PART2-

SEARCH ALGORITHMS
SORT ALGORITHMS
&
THEIR ANALYSIS
**LINEAR SEARCH**

**Pseudocode**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>function LinearSearch ( L[1:n], X )</td>
</tr>
<tr>
<td>2.</td>
<td>for i := 1 to n do</td>
</tr>
<tr>
<td>3.</td>
<td>if X := L[i] then</td>
</tr>
<tr>
<td>4.</td>
<td>return (i)</td>
</tr>
<tr>
<td>5.</td>
<td>end if</td>
</tr>
<tr>
<td>6.</td>
<td>end for</td>
</tr>
<tr>
<td>7.</td>
<td>return (0)</td>
</tr>
<tr>
<td>8.</td>
<td>end LinearSearch</td>
</tr>
</tbody>
</table>

**Complexity**

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B (n) = 1</td>
<td>when X = L[1]</td>
</tr>
<tr>
<td>W (n) = n</td>
<td>when X = L[n]</td>
</tr>
<tr>
<td>A (n) = ( n+1 ) /2</td>
<td></td>
</tr>
</tbody>
</table>
BINARY SEARCH

Pseudocode

function BinarySearch ( L[1:n], X )

Found := .false.
low := 1
high := n

while .not. Found .and. low ≤ high do

Mid := ⌊(low+high)/2⌋

  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) = log₂ (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**

```plaintext
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
    endif
end MaxMin

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

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**

```plaintext
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
- \( W(n) = \frac{n(n-1)}{2} \) 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 9</td>
<td>9 4</td>
<td>4 4</td>
</tr>
<tr>
<td>23 6</td>
<td>6 6</td>
<td>6 6</td>
</tr>
<tr>
<td>4 4</td>
<td>4 8</td>
<td>8 7</td>
</tr>
<tr>
<td>9 24</td>
<td>9 23</td>
<td>7 8</td>
</tr>
<tr>
<td>6 8</td>
<td>8 9</td>
<td>9 9</td>
</tr>
<tr>
<td>7 9</td>
<td>9 9</td>
<td>9 9</td>
</tr>
<tr>
<td>7 7</td>
<td>7 7</td>
<td>7 7</td>
</tr>
<tr>
<td>8 8</td>
<td>8 8</td>
<td>8 8</td>
</tr>
<tr>
<td>6 6</td>
<td>6 6</td>
<td>6 6</td>
</tr>
<tr>
<td>45 45</td>
<td>45 45</td>
<td>45 45</td>
</tr>
<tr>
<td>23 23</td>
<td>23 23</td>
<td>23 23</td>
</tr>
<tr>
<td>12 12</td>
<td>12 12</td>
<td>12 12</td>
</tr>
<tr>
<td>24 24</td>
<td>24 24</td>
<td>24 24</td>
</tr>
<tr>
<td>91 91</td>
<td>91 91</td>
<td>91 91</td>
</tr>
</tbody>
</table>
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 $2^i3^j$ 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

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

3 > 0, so copy 0 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

3 > 2, so copy 2 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

3 < 4, so copy 3 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

5 < 4, so copy 4 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

5 < 6, so copy 5 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

7 > 6, so copy 6 to new

\[ \begin{array}{c}
3 & 5 & 7 & 9 \\
\end{array} \]

The entire second array has been copied to the new array

Copy the rest of the first array to the new array

Second array

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

\[ \begin{array}{c}
0 & 2 & 4 & 6 \\
\end{array} \]

New merged array

\[ \begin{array}{c}
0 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
9 \\
\end{array} \]
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

Effect of recursive calls to mergeSort

Merge steps

Copy to original array
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

continued
End of Partitioning

(e) indexFromLeft

(f) indexFromLeft

(g) indexFromLeft

(h) indexFromLeft
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 $n-1$. 
9.1 RECAP:

ELEMENTARY PROBABILITY THEORY

- Consider throwing two fair dice.
- The set of all outcomes (called sample space) given below:

<table>
<thead>
<tr>
<th>D1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
</tr>
<tr>
<td>2</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
</tr>
<tr>
<td>3</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
</tr>
<tr>
<td>4</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
</tr>
<tr>
<td>5</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
</tr>
<tr>
<td>6</td>
<td>(6,1)</td>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
</tr>
</tbody>
</table>
9.1 ELEMENTARY PROBABILITY THEORY

Since dice are fair the probability that the die shows number $i$ is $1/6$ $i \in \{1, \ldots, 6\}$

We wish to calculate sum of two dice is 5

There are 4 possibilities that sum of dice equals 5 which are: $(1,4),(2,3),(3,2),(4,1)$

So $P(\text{sum of two dice is 5}) = 4/36 = 1/9$
9.1.1 SAMPLE SPACES & PROBABILITY DISTRIBUTIONS

• Sample space (denoted by $S$) ⇒ set of all outcomes in the experiment

• Event ⇒ A subset of outcomes from $S$

• A probability distribution arises by $P(E)$ ⇒ each event $E$ in $S$

$$P(E) = \frac{|E|}{|S|}$$

$P(E)$ of an event $E$, if $S$ is finite AND Each event equally likely to occur

• Two events are *mutually independent* if $E \cap F = \emptyset$
A probability distribution $P$ on the sample space $S$ is a mapping from the events in $S$ to the real numbers satisfying the following

**Axiom 1:** $0 \leq P(E) \leq 1$

**Axiom 2:** $P(S) = 1$

**Axiom 3:** $P(\bigcup_{i=1}^{\infty} E_i) = \sum_{i=1}^{\infty} P(E_i)$
9.1.2 CONDITIONAL PROBABILITY

- Plays big role in finding average behaviour of algorithms

**Example**: Find the probability of that the sum of 2 dice is at most 5 given first die is 1

**Solution**: Denote $E \Rightarrow \text{sum of 2 dice is at most 5}$

$F \Rightarrow \text{first die is 1}$

$P(E|F) \Rightarrow \text{cond. probability of } E \text{ given } F$

$P(E) = \frac{10}{36} = \frac{5}{18}$ and $P(F) = \frac{6}{36} = \frac{1}{6}$

$E \cap F = \{(1,1),(1,2),(1,3),(1,4)\}$ So,

$P(E|F) = \frac{|E \cap F|}{P(F)} = \frac{4}{6} = \frac{2}{3}$
9.1.2 CONDITIONAL PROBABILITY

• General definition of conditional probability:
  \[ P(E|F) = \frac{P(E \cap F)}{P(F)} \]

• Events E and F are independent if
  \[ P(E|F) = P(E) \text{ or } P(E \cap F) = P(E)P(F) \]

• By using mutually exclusiveness and axioms of probability
  Baye’s formula can be derived which is
  \[ P(E) = P(F) \cdot P(E|F) + (1 - P(F)) \cdot P(E|F^c) \]
9.1.3 RANDOM VARIABLES AND EXPECTATION

• A random variable $X$ on sample space $S$ is a mapping from $S$ to the set $\mathbb{R}$ of real numbers

$$P(X=x) \Rightarrow \text{prob. of occurrence of event } E = \{s \in S \mid X(s) = x\}$$

$$F(x) = P(X=x) \Leftarrow \text{probability distribution function (PDF)}$$

determines distribution of the random variable $X$

• Binomial Random variable $\Rightarrow$ number of successes associated with given outcome

$$P(X=i) = C(n,i) (1-p)^{n-i} p^i$$
9.1.3 RANDOM VARIABLES AND EXPECTATION

- Geometric Random variable $\Rightarrow$ first success after $i$ trials
  
  \[ P(X=i) = (1-p)^{i-1} p \]

- N-th Moment
  
  \[ E[X^N] = \sum_x x^N P(X=i) \]

  N=0, checks whether the function is pdf or not,
  N=1 gives the mean or expectation of $X$,
  N=2, will be used for variance

  • **Properties:**
    
    \[ E[cX] = c \ E[X] \] where $c$ is a constant

  \[ E[X] = E[X_1] + E[X_2] + \ldots \ldots E[X_n] \text{ if } X = X_1 + X_2 + \ldots + X_n \]

  • It is also useful to know how much $X$ deviates from $E[X]$
9.1.3 RANDOM VARIABLES AND EXPECTATION

- The deviation of $X$ from $E[X]$ is formally defined in terms of variance: $V[X]=E[(X-E[X])^2]$.

- This can be simplified as: $V[X]=E[X^2]-(E[X])^2$.

- Conditional expectation $E[X|F]$ to be $E[X|F]=\sum_{s \in F} X(s)P(s|F)(s)P(s|F)$ or equivalently $E[X|F]=\sum_x xP(X=x|F)$.

Example: Consider rolling of two dice and let $X$ sum of two dice ie. $d1+d2$ Suppose $F$ is the event that $d1=2$. Since $d1$ and $d2$ are independent:

$$P((2,d2)|F)=\frac{1}{6}$$
Example ctd. Let’s find $E[X|F]$

$$(s)P(s|F)(s) = \frac{1}{6} \sum_{d2 \in \{1..6\}} (2+d2)$$

$$= (1/6)(3+4+5+6+7+8) = 5.5$$

• Let $x$ be a random variable on sample space $S$ and we partition $S$ into disjoint subsets $F_i, i=1..m$. Then we have:

$$E[X] = \sum_{i=1}^{m} E[X|F_i]P(F_i)$$

• We also have two following propositions

$$E[X] = \sum_{Y} E[X|Y=y]P(Y=y)$$

$$E[X|Y=y] = \sum_{X} xP(X=x|Y=y)$$
9.2 AVERAGE COMPLEXITY REVISITED

• When analyzing the complexity of algorithm the critical issue is often average complexity

Average complexity ⇒

\[ J_n \rightarrow \text{set of all inputs of size } n \text{ to a given algorithm} \]

\[ \tau(I) \rightarrow \# \text{ of basic operations performed by alg. on input } I. \ I \in J_n \]

Given \( J_n \) the average complexity:

\[ A(n) = E[\tau] \]
9.2.1 TECHNIQUES FOR COMPUTING AVERAGE COMPLEXITY

• Depending on the characteristics of the algorithm one or some combinations of formulas will be most applicable for A(n)

FORMULA 1

\[ A(n) = E[\tau] = \sum_{I \in J_n} \tau(I)P(I) \]

Rarely used since it is too cumbersome to examine each element in \( J_n \)

FORMULA 2

\[ A(n) = E[\tau] = \sum_{i=1}^{W(n)} iP(\tau=i) \]

\( P(\tau=i) \) ⇒ prob that alg performs exactly i basic operations
9.2.1 Techniques for Computing Average Complexity

**Formula 3**  
\[ A(n) = E[\tau] = \sum_{i=1}^{W(n)} P(\tau \geq i) \]

- \( P(\tau \geq i) \) \( \Rightarrow \) prob that alg performs at least i basic operations

**Formula 4**  
\[ A(n) = E[\tau] = \sum_{i=1}^{k} E[\tau_i] \]

**Formula 5**  
\[ A(n) = E[\tau] = \sum_{Y} E[\tau | Y = y] \cdot P(Y = y) \]

- When determining which formulation to use, use following techniques:
  1. Partitioning the algorithm
  2. Partitioning input space
  3. Recursion
9.3 AVERAGE COMPLEXITY OF LINEAR SEARCH

• Let the list consists of distinct elements and the search element \( X \) occurs with a probability \( p \)

• \( p_i \Rightarrow \) \( X \) is in the \( i \)th element in \( L[i] \)

\[
P_i = P(X = L[i] \mid X \text{ is in the list}) \quad P(X \text{ is in the list}) = \frac{1}{n}p
\]

• Use formula 2 to obtain:

\[
A(n) = 1\left(\frac{p}{n}\right) + 2\left(\frac{p}{n}\right) + \ldots + (n-1)\left(\frac{p}{n}\right) + n\left(\frac{p}{n} + 1-p\right)
\]

\[
= (1 - \frac{p}{2})n + \frac{p}{2}
\]

\( n \) comparisions when \( X \) is in \( L[n] \) or \( X \) is not in the list
9.3.1 AVERAGE COMPLEXITY OF LINEAR SEARCH WITH REPEATED ELEMENTS

- Determine $A(n,m)$ where $m$ is the number of distinct elements.

- $L[i]$ has equal probability of $1/m$ in $S$.

- The probability that $X$ does not occur in position $i$ $\Rightarrow (m-1)/m$.

- The probability that $X$ is not in the list $\Rightarrow ((m-1)/m)^n$.

- The probability that $X$ is in the list $\Rightarrow 1 - ((m-1)/m)^n$.

- $p_i \Rightarrow$ first occurrence of search element $X$ in position $i$. 
9.3.1 AVERAGE COMPLEXITY OF LINEAR SEARCH WITH REPEATED ELEMENTS

\[ p_i = \begin{cases} 
((m-1)/m)^{i-1}(1/m) & \text{if } 1 \leq i \leq n-1 \\
((m-1)/m)^n & \text{if } i = n 
\end{cases} \]

- Substitute formula 2 to obtain

\[ A(n,m) = \sum_{i=1}^{W(n)} i p_i = \sum_{i=1}^{n-1} i ((m-1)/m)^{i-1}(1/m) + ((m-1)/m)^n n \]

- By simplifying we obtain

\[ m(1 - ((m-1)/m)^n) + (m-1)/m)^{n-1} \]
9.3.1 AVERAGE COMPLEXITY OF INSERTION SORT

• Inputs of insertion sort is permutations of \{1,2...,n\}

• We have to partition the algorithm into n-1 stages

• The i^{th} stage consists of inserting the (i+1)^{th} element L[i+1] into its proper place in the list L[1],..,L[i]

\[ A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + .. + E[\tau_{n-1}] \]

where \( \tau_i \Rightarrow \) # of comparisons in ith stage

• Calculate \( E[\tau_i] \) with \[ E[\tau_i] = \sum_{j=1}^{i} jP(\tau=j) \]

  1. \( P(\tau=j)=1/{i+1}, \quad j=1,\ldots,i-1 \)
  2. \( P(\tau=i)=2/{i+1}, \quad j=1,\ldots,n-1 \)
9.3.1 AVERAGE COMPLEXITY OF INSERTION SORT

- Substitute 1 and 2 into previous formula and simplify to get:
  \[ E[\tau_i] = (\sum_{j=1}^{i} (j/i+1)) + i/i+1 = (i/2) + 1 - (1/i+1) \]

- Substitute this to our first formula to find \( A(n) \):
  \[ \sum_{i=1}^{n-1} ((i/2) + 1 - (1/i+1)) \]

\[ A(n) = (n-1)n/4 + n - H(n) \]

, where \( H(n) \) is harmonic series
\[ 1 + 1/2 + 1/3 + \ldots + 1/n \approx \ln n \]
Average Complexity of QuickSort
Average Complexity of QuickSort-1

Assumptions:

- The input lists L[1:n] to QuickSort are all permutations of 1,2,...,n, with each permutation being equally likely.

The QuickSort consists of two stages:

- We partition QuickSort into two stages, where the first stage is the call to `RearrangeAndPlace` and
  the second stage is the two recursive calls with input lists consisting of the sublists on either side of the proper placement of pivot element L[1].
Average Complexity of QuickSort-2

Average Complexity of Quicksort = Average complexity for RearrangeAndPlace + average complexity of RecursiveCall

- **RearrangeAndPlace** (Lets denote this as T1)
  Rearranges the list w.r.to previously chosen pivot element.
  So (n+1) comparisons are performed.

- **Recursive Calls** (Lets denote this as T1)
  Sort sublists L[1: i-1] and L[i+1: n] with n different choices of i, (ith element as pivot)
Formulation

\[ A(n) = E[T] = E[T1] + E[T2] = (n+1) + \frac{1}{n} \sum (A(i-1) + A(n-i)) \]

\[ \ldots \]

\[ = (n+1) + \frac{2}{n} (A(0)+A(1)+\ldots+A(n-1)) \]

\[ , A(0)=A(1)=0. \]
Average Complexity of QuickSort-4

\[ nA(n) = n(n+1) + 2(A(0)+A(1)+\ldots+A(n-1)) \]

**Substituting \( n-1 \) for \( n \) in the previous formula**

\[ (n-1)A(n-1) = n(n-1) + 2(A(0)+A(1)+\ldots+A(n-2)) \]

**Subtract**

\[ nA(n)-(n-1)A(n-1) = 2n + 2A(n-1) \]

\[ A(n)/(n+1) = A(n-1)/n + 2/(n+1) \]

**Let** \( t(n) = A(n)/(n+1) \)

\[ t(n) = t(n-1) + 2/(n+1), \ t(1)=0 \]

**Solve the above recurrence relation**...
Average Complexity of QuickSort-3

\[ t(n) = 2\left(\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right) \\
= 2H(n+1) - 3 \\
\sim 2\ln \]

Remember that \( t(n) = A(n)/(n+1) \)

\[ A(n) \sim 2n\ln \]

In particular, \textbf{QuickSort} exhibits \( O(n\log n) \) average behavior, which is the order optimal for a comparison based algorithm.
Reading Assignment (*)

- Average Complexity of MaxMin2
- Average Complexity of BinarySearch and BinSrchTreeSearch
Average Complexity of MaxMin2 (*)

Pseudo code

procedure MaxMin2 (L[1:n], MaxValue, MinValue)
MaxValue:=L[1]
MinValue:=L[1]
for i:=2 to n do
  if L[i]>MaxValue then
    MaxValue:=L[i]
  else
    if L[i]<MinValue then MinValue:=L[i] endif
  endif
endfor
end MaxMin2
Average Complexity of MaxMin2(2) (*)

Assumptions:

- The inputs permutations of 1, 2, ..., n, and that each permutation is equally likely.
- We already know that $B(n) = n-1$ and $W(n) = 2(n-1)$.

$(n-1)$ comparisons involving MaxValue are performed for any input permutation. An additional comparison involving MinValue is performed for each iteration of the loop in which MaxValue is not updated.
**Average Complexity of MaxMin2(3) (*)**

- **D**: random variable that denotes the number of times that MaxValue is updated.
- \( T = n-1 + (n-1-D) = 2n-2-D \)
- \( A(n) = E[T] = 2n-2-E[D] \)

We compute the expected number of updates \( E[D] \) by partitioning the input space by utilizing the r.v M.

\[
E[D] = \sum E[D|M=i] P(M=i) \quad \text{where} \quad P(M=i) = \frac{1}{n} \quad i=1,2,..,n
\]

\[
E[D|M=i] = \alpha(n-1) + \frac{1}{n}
\]

\( \alpha(n) = (1/n)(\alpha(n-1) + 1) + ((n-1)/n) \alpha(n-1) \)

\[
\alpha(n) = \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} = H(n)-1
\]

\( A(n) = 2n-2- \alpha(n) = 2n-H(n)-1 \quad A(n) = 2n - \ln n - 1 \quad \sim W(n) = O(n) \)
Average Complexity of BinarySearch (*)

Remember *BinarySearch* given in Ch.3, we choose the 3-brach comparison of the *do case* statement as the basic operation:

```plaintext
case
  :X < L[mid] : high:= mid-1
  :otherwise: low:= mid+1
endcase
```
Average Complexity of BinarySearch (2) (*)

Assumptions:

- **p**: the probability that the search element X is on the list.
- Given that X is **on the list** L[1:n]; we assume that it is equally likely to occur in any of n positions.
- Given that X is **not on the list**; we assume it is equally likely to occur in any of the (n+1) intervals.

X < L[1], L[1]<X<L[2], ...L[n-1]<X<L[n], X=L[n]
Average Complexity of BinarySearch (3) (*)

- the probability that X occurs on the list and is equal to any given element $L[i] : \frac{p}{n}$

- the probability that X does not occur in any of the n+1 intervals : $\frac{1-p}{n+1}$
Average Complexity of BinarySearch (4) (*)

**Reminder** *(properties of Binary Search Tree)*

- **IPL**: internal path length, the sum of the lengths of the paths from root to the internal nodes as the internal nodes vary over the entire tree.
- **LPL**: leaf path length, defined similarly.

Note: length of a path from the root to a node at level \( i \):

\[
i \]

(from Ch.7, proposition 7.3.6)

\[
IPL(T) = LPL(T) - 2I,
\]

where \( I \) is the number of internal nodes.
Average Complexity of Binary Search (5) (*)

Thus \( \text{IPL}(T) = \text{LPL}(T) - 2n \)

So, \( A(n) = \frac{p}{n} (\text{IPL}(T)+n) + \frac{1-p}{(n+1)}\text{LPL}(T) \)

\[= \frac{p}{n} (\text{LPL}(T)-n) + \frac{1-p}{(n+1)}\text{LPL}(T) \]

\[= \left( \frac{p}{n} + \frac{1-p}{(n+1)} \right) \text{LPL}(T) - p \]

(from Ch.7, proposition 7.3.7)

\( \text{LPL}(T) = L \text{ floor} (\log L) + 2(L-\text{ power}(2,\log L)) \),

if \( T \) is a 2-tree and is full at the second-deepest level.

\( \text{LPL}(T) \geq L \text{ ceiling} (\log L) = \text{ceiling} ( (n+1)\log(n+1) ) \)

\( A(n) \geq \left( \frac{p}{n} + \frac{1-p}{(n+1)} \right) ( (n+1)\log(n+1) ) - p \)
Average Complexity of BinarySearch & BinSrchTreeSearch (*)

- The lower bound estimate for $A(n)$
  \[ A(n) \sim W(n) = \log(n+1) \]

- For BinSrchTreeSearch,
  \[ A(n) = \text{LPL}(T) \left( \frac{(1+ (p/n))}{(n+1)} \right) - p \]
  
  ...  
  \[ A(n) \sim \Omega(\log n) \], apply thm 7.3.7