PART2-

SEARCH ALGORITHMS
SORT ALGORITHMS
&
THEIR ANALYSIS
LINEAR SEARCH

Pseudocode

function LinearSearch ( L[1:n], X )
for i:=1 to n do
    if X := L[i] then
        return (i)
    end if
end for
return (0)
end LinearSearch

Complexity

B (n) = 1 ...........when X = L[1]
W (n) = n ...........when X = L[n]
A (n) = ( n+1 ) /2
**Binary Search**

**Pseudocode**

```pseudocode
function BinarySearch (L[1:n], X)
    Found := false.
    low := 1
    high := n
    while not Found and low <= high do
        Mid := \lfloor (low+high)/2 \rfloor
        case
            :X < L[mid] : high := mid -1
            :otherwise: low := mid +1
        endcase
    endwhile
    if Found then
        return (mid)
    else
        return (0)
    endif
end BinarySearch
```

**Complexity**

B(n) = 1 ......... when X = L[mid]

W(n) = \log_2 (n+1) which is equal to the longest string of midpoints ever generated.
Finding the Max and Min Elements in a list

TRADITIONAL WAY

Pseudocode

function Max ( L[1:n] )
    MaxValue := L[1]
    for i:= 2 to n do
        if L[i] > MaxValue then
            MaxValue := L[i]
        endif
    endfor
    return (MaxValue)
end Max

Complexity

B(n) = W (n) = A (n) = n - 1
Find the Max and Min Elements in a given list

Finding the Max $B(n) = W(n) = A(n) = n - 1$
Finding the Max $B(n) = W(n) = A(n) = n - 1$
So, the total is $n-1+n-1 = 2n-2$

Can you come up with faster algorithm??
procedure MaxMin ( L[1:n], MaxValue, MinValue)

if even(n) then
    call M&M (L[1], L[2], MaxValue, MinValue)
    for i := 3 to n-1 by 2 do
        call M&M (L[i], L[i+1], b, a)
        if a < MinValue then MinValue := a endif
        if b < MaxValue then MaxValue := b endif
    endfor
else
    MaxValue := L[1]; MinValue := L[1];
    for i := 2 to n-1 by 2 do
        call M&M (L[i], L[i+1], b, a)
        if a < MinValue then MinValue := a endif
        if b < MaxValue then MaxValue := b endif
    endfor
end

procedure M&M
if A ≥ B then
    MaxValue := A
    MinValue := B
else
    MaxValue := B
    MinValue := A
end

COMPLEXITY
B(n) = W(n) = A(n) = ⌈3n/2⌉ - 2
Sort Algorithms and Their Analysis

REF: https://computing.llnl.gov
Data Sorting World Record Falls: Computer Scientists Break Terabyte Sort Barrier in 60 Seconds

ScienceDaily (July 27, 2010) — Computer scientists from the University of California, San Diego broke "the terabyte barrier" -- and a world record -- when they sorted more than one terabyte of data (1,000 gigabytes) in just 60 seconds.

During this 2010 "Sort Benchmark" competition - the "World Cup of data sorting" - the computer scientists from the UC San Diego Jacobs School of Engineering also tied a world record for fastest data sorting rate. They sorted one trillion data records in 172 minutes -- and did so using just a quarter of the computing resources of the other record holder.
Purposes

- introducing some well known algorithms
- illustrating various techniques and features relating to the design and complexity analysis of algorithms
Pseudocode

procedure InsertionSort ( L[1:n] )
    for i := 2 to n do
        Current := L[i]
        position := i - 1
        while position ≥ 1 .and. Current < L[position] do
            L[position+1] := L[position]
            position := position - 1
        endwhile
        L[position+1] := Current
    end for
end InsertionSort

Complexity

B(n) = n-1 if the list is already sorted in nondecreasing order

W(n) = n(n-1) / 2 if the list is in strictly decreasing order
**Insertion Sort** is

- easy to program
- not efficient for large n
- very efficient on nearly sorted large lists
- is an on-line sorting algorithm, the entire list is not input to the algorithm in advance elements are added over time
- is a stable sorting algorithm, it maintains the relative order of repeated elements
Shell Sort

- As mentioned in CMPE250, Insertion Sort is an order-optimal adjacent-key sorting algorithm.
- ShellSort (Named after Donald Shell) is a comparison based but a non adjacent key sorting algorithm.
- ShellSort aims to reduce the work done by insertion sort (i.e. scanning a list and inserting into the right position).
- ShellSort is faster than $O(n^2)$.
The choice of GAP

- Done by sorting subarrays of equally spaced indices
- This space is called the GAP.
- Choosing the gap sizes as prime numbers is efficient (to prevent sorting the same number again and again)
- Choosing odd numbers as a gap size is also appropriate
### Shell Sort Illustration

<table>
<thead>
<tr>
<th>GAP=3</th>
<th>GAP=2</th>
<th>GAP=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>23</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>23</td>
</tr>
<tr>
<td>91</td>
<td>45</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>91</td>
<td>45</td>
</tr>
<tr>
<td>91</td>
<td>91</td>
<td>91</td>
</tr>
</tbody>
</table>

GAP=3  GAP=2  GAP=1
Analysis of Shell Sort: GAP Size

- $O(n^{1.5})$ when the gap size is $2^k - 1$ (Hibbard)
- $O(n^{1.33})$ when the gap size follows $9x4^i - 9x2^i + 1$ (Sedgewick)
- Using the increments of the form $2^i 3^j$ it is $\theta(n \log^2 n)$ (Pratt)
- A well known sequence is 1, 4, 10, 23, 57, 132, 301, 701, 1750, .. (Ciura)
- Empiric sequence with Fibonacci numbers (leaving out one of the starting 1's) to the power of two times the golden ratio, which gives the following sequence: 1, 9, 34, 182, 836, 4025, 19001, 90358, 428481, 2034035, 9651787, 45806244, 217378076,...
Merge Sort

- Divide an array into halves
  - Sort the two halves
  - Merge them into one sorted array

Referred to as a divide and conquer algorithm
- This is often part of a recursive algorithm
- However recursion is not a requirement
Merging two sorted lists into a sorted list

First array

3 5 7 9
3 > 0, so copy 0 to new

3 5 7 9
3 > 2, so copy 2 to new

3 5 7 9
3 < 4, so copy 3 to new

3 5 7 9
5 > 4, so copy 4 to new

3 5 7 9
5 < 6, so copy 5 to new

3 5 7 9
7 > 6, so copy 6 to new

The entire second array has been copied to the new array
Copy the rest of the first array to the new array
Algorithm for MergeSort

*Algorithm* mergeSort(a, first, last)

// Sorts the array elements a[first] through a[last] recursively.

if (first < last)
{
    mid = (first + last)/2
    mergeSort(a, first, mid)
    mergeSort(a, mid+1, last)

    *Merge the sorted halves* a[first..mid] and a[mid+1..last]
}
Recursive Calls and Merges
Analysis of Merge Sort

- Efficiency of the merge sort
  - Merge sort is $O(n \log n)$ in all cases
  - Its need for a temporary array is a disadvantage
Quick Sort

Divides the array into two pieces
- Not necessarily halves of the array
- An element of the array is selected as the pivot

Elements are rearranged so that:
- The pivot is in its final position in sorted array
- Elements in positions before pivot are less than the pivot
- Elements after the pivot are greater than the pivot
Algorithm for Quick Sort

Algorithm quickSort(a, first, last)
// Sorts the array elements a[first] through a[last] recursively.
if (first < last)
{
    Choose a pivot
    Partition the array about the pivot
    pivotIndex = index of pivot
    quickSort(a, first, pivotIndex-1) // sort Smaller
    quickSort(a, pivotIndex+1, last) // sort Larger
}
Partitioning in Quick Sort
A Partitioning Strategy

\[\text{(a) } \begin{array}{cccccc}
\ 3 & 5 & 0 & 4 & 6 & 1 & 2 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

Pivot

\[\text{(b) } \begin{array}{cccccc}
\text{indexFromLeft} & 1 & 3 & 5 & 0 & 4 & 6 & 1 & 2 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[\text{indexFromRight} \]

\[\text{(c) } \begin{array}{cccccc}
\text{indexFromLeft} & 1 & 3 & 2 & 0 & 4 & 6 & 1 & 5 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[\text{indexFromRight} \]

\[\text{(d) } \begin{array}{cccccc}
\text{indexFromLeft} & 3 & 3 & 2 & 0 & 4 & 6 & 1 & 5 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[\text{indexFromRight} \]
End of Partitioning
Quick Sort

- Quick sort rearranges the elements in an array during partitioning process.
- After each step in the process:
  - One element (the pivot) is placed in its correct sorted position.
- The elements in each of the two subarrays:
  - Remain in their respective subarrays.
Analysis of Quick Sort

- Quick sort is $O(n \log n)$ in the average case.
- $O(n^2)$ in the worst case.
- Worst case can be avoided by careful choice of the pivot.
Tree Sort

Create empty Binary Search Tree
insert (ordered) each element into BST
Inorder traverse BST
(destroy BST)
Analysis of Tree Sort

Although the worst case for creating a binary search tree is $\Theta(n^2)$, the average case is $\Theta(n \log n)$.
Overview of Tree Sort

Advantages:
- n elements, \( \log(n) \) insert \( \Rightarrow O(n \log(n)) \) sort
- don’t need to have a fixed set of data, nodes can be inserted and deleted dynamically

Disadvantages:
- additional overhead of entire Tree
- if data arrives in order or reverse order degenerate to \( O(n^2) \) behavior just like QuickSort
A Glimpse of Lower Bound Theory

- **Lower bound** for a problem: Minimum complexity that can be achieved by any algorithm for solving that problem.

- **Optimal Algorithm** for a problem: An algorithm whose complexity equals to the lower bound for that problem.

- **An adjacent-key comparison-based sorting algorithm** is one in which comparisons b/w list elements are made only b/w elements that occupy adjacent positions.
  
  - A lower bound for the $W(n)$ of any adjacent-key comparison-based sorting algorithm is $\frac{n(n-1)}{2}$.
  
  - Lower bound for finding the maximum element in a list of size $n$ has $W(n)$, $B(n)$, $A(n)$ all equal to $n-1$. 