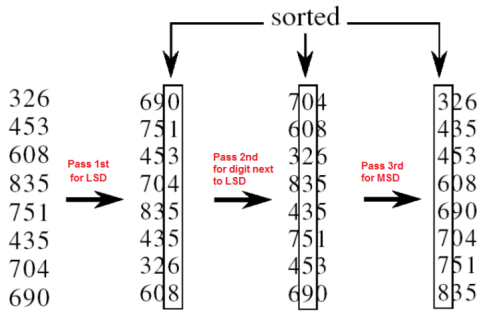
```
1:  procedure COUNTINGSORT(A[], k)
2:      for (i = 0; i < k; i++) do
3:          C[i] = 0
4:      end for
5:      for (j = 0; j < n; j++) do
6:          C[A[j]] = C[A[j]] + 1
7:      end for
8:      for (i = 1; i < k; i++) do
9:          C[i] = C[i] + C[i − 1]
10:     end for
11:     for (j = n − 1; j >= 0; j++) do
12:         B[C[A[j]] − 1] = A[j]
13:         C[A[j]] = C[A[j]] − 1
14:     end for
15: end procedure
```

- Complexity Analysis

$$
\begin{aligned}
T(n,k) &= \sum_{i=1}^{k} 1 + \sum_{j=1}^{n} 1 + \sum_{i=1}^{k-1} 1 + \sum_{j=1}^{n} 1 \\
&= k + n + k - 1 + n \\
&= 2k + 2n - 1 \\
&\in \Theta(n + k)
\end{aligned}
$$

Radix sort

- Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value).
- Radix sort uses counting sort as a subroutine to sort an array of numbers.
- Radix Sort is a non-comparative sorting algorithm
- Radix sort is a stable sort, which means it preserves the relative order of elements that have the same key value.
- Passes
  - For 1st pass: we sort the array on basis of the least significant digit (1s place) using counting sort.
  - For 2nd pass: we sort the array on basis of the next digit (10s place) using counting sort.
  - For 3rd pass: we sort the array on basis of the most significant digit (100s place) using counting sort.



Radix sort (cont...)

- Algorithm

```
1: procedure RadixSort(S[])
2:     d = maximum number of digits
3:     n=size(A)
4:     for j = 1 to d do
5:         count = [0] * 10
6:         for j = 1 to n do
7:             count[key of(A[i]) in pass j]++
8:         end for
9:         for k = 1 to 10  do
10:            count[k] = count[k] + count[k-1]
11:        end for
12:        for i = n down to 1  do
13:            result[ count[key of(A[i])] ] = A[j]
14:            count[key of(A[i])]–
15:        end for
16:        for j = 1 to n do
17:            A[i] = result[i]
18:        end for
19:    end for
20: end procedure
```

- Complexity Analysis
  - Let there be d digits in input integers.

$$
\begin{aligned}
T(d,n,k) &= \sum_{m=1}^{d}\left[\sum_{i=1}^{k}1+\sum_{j=1}^{n}1+\sum_{i=1}^{k-1}1+\sum_{j=1}^{n}1\right] \\
&= \sum_{m=1}^{d}\left[k+n+k-1+n\right] \\
&= d\left[2k+2n-1\right] \\
&\in \Theta(d(n+k))
\end{aligned}
$$

  - Radix Sort takes $\Theta(d \times (n+k))$ time where k is the base for representing numbers, for example, for the decimal system, k is 10.

Summary