Lower Bounds

Introduction

- After a problem has been solved, new and better algorithms and implementations may still be found.
- For example, Dijkstra's algorithm for finding the shortest paths in a graph began in 1959 with an \(O(n^2)\) implementation based on an unsorted array, then progressed through \(O(m \log n)\) and \(O(m \log 2 + m/n)\) implementations to the current best, \(O(n \log n + m)\) using Fibonacci heap, in 1983.
- The questions are:
  - Will the algorithms of today be superseded by more efficient ones in the future?
  - For which problems is it worth trying to find better algorithms?
- Such questions can only be answered by a theorem of the form
  "All algorithms for problem \(P\) have complexity \(T(n) \geq f(n)\)"
- A function \(f(n)\) appearing in such a theorem is called a lower bound on the complexity of \(P\).
- If we could find an algorithm of complexity \(f(n)\) for \(P\), then that algorithm is optimal (it cannot be bettered).

Adversary Bound

- Consider the problem of summing the array element \(a[1] \ldots [n]\). Every element must be examined, therefore any algorithm for this problem is \(O(n)\).
- This argument is called an input lower bound.
- Another way to express this argument is to use an adversary (i.e. someone who watches the algorithm runs, and tries to break it).
- For the summing problem,
  - Suppose some algorithm does not examine every element.
  - The adversary watches and, when the algorithm terminates, it moves in and changes the value of some unexamined element.
  - The algorithm is run again, and it gives the same result as before, since it does not see the change. But the true result has changed, so the algorithm is incorrect.
  - It follows that any correct algorithm examines every element, and so is \(O(n)\).

LOWER BOUND FOR A SORTING PROBLEM

In the sorting problem you do comparison between pairs of elements plus exchanges. Count the number of comparisons one may need to do in the worst case. Initially we do not know anything. So, all orderings are possible.
Then we start comparing pairs of elements - one by one. We do not have to compare all pairs, because some of them are inferred by the transitivity of the "comparable" property (if a<b, and b<c we can infer a<c). [Ignore actual data movements.] However, the comparison steps are actually navigating along a binary tree for each decision (a<b, or a>b).

The number of leaves in the tree is N! (all possible permutations of the N-elements list). Hence the depth of the tree is bound by \( \log_2 N! = O(N \log N) \). Since, the number of steps of the decision algorithms is bound by the maximum depth of this (decision) tree any algorithm’s worst-case complexity is at best \( O(N \log N) \).

Hence any sorting algorithm based on comparison operations cannot have a better complexity than \( O(N \log N) \) on an average.

Note 1: An algorithm actually may be lucky on an input and finish in less number of steps, note the tree.

Note 2: You may, of course, develop algorithms with worse complexity – insertion sort takes \( O(N^2) \). \( O(N \log N) \) is the best worst-case complexity you could expect from any comparison-based sort algorithm!

This is called the information theoretic lower bound, not of a particular algorithm - it is valid over all possible comparison-based algorithms solving the sorting problem.

Any algorithm that has the complexity same as this bound is called an optimal algorithm, e.g. merge-sort.

**LOWER BOUNDS ON SELECTION PROBLEM**

- Designing algorithm against an adversary
  
  &lt;&lt; An algorithm playing Information game against an adversary &gt;&gt;
- The algorithm wants to get as much Information as possible in order to get as much work done as effective as possible.
  - The adversary wants to give as least Information as possible to give the algorithm the worst case.
  - The rule of the game is Consistency.
  - The adversary can trick but cannot cause inconsistency in the given information.
- e.g. your algorithm needs to guess a date (a month and day) and your adversary gives yes/no answers.

**Strategy for Designing against an adversary**

- Assume a strong adversary!
  - the adversary will give as least information as possible
- Choose questions (or operations) as balance as possible
  - e.g. for comparison of two keys, \( x > y \)
  - Yes: \( x > y \) and
  - No: \( x \not> y \)
  - should provide about the same amount information

---

2
The selection problem:
<Ranking elements of a set in nondecreasing order, find an element rank K >
e.g. Finding max and min

• Find an element with rank k
  ➔ in an array E using indexes 1 through n
  ➔ the elements in E are assumed to be unsorted
  ➔ where 1 <= k <= n

• Finding an element with rank k is equivalent to answering the question:
  ➔ If the array were sorted in nondecreasing order which element would be in E[k]?
  • The largest key (called max) should be k = n
  • The smallest key (called min) should be k = 1
  • The median key should be k = Ceiling[n/2]

Finding min, finding max,
Finding min in an unsorted array of n elements
  ➔ require at least n-1 comparisons
  • Finding max in an unsorted array of n elements
  ➔ require at least n-1 comparisons
  • Now we want to find both min and max
  ➔ can we do better than 2(n-1) ?
  ➔ What is the lower bound?

• Theorem: Any algorithm to find both min and max of n keys by comparison of keys must do at least 3n/2 – 2 key comparisons in the worst case.

Proof by adversary arguments and units of information
• To know that a key v is max, an algorithm must know that every key other than v has lost some comparison
  ➔ To know that a key u is min, an algorithm must know that every key other than u has won some comparison.
  • If we count each win as one unit of information and each loss as one unit of information
  ➔ Then an algorithm must have at least 2(n-1) units of information for finding both min and max
  • We need to determine how many comparison are required (in the worst case) to get total 2(n-1) units of information.

  ➔ The adversary to give us the worst case will provide as few information as possible.
    Our strategy to gain as much information
  • Our algorithm can do at most n/2 comparisons of previously unseen keys
  ➔ suppose for the moment that n is even
  ➔ each of these comparison give us 2 units of information
  ➔ now we have n units of information

  • Our algorithm need total 2(n-1) = 2n – 2, so now we need n – 2 additional units of information
for each other comparison we gain at most one unit of information
so we need at least \( n - 2 \) additional comparisons

• In total, our algorithm requires at least \( n/2 + n - 2 \) comparisons. For \( n \) is odd, \( 3n/2 - 3/2 \) comparisons are needed. QED

• Now we know the lower bound (in the worst case)
• Can we design an algorithm to reach the lower bound?
General Notes on Algorithm Design and Analysis

Types of Algorithms

- Divide and Conquer Algorithms
- Greedy Algorithms
- Dynamic Programming Algorithms
- Backtracking

1- Divide and Conquer Algorithms

Example: MergeSort, QuickSort, binary-search.
Recursive calls on divided parts of the input combine the results from those calls, and then return.

The complexity of the Divide and Conquer Algorithms is characterized by recurrence relations.

A special case of recurrence equation becomes important in analysis (Master Theorem)
\[ T(N) = a \cdot T(N/b) + (N^k), \] where \(a>1 \) and \(b>1\).
Solution:

- [case 1] \( (N^{\log_b a}) \), for \(a>b^k\);
- [case 2] \((N^k \log(N))\), for \(a = b^k\);
- [case 3] \((N^k)\), for \(a < b^k\)

Example:
Binary Search Algorithm (Time complexity is \(T(n) = T(n/2) + 1\))

NOTES ON RECURRENCE EQUATION

We solve recurrence equations often in analyzing complexity of algorithms, circuits, and such other cases.

A homogeneous recurrence equation is written as:
\[ a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0. \]

Solution technique:

Step 1: Set up a corresponding Characteristic Equation:
\[ a_0 x^n + a_1 x^{(n-1)} + \ldots + a_k x^{(n-k)} = 0, \]
\[ x^{(n-k)} [a_0 x^k + a_1 x^{(k-1)} + \ldots + a_k] = 0, \]
\[ a_0x^k + a_1x^{k-1} + \ldots + a_k = 0 \ [\text{for} \ x \neq 0] \]

**Step 2:** Solve the characteristic equation as a polynomial equation. Say, the real roots are \( r_1, r_2, \ldots, r_k \). Note that there are \( k \) solutions for \( k \)-th order polynomial equation.

**Step 3:** The general solution for the original recurrence equation is:

\[ t_n = \sum_{i=1}^{k} c_i r_i^n \]

**Step 4:** Using initial conditions (if available) solve for the coefficients in above equation in order to find the particular solution.

**Example 1:**

\[ t_n - 3t_{n-1} - 4t_{n-2} = 0, \text{ for } n \geq 2. \ {\text{Initial condition:}} \ t_0 = 0, \ t_1 = 1 \]

Characteristic equation: \( x^n - 3x^{(n-1)} - 4x^{(n-2)} = 0, \)

Or, \( x^{(n-2)} [x^2 - 3x - 4] = 0, \)
Or, \( x^2 - 3x - 4 = 0, \)
Or, \( x^2 + x - 4x - 4 = 0, \)
Or, \( x(x+1) - 4(x+1) = 0, \)
Or, \( (x+1)(x-4) = 0, \)

Therefore, roots are, \( x = -1, 4. \)

So, the general solution of the given recurrence equation is:

\[ t_n = c_1(-1)^n + c_2(4^n) \]

Use \( t_0 = c_1 + c_2 = 0, \) and \( t_1 = -c_1 + 4c_2 = 1. \) [Note that we need two initial conditions for two coefficients.]

Solve for \( c_1 \) and \( c_2, \)
\( c_1 = -(1/5), \ c_2 = (1/5). \)

So, the particular solution is:

\[ t_n = (1/5)[4^n - (-1)^n] = \Theta(4^n) \]

**End example 1.**
Inhomogeneous recurrence equation

\[ a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = b^n p(n), \]  
where \( b \) is a constant and \( p(n) \) is a polynomial of order \( n \).

Solution technique:

Step 0: Homogenize the given equation to an equivalent homogeneous recurrence equation form.

Step 1 through 3 (or 4) are the same as in the case of solving homogeneous recurrence equation.

Example 2:

\[ t_n - 2t_{n-1} = 3^n. \]  
[Note, this is a special case with \( p(n) = 1 \), polynomial of 0-th order, and there is no initial condition – so we get the general solution only.]

Transform with \( n \rightarrow n+1 \):

\[ t_{n+1} - 2t_n = 3^{n+1} \]  
…………… (1)

Multiply original equation with 3 on both sides:

\[ 3t_n - 6t_{n-1} = 3^{n+1} \]  
…………… (2)

Subtract Eqn (2) from Eqn (1):

\[ t_{n+1} - 5t_n + 6t_{n-1} = 0, \]  
this is a homogeneous recurrence equation which is equivalent to the given inhomogeneous equation.

Characteristic equation: \( x^2 - 5x + 6 = 0. \)
Which is \((x-2)(x-3) = 0.\)

So, roots are \( x = 2, 3. \)

General solution of the given recurrence equation is:

\[ t_n = c_1 * (2^n) + c_2 * (3^n) = \Theta(3^n) \]

End example 2.

Homogenizing may need multiple steps. See the following example:

Example 3:

\[ t_n - 2t_{n-1} = n \]

So, \( t_{n+1} - 2t_n = n + 1 \)

Subtracting the former (given) eqn from the latter eqn,

\[ t_{n+1} - 3t_n + 2t_{n-1} = 1 \]
Still this is not a homogeneous eqn. We need second stage of homogenizing, substitute \( n \rightarrow n+1 \)

\[ t_{n+2} - 3t_{n+1} + 2t_n = 1 \]

Subtract once again,

\[ t_{n+2} - 4t_{n+1} + 5t_n - 2t_{n-1} = 0 \]

Now it is a homogeneous recurrence equation and one can solve it in the usual way.

**End example 3.**

**An important special case:**

If the characteristic eqn. is like

\((x-2)(x-3)^2 = 0, \quad [CAN\ YOU\ CONSTRUCT\ THE\ ORIGINAL\ HOMOGENEOUS\ RECURRENCE\ EQN\ BACK?]\)

The roots are \( x = 2, 3, 3 \) for a polynomial eqn. of order 3.

The general solution for the given recurrence eqn. is then,

\[ t_n = c_1*(2^n) + c_2*(3^n) + c_3*n*(3^n) \]

Note the additional \( n \) in the third term.

If the roots were \( x = 3, 3, 3 \), then

The general solution would be

\[ t_n = c_1*(3^n) + c_2*n*(3^n) + c_3*(n^2)*(3^n), \]

* A special type of recurrence equation that is frequently encountered in algorithms' analyses

**(Master Theorem)**

\( T(n) = a \cdot T(n / b) + c \cdot n^i, \) for some constant integer \( i \), and constants of coefficients \( a \) and \( c \).

Three cases:

- \( a = b^i \), the solution is \( T(n) = O(n^i \log_b n) \);
- \( a > b^i \), the solution is \( T(n) = O(a^{\log_b a}) \);
- \( a < b^i \), the solution is \( T(n) = O(n^i) \);
2- Dynamic Programming Algorithms
In case the divide and conquer strategy can divide the problem at a very small level, and there are repetition of some calculation over some components, then one can apply a bottom up approach: calculate the smaller components first and then keep combining them until the highest level of the problem is solved.

Example1: Fibonacci series.
Draw the recursion tree of Fibonacci-series calculation, you will see example of such repetitive calculations
\[ f(n) = f(n-1) + f(n-2), \text{ for } n>1; \quad \text{and } f(n)=1 \text{ otherwise} \]

\begin{align*}
\text{fib(n) calculation} \\
n & = 1, 2, 3, 4, 5 \\
\text{fib} & = 1, 2, 3, 5, 8
\end{align*}

Implementation strategies for this problem:

* Recursive approach
\[
\text{fib(n)} \\
\text{if } (n<=1) \text{ return } 1; \\
\text{else return (fib(n-1) + fib(n-2)).}
\]

Time complexity: exponential, \(O(k^n)\) for some \(k>1.0\)

* Iterative approach
\[
\text{fib(n)} \\
\text{fib(0) = fib(1) = 1; } \\
\text{for i=2 through n do} \\
\text{fib(i) = fib(i-1) + fib(i-2); } \\
\text{end for;} \\
\text{return fib(n).}
\]

Time complexity: \(O(n)\), Space complexity: \(O(n)\)

* Dynamic programming approach
\[
\text{DP-fib(n)} \\
\text{if } (n<=1) \text{ return } 1; \\
\text{int last=1, last2last=1, result=1;} \\
\text{for i=2 through n do} \\
\text{result = last + last2last;} \\
\text{last2last = last;} \\
\text{last=result} \\
\text{end for;} \\
\text{return result.}
\]

Time complexity: \(O(n)\), Space complexity: \(O(1)\)
The recursive call recalculates fib(1) 5 times, fib(2) 3 times, fib(3) 2 times - in fib(5) calculation. The complexity is exponential.
In iterative calculation we avoid repetition by storing the needed values in variables - complexity of order n.
Dynamic Programming approach consumes more memory to store the result of calculations of lower levels for the purpose of calculating the next higher level. They are typically stored in a table.

Example 2: Ordering of Matrix-chain Multiplications

ABCD a chain of matrices to be multiplied, where A (5x1), B(1x4), C(4x3), and D(3x6). Resulting matrix would be of size (5x6).

# scalar / integer multiplications for (BC), is 1.4.3, and the resulting matrix’s dimension is (1x3), 1 column and 3 rows.
There are multiple ways to order the multiplication: (A(BC))D, A(B(CD)), (AB)(CD), ((AB)C)D, and A((BC)D). Resulting matrix would be the same but the efficiency of calculation would vary drastically.
Efficiency depends on #scalar multiplications. In the case of (A(BC))D, it is = 1.4.3 + 5.1.3 + 5.3.6 = 117. In the case (AB)CD, it is = 4.3.6 + 1.4.6 + 5.1.6 = 126.
Our problem here is to find the best such ordering. An exhaustive search is too expensive - Catalan number involving n!
For a sequence A1...(Aleft....Aright)...An, we want to find optimal break point for the parenthesized sequence.
Calculate for ( right-left+1) number of cases and find minimum: min{(Aleft...Ai)(Ai+1... Aright), with left ( i < right)}, or
M(left, right) = min{M(left, i) + M(i+1, right) + rowleft .coli .colright, with left ( i < right)}. Start the calculation at the lowest level with two matrices, AB, BC, CD etc. Then calculate for triplets, ABC, BCD etc. And so on..

Implementation strategies for the problem

* Matrix-chain Recursive algorithm

Algorithm M(left, right)
    If left > right return 0
    Else
        return min{M(left, i) + M(i+1, right) + rowleft .coli .colright, for left<=i<right};
    End algorithm.

Driver: call M(1, n).
**Matrix-chain DP-algorithm**

For all 1 ≤ j ≤ n do $M[i][j] = 0$;

For size = 1 to n do

   For left = 1 to n-size+1 do

      right = left+size-1;

      $M[left][right] = \infty$;

      For i = left to right do

         $x = M(left, i) + M(i+1, right) + rowleft \cdot coli \cdot colright$;

         if $x < \text{min}$ then

            $M(left, right) = x$

Ordering of Matrix-chain Multiplications (Example)

A1 (5x3), A2 (3x1), A3 (1x4), A4 (4x6).

$c(1, 1) = c(2, 2) = c(3, 3) = c(4, 4) = 0$
$c(1, 2) = c(1, 1) + c(2, 2) + 5 \cdot 3 = 0 + 0 + 15.$
$c(1, 3) = \min\{ \min_{i=1} I=1 c(1, i) + c(2, 3) + 5 \cdot 3 \cdot 4, I=2 c(1, 2) + c(3, 3) + 5 \cdot 1 \cdot 4 \}$
   = $\min\{72, 35\} = 35(2)$
$c(1, 4) = \min\{ \min_{i=1} I=1 c(1, i) + c(2, 4) + 5 \cdot 3 \cdot 6, I=2 c(1, 2) + c(3, 4) + 5 \cdot 1 \cdot 6,$
   $I=3 c(1, 3) + c(4, 4) + 5 \cdot 4 \cdot 6 \}$
   = $\min\{132, 69, 155\} = 69(2)$

69 comes from the break-point i=2: (A1.A2)(A3.A4)
You may need to recursively break the sub-parts too, by looking at which value of i gave the min value at that stage, e.g., for (1, 3) it was i=2: (A1.A2)A3

<table>
<thead>
<tr>
<th>i</th>
<th>j=1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>15</td>
<td>35(2)</td>
<td>69(2)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Calculation goes diagonally.

**Triplets**

**Pairs**
Calculation goes diagonally.

Computing the actual break points

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>-</td>
<td>-</td>
<td>(2)</td>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>-</td>
<td>-</td>
<td>(3)</td>
<td>(3)</td>
<td>(4)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>-</td>
<td>-</td>
<td>(4)</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ABCDEF -> (ABC)(DEF) -> ((AB)C) (D(EF))

**Ordering of Matrix-chain Multiplications (DP Algorithm)**

Algorithm optMult: returns cost matrix m and tracking matrix lastChange.
(1) for (left=1 through n) do  cost[left][left] = 0;
(2) for k=2 through <= n do  // k is the size of the sub-chain
   for left=1 through (n-k+1) do
      right=left+k-1;
      //break is between I and I+1
      for I=left through (right-1) do
         cst=cost[left][I]+cost[I+1][right] +
         row[left]*col[I]*col[right];
      if (cst<cost[left][right]) then
         cost[left][right] = cst;
      lastChange[left][right]=I;

**3- Backtracking Algorithms**

Algorithm Unknown (level, array A[]) 
if (level == 4) then print A;
else
   A[level] =0;
   Unknown (level+1, A);
   A[level] =1;
Unknown (level+1, A);
End algorithm.

Driver: Unknown (1, A[1..3])

**What does the algorithm do?**

This algorithm prints (000), (001), (010), (011),..., when called as Unknown(1, A).
Draw recursion tree: this is an exhaustive Backtracking (BT) algorithm.

**4- Greedy Algorithms**

**GRAPH ALGORITHMS**

**Definitions (background):**
Graph: nodes/vertices, and edges/arcs as pairs of nodes. \( \{V, E\} e_{12}=(v_1, v_2, 12) \)
Directed graph: edges are ordered pairs of nodes.
Weighted graph: each edge (directed/undirected) has a weight.
Path between a pair of nodes: sequence of edges with those two nodes at the two ends.
Simple path: covers no node in it twice.
Loop: a path with the same start and end node.
Path length: number of edges in it.
Path weight: total wt of all edges in it.
Connected graph: there exists a path between every node, no node is disconnected.
Complete graph: edge between every pair of nodes [NUMBER OF EDGES?].
Acyclic graph: a graph with no cycles.
Etc.

Graphs are one of the most used models of real-life problems for computer-solutions.

Representations: visual pictures are not useful data structures for storing a graph! Adjacency list (link list of directly connected nodes for each node), and matrix are two representations. The second one is less efficient but easy to use, while the first one is good for sparse graph (sparsely distributed edges, less connected).
Problem size includes both \(|V| \) (number of nodes \( N \)), and \(|E| \) (number of edges, in the worst case \( O(N^2) \) for complete graph, but not “fair” for a sparse graph). Also, with matrix representation \(|E| \) is always \( N^2 \), because one has to go over all pairs of nodes to check which ones are in \( E \).

**Complexity**

Algorithm 1:
For each node in \( V \) do
- Steps- // \((N)\)
Algorithm 2:
For each node in V do
  For each edge in E do
    -Steps- // ((N.|E|)

Algorithm 3:
For each node v in V do
  For each edge of v do
    -Steps- // (( |E| ) with adjacency list

Algorithm 4:
For each node v in V do
  For each node w adjacent to v do
    -Steps- // (( |E| )

Algorithm 5:
For each node v in V do
  -steps-
  For each edge of v do
    -Steps- // ((N+|E|) or (( max{N,|E|} )

---

**SHORTEST-PATH (ON UN-WEIGHTED) GRAPH**

Path length = number of edges on a path.

Compute shortest path-length from a given source node to all nodes on the graph.

Strategy: starting with the source as the "current-nodes," in each of the iteration expand children (adjacent nodes) of "current-nodes." Assign the iteration# as the shortest path length to each expanded node, if the value is not already assigned. Also, assign to each child, its parent node-id in this expansion tree (to retract the shortest path if necessary).

**Algorithm naïve-shortest-path**

distance-of source = 0; //source to source distance
distance-of all other nodes = infinity; // (N)

// let the current distance from source be called curdist
for curdist = 0 through N-1 do
  for each vertex v do // (N2)
    if (v is not yet “visited” && distance-of v is curdist) then
      // the first check is needed because a graph may have cycles– to
      // avoid looping, the second check is needed for expanding only the
      // nodes at current level curdist,
// not all the nodes, only adjacent ones
for each w adjacent to v do
  // ((N |E|), |E| from the two inner for-loops,  
  // N from the outermost loop
  if distance-of w is yet unassigned then
    distance-of w = curdist +1;
    path-of w (from source) = v;
    // w’s parent is v
    end if;
  end for;
end if;
mark v as “visited”;
// because its children have been just expanded
end for;
end for;

End algorithm.

This is a breadth-first traversal, nodes are expanded as increasing distances from the source: 0, then 1, then 2, etc. [DRAW EXPANSION LIST AS A TREE.]

Exercise: HOW TO FIND SHORTEST PATH backward FROM THE PATH-OF-W VALUES?
Write an Algorithm
  Find-Shortest-Path (vertex w, array path-of[])

Complexity: ((N2 + N |E|) from the for-loops. [((N2) in the text]

Once again,
a better idea is to have the children being pushed at the back of a queue.

Algorithm q-based-shortest-path
  distance-of source s = 0;
 enqueue only s in Q;  // ((1), no loop, constant-time

  while (Q is not empty) do  // each node goes to Q only once: ((N)
    v = dequeue Q;
    for each vertex w adjacent to v do  // ((|E|)
      if distance-of w is yet unassigned then
        distance-of w = distance-of v + 1;
        path-of w = v;
        enqueue w in Q;
      end if;
    end for;
  end while;
End algorithm.
Complexity: ($|E| + N$), by a similar analysis as that of the previous queue-based algorithm.

Exercise: Implement this algorithm

**DIJKSTRA'S ALGORITHM FOR SHORTEST PATH ON WEIGHTED GRAPHS**

Previously weight (or cost) on each path was $= 1$. Now any positive value ($>0$) exists on each of them.

We will ignore negative weight, because the problem of finding shortest path is not well defined when negative edge is allowed (you can loop infinitely and keep reducing the path weight!).

Strategy: from the set of "unfinished" nodes, pick up the one whose path-weight (from the source) is shortest in the set, declare this node to be “finished” (shortest path is now found for this node), and try to update the path-weights to the other unfinished nodes adjacent to the just-finished node, using the direct edges from this node.

Each node other than the source is initialized with a variable called distance (shortest from the source) $= \infty$, and is marked as unfinished.

**Algorithm Dijkstra**

1. $s$.distance $= 0$; (all-other-nodes).distance $= \infty$; mark $s$ as “finished”; // "known" in the text book
2. **while** there is a node marked as unfinished do //($|N|$)
   1. cur-node $= \text{node from unfinished list with the smallest distance}$; // loop on unfinished
   2. mark cur-node as “finished”;
   3. **for** each node $w$ adjacent to cur-node do //($|E|$)
      1. if $w$ is “unfinished” then
         1. if (cur-node.distance + Dcw $< w$.distance) then
             1. $w$.distance $= \text{cur-node.distance + Dcw}$;
             1. // Dcw is the direct-distance from cur-node $c$ to $w$
             1. $w$.previous $= \text{cur-node}$;
             1. // parent on the shortest path
         1. end if (both);
      1. end for;
   3. end while;

**End algorithm.**

Complexity: the while loop runs $N-2$ times: $((|N|)$, find-minimum-distance may run another $(|N|)$ within it, thus the complexity is $(|N|2)$; for-loop as usual runs for $|E|$ times including the outside while loop.

Grand total: $((|E| + N2) = (|N|2)$, as $|E|$ is always $(O(N2)$.

12
Findmin complexity could be reduced to \((N\log N)\) from \((N^2)\), by using a heap data-structure, but as distance values get changed the heap needs to be kept reorganized, thus, increasing the for-loop's complexity to \((|E|\log N)\). Grand total: \((|E|\log N + N\log N) = (|E|\log N)\), as \(|E|\) is always \(N\) in a connected graph.

**Proof of correctness (Why Dijkstra’s algorithm works)**

**By induction:** Suppose it works up to the \(k\)-th iteration. This means that for \(k\) nodes the corresponding shortest paths from source have been already identified (say, set \(F\)). Rest of the nodes have their distances possibly reduced, but shortest distances from source are not found for them yet (say, set \(U\)).

Picking up the shortest-current-distance node (say, \(p\)) from the set \(U\) makes \(p\) to be the one for which the shortest distance is just found, so, should move into \(F\). If this were not true, then path via some other node in \(U\) could improve its path. But all of them have a longer (or equal) path than that for \(p\) currently, how could they improve it for someone else. And, all the nodes in \(F\) already had their chance to improve the path for \(p\) in the previous iterations; so, no further improvement is possible via anyone of them. That implies, the shortest path for \(p\) has been found.

Next, why do we use \(p\), to improve paths for the nodes in the set \(U\)-{\(p\)}? All other nodes in \(F\) had had their chance to improve paths for the nodes in \(U\), so they cannot be useful anymore. And shortest paths for nodes in \(U\)-{\(p\)} have not been found yet, so how can they find shortest paths for others. Even though one of them (say, \(u(p)\)) may improve paths for others in their set \(U\), that path may have to be changed later when path to such a node(u) gets improved further. Hence, \(p\) is the only candidate, which should improve path for others in the next iteration.

Thus, in the next iteration we have \(k+1\) nodes (including node \(p\)) for which shortest paths have been found (from the \(k\)-node stage), and in the next iteration we are ready to declare our \((k+2)\)-th node to be included in the set \(F\). This will conclude this inductive proof, when we find an induction base-case.

**Induction base:** Shortest distance for the source itself must have been found – no other node can improve it.

**Exercise:** Prove the unweighted shortest path q-based algorithm’s correctness.

**Greedy Strategy**

Dijkstra’s algorithm works with greedy strategy: pick up the best of something to work with at every stage. Doesn’t care the global implication: what is best now - may proved to be a bad choice in the future. In case of single-source-shortest-path Dijkstra’s algorithm is provably correct and the strategy has worked. However, greedy strategy is not always so lucky.
Maximum Flow Problem
A weighted diagraph, weights mean capacity of flow (traffic, fluid, etc.) on an edge. Given also are source and sink nodes. Find out the maximum flow possible from source to sink including all paths between them, with a constraint: for any node the total inflow should be equal to the outflow from it.

A strategy: find a path from the source to the sink, subtract the minimum weight on any edge on the path from the weights of all edges on that path, eliminate any edge with zero weight, repeat these steps until no more path from source to sink exists. On every iteration, keep track of the path, on a separate graph, with the weight that is being subtracted on each of its edges (max flow). Calculate max-flow between the source and the sink from this second graph.

Depth-first search on graphs and its use in some algorithms
Recursion stack implements depth-first traversal. So typically dfs are implemented as recursive algorithm.

Graph-traversal needs additional check: in case a node is already traversed - since there can be loops (which are not there in a tree by definition).

Algorithm dfs(v)
mark node v as visited;
operate on v (e.g., print);
for each node w adjacent to node v do
  if w is not marked as visited then
dfs(w);  // an iterative algorithm
end for;
End algorithm.

Starter algorithm
on a given graph call dfs(root);
End starter.

This algorithm will not be complete for not-connected graph, or if directed but not strongly-connected graph.

One needs to restart it again and again until all nodes are marked. So the starter algorithm looks like:

repeat until all nodes are marked
  pick up next unmarked node s;
dfs(s);
end repeat;
Checking and finding unmarked node might be unduly expensive.

DFS produces a spanning-tree (or forest, for a directed graph) of the graph: dfs-spanning-tree. This is utilized in solving the following problems.

Algorithm Iterative-DFS
let start node be v;
initialize stack S with Push(S, v);

while stack S not empty do
    node r = Pop(S);
    mark r visited;
    for each node w adjacent to r do
        if w is not marked Push(S, w)
    end for;
end while;
End Algorithm

Complexity O(|E|) from the two loops.

Euler circuit
Problems: (1) Follow the edges of a graph (undirected) without lifting the pen to cover all edges only once, (2) problem 1, and have the start and end points both be the same node.

Example:

Necessary and sufficient conditions for existence of a solution:

Problem 2. Graph must be connected and every vertex must have even number of edges (for the pen to come in and then go out).

Problem 1. Relax the previous condition by allowing exactly two nodes to have odd number of edges, which must be the start and the end nodes.

Checking the conditions take linear time (one pass over each node ((N)).

Algorithm for drawing Euler circuit for problem 2 (i.e., ordering the nodes as they are traversed), knowing that there exists one:
Step 1: Beginning from the start node traverse nodes DFS-manner, eliminate edges as they are traversed (because each edge can be used only once), until stuck at a node because no more edge is left at this node
Step 2: If all edges are not yet covered, pick up the first unfinished node X in the last path that has un-traversed edges left, and starting from X traverse nodes DFS-manner, eliminate edges as they are traversed, until X is reached back and is finished. Insert this new path in the previous path replacing X in the latter.
Step 3: Keep doing step 2 until all edges are covered.
End algorithm-sketch.

Complexity: O(N + |E|), because all edges are covered only once, (but note that some nodes may be covered multiple times).

DFS on directed graph

Could create forest of traversed trees

Until no unmarked nodes remain do
    pick up a unmarked node v;
    mark v as visited;
    for each adjacent node w to v do
        if w is not visited
            traverse w recursively;
    end for;
end until-loop.

Finding Strongly-connected components (SCC) in a Directed graph

Strongly-connected component: a subgraph in which there is a path between every pair of nodes

The whole graph could be strongly connected.
Note: Only a single node could form its own SCC.

SCC creates partition of nodes in a graph: every node is in a SCC, & no node is outside all SCC’s.

A sub-graph of a SCC could be an SCC itself, although not necessarily.

We are interested in finding the set of all SCC’s in a directed graph

Strategy for finding SCC’s of a graph G:
Traverse G using DFS to create spanning forest of the graph;
Number the nodes in the post-order traversal of the spanning-forest;
Create Gr by reversing the directions of arcs in G;
Traverse using DFS the nodes of Gr in a decreasing order of the numbers,
DFS spanning-forest from this second traversal creates the set of SCC’s for G.

Intuition: First dfs-spanning-trees create a partition for forward paths (v to w); The second dfs-trees guarantee reverse path (w to v); v to x double-directed paths and x to w double-directed-paths
guarantee double-directed path between v and w; The second dfs-trees generation by using the first one’s result (numbering) guarantees catching all the SCC’s.

To prove:
Theorem 1: Each Spanning Tree ST(Gr) is a strongly connected component of G.

Theorem 2 (equivalent): for every pair of nodes (v, w) in any Spanning Tree ST(Gr) there exists a path v -> w and a path w -> v.

Theorem 3 (equivalent): above can be proved if we prove for any pair of nodes (v, w) in ST(Gr) with the root r, there exists a path from v -> r and from r -> w in G.

Theorem 4 (equivalent): above can be proved if we prove that for any node v in ST(Gr) with the root r, there exists a path r -> v and a path v -> r in G.

Lemma 1: every spanning tree ST(Gr) is a sub-tree of a spanning tree ST(G).
Proof: by observation that backarcs between the spanning trees ST(G) are in “forward” direction only. So when directions of the graph is reversed in Gr, and you start DFS traversal from Right to Left (in reverse post-order traversal numbering of ST(G)), there is no way you can get to the next tree from the current one (tree) – they are disconnected with Gr’s directed arcs. So, each ST(Gr) is a sub-tree of the corresponding ST(G). QED.

Lemma 2: root r of every ST(Gr) has a higher post-traversal index (in ST(G)) than any node n of that ST(Gr). 
Proof: Trivial. Traversal on Gr was done in that order. QED.

Lemma 3: for any node v in an ST(Gr) whose root is r, there exist a path v -> r in G.
Proof: Trivial. Since there exists a path in r -> v in Gr, there exist a path v -> r in G.

Lemma 4: for any node v in an ST(Gr) whose root is r, there exist a path r -> v in G.
Proof: Both r and v belongs to the same ST(G) by lemma 1. And, post-ord-index(r) > post-ord-index(v) by lemma 2. Thus, all the work of processing r was completed after the work of processing v during DF-traversal of G. Since there is a path from v -> r in G by lemma 3, v must be descendant of r in ST(G) – otherwise v would finish after r. In other words, there exists a path r -> v as indicated in ST(G) they both belong to. QED. [Example from book: path exists F -> B in G as suggested in ST(Gr), yet order(B)=6 > order(F)=5. So, there must be a path B -> F in ST(G).]

Lemma 3 and 4 prove version (4) of the Theorem as above.
QED.
Design and Analysis of Algorithms
(750322)

A Core Module for Students in
Faculty of IT
CS, CIS, SE, and ACS Departments
Philadelphia University

First Semester 2006/2007
Design and Analysis of Algorithms (750322)

Lecturer: Dr. Nadia Y. Yousif

Email: nyaaqob@philadelphia.edu.jo
      nadiayy@hotmail.com

Room: IT 332
Course Outline

- Aims
- Objectives
- Assessment and Passing the Subject
- Lectures and Tutorials
- Lecturer and Consultation
- Recommended Reading
- Course Overview
- Chapter 1
Aims of This Module

- To learn how to develop efficient algorithms for simple computational tasks and reasoning about the correctness of them.
- To be able to measure the complexity of algorithms.
- To understand different range of behaviors of algorithms and the notion of tractable and intractable problems.
- To introduce formal techniques to support the design and analysis of algorithms, focusing on both the underlying mathematical theory and practical considerations of efficiency.
- To introduce topics include asymptotic complexity bounds, techniques of analysis, and algorithmic strategies.
Course Objectives

When completing this module, you should be able to:

- understand basic ideas about algorithms
- develop efficient algorithms for simple computational tasks
- reason about the correctness of algorithms
- understand the concepts of time and space complexity, worst case, average case and best case complexities and the big-O notation
- compute complexity measures of algorithms, including recursive algorithms using recurrence relations
- understand the range of behaviors of algorithms and the notion of tractable and intractable problems
- know and understand a wide range of searching and sorting algorithms
Assessment and Passing

- There are three assessment components:
  - Two midterm exams worth 15% of the marks each
  - Course work worth 15% of the marks
  - Tutorial contribution 5% of the marks
  - Final exam worth 50%

- You need to achieve an overall mark of 50% to pass the course.
Lectures and Tutorials

- Lectures will be held at:
  12:45 – 14:15 pm on Monday and Wednesday in Room 7401
- There will be one tutorial hour per week on average
- Tutorials will start from the fourth week; 1 hour/week

- During the 16 weeks of the course, you will be given four assignments to work on.
- The lecturer will be available to comment on, and help with your development of assignments’ solutions during the office hours.
Lecturer and Consultation

- Lecturer:
  Dr. Nadia Y. Yousif
  Faulty of IT, Room 332, Phone Ext: 544
  email: nyaaqob@philadelphia.edu.jo

- Consultation is at the office hours (in room 332) on:
  (Sun, Tues, Thu) 13:00 – 14:00
  (Mod, Wed) 11:15 – 12:45
Recommended Reading

- **The Textbooks:**

- **Supporting Material:**

**Note:**
Most of the slides of this module are from the 2\textsuperscript{nd} reference
Course Overview

- Introduction, Algorithm definition, Algorithm Analysis
- Mathematical Induction
- Summation Techniques
- Recurrence Relations
- Design & Analysis of Algorithms: Divide and conquer
- Greedy Algorithm
- Dynamic Programming
- Backtracking, Branch-Bound
- Lower Bound Theory
- Sorting and Searching
- NP-Complete Problems: Basic Concepts
- NP-Hard & NP-Complete Problem
Introduction to Algorithm

- Introduction
- Analyzing Algorithms and Problems
Introduction

- To say that a problem is solvable algorithmically means:
  - a computer program can be written that will produce the correct answer for any input if we let it run long enough and allow it as much storage space as it needs.
What is a Computer Algorithm?

- A computer algorithm is
  - a detailed step-by-step method for
  - solving a problem
  - by using a computer.

- Many problems with practical applications can be solved— that is, programs can be written for them.

- The **Time** and **Space** requirements of a program are of practical importance

- they form “Computational Complexity”
What is to study about Algorithms?

- We want to focus on
  - analyzing specific problems and specific algorithms
  - introducing some general design techniques, tools and principles for analyzing algorithms and problems
  - introducing methods of proving correctness
- We will study some techniques such as
  - Divide-and-conquer
  - Greedy algorithms
  - Depth-first search
  - Dynamic Programming
- Given a problem, how do we find an efficient algorithm for its solution?
- How can we compare our algorithm with other algorithms that solve the same problem?
- How should we judge the goodness of an algorithm?
Problem-Solving (Science and Engineering)

- **Analysis**
  - How does it work?
  - Breaking a system down to known components
  - How the components relate to each other
  - Breaking a process down to known functions

- **Synthesis**
  - Building tools and toys!
  - What components are needed
  - How the components should be put together
  - Composing functions to form a process
Problem Solving Using Computers

- **Problem**: What is the problem to be solved
- **Strategy**: In what strategy we are going to solve the problem
- **Algorithm**: Specify the algorithm:
  - Input:
  - Output:
  - Step:
- **Analysis**:
  - Correctness:
  - Time & Space:
  - Optimality:
- **Implementation**:
- **Verification**:
Analyzing Algorithms and Problems

- We analyze the algorithm with the intention of improving it
- and to choose among several available algorithms for the problem.
- We will use the criteria:
  1. Correctness
  2. Amount of work done
  3. Amount of space used
  4. Simplicity, clarity
  5. Optimality
Correctness

- We must have a clear understanding of what “correct” means.
- There are 3 major steps involved in establishing the correctness of an algorithm:
  1- We need a precise statement about the characteristics of the input it is expected to work on (called the **preconditions**)
  2- What result it is to produce for each input (called the **postconditions**)
  3- We can try to prove statements about the relationships between the input and the output.
    - That is, if the preconditions are satisfied, the postconditions will be true when the algorithm terminates.
Correctness

- There are two aspects to an algorithm:
  - the solution method
  - its implementation

- establishing the correctness of the method used may be easy or may require a long sequence of lemmas and theorems.

- If the algorithm is short and straightforward, use some informal means to convince yourself that the parts of the algorithm do what you expect them to do.

- To prove the correctness of a large program, try to break the program into smaller modules that are largely independent; show that, if all modules do their job properly, then the whole program is correct.
Amount of Work Done

- How shall we measure the “amount of work done” by an algorithm (i.e. complexity measure)?
- The measure should aid in comparing two algorithms for the same problem to determine which one is more efficient than the other.
- To analyze an algorithm, we just count the basic operations performed by the algorithm and ignore initialization, loop control, etc.
Average and Worst-Case Analysis

- Complexity measure depends on the size of the input
- Even if the input size is fixed at, say n, the number of operations performed may depend on the particular input.
- How are the results of the analysis of an algorithm to be expressed? We find
  - Worst-cost complexity
  - Average complexity
Space Usage

- The number of memory cells used by a program, depends on the particular implementation.

- A program needs space for
  - instructions
  - constants and variables
  - the input data
  - some workspace for manipulating data & storing information needed to carry out computation
Simplicity

- It is often the case that the simplest and most straightforward way to solve a problem is not the most efficient.
- Simplicity in an algorithm is a desirable feature:
  - it may make verifying the correctness of the algorithm easier
  - it makes writing, debugging, and modifying a program easier
Optimality

- We cannot improve an algorithm for a problem beyond a certain point.
- Each problem has inherent complexity; that is, there is some minimum amount of work required to solve it.
- To analyze the complexity of a problem, we choose class of algorithms and a measure of complexity.
- An algorithm is optimal (in the worst case) if there is no algorithm in the class under study that performs fewer basic operations.
Example: Search in an Unordered Array

Problem:
- Let $E$ be an array containing $n$ entries, $E[0], \ldots, E[n-1]$, in no particular order.
- Find an index of a specified key $K$, if $K$ is in the array;
- return $-1$ as the answer if $K$ is not in the array.

Strategy:
- Compare $K$ to each entry in turn until a match is found or the array is exhausted.
- If $K$ is not in the array, the algorithm returns $-1$ as its answer.
Example: Search in an Unordered Array

Algorithm (and data structure)

- **Input:** E, n, K, where E is an array with n entries (indexed 0, …, n-1), and K is the item sought. For simplicity, we assume that K and the entries of E are integers, as is n.
- **Output:** Returns *ans*, the location of K in E (-1 if K is not found.)
Example: Search in an Unordered Array

Algorithm: Step (Specification)

1. int seqSearch(int[] E, int n, int K)
2. int ans, index;
3. ans = -1;    // Assume failure.
4. for (index = 0; index < n; index++)
5.    if (K == E[index])
6.        {  ans = index;    // Success!  }
7.          break;          // Done!  }
8. return ans;
Example: Search in an Unordered Array

Analysis of the Algorithm

- How shall we measure the amount of work done by an algorithm?

- **Basic Operation:**
  - Comparison of x with an array entry

- **Worst-Case Analysis:**
  - Let \( W(n) \) be a function. \( W(n) \) is the maximum number of basic operations performed by the algorithm on any input size \( n \).
  - For our example, clearly \( W(n) = n \).
  - The worst cases occur when \( K \) appears only in the last position in the array and when \( K \) is not in the array at all.
Example: Search in an Unordered Array

More Analysis of the Algorithm

- **Average-Behavior Analysis:**
  - Let $q$ be the probability that $K$ is in the array
  - $A(n) = n(1 - \frac{1}{2} q) + \frac{1}{2} q$

- **Optimality:**
  - The Best possible solution?
  - Searching an Ordered Array
  - Using Binary Search
  - $W(n) = \text{Ceiling}[\log(n+1)]$
  - The Binary Search algorithm is optimal.

- **Correctness:** (Proving Correctness of Procedures, see later)
Algorithm Language (Specifying the Steps)

- Java as an algorithm language
- Syntax similar to C++
- Some steps within an algorithm may be specified in pseudocode (English phrases)
- Focus on the strategy and techniques of an algorithm, not on detail implementation
Analysis Tool: Mathematics: Set

• A set is a collection of distinct elements.
• The elements are of the same “type”, common properties.
• “element e is a member of set S” is denoted as $e \in S$
• Read “e is in S”
• A particular set is defined by listing or describing its elements between a pair of curly braces:
  $S_1 = \{a, b, c\}$, $S_2 = \{x \mid x \text{ is an integer power of } 2\}$
  read “the set of all elements $x$ such that $x$ is …”
• $S_3 = \{\} = \emptyset$, has not elements, called empty set
• A set has no inherent order.
Subset, Superset; Intersection, Union

• If all elements of one set, $S_1$
  – are also in another set, $S_2$,
• Then $S_1$ is said to be a subset of $S_2$, $S_1 \subseteq S_2$
  – and $S_2$ is said to be a superset of $S_1$, $S_2 \supseteq S_1$.
• Empty set is a subset of every set, $\emptyset \subseteq S$
• Intersection
  
  \[ S \cap T = \{x \mid x \in S \text{ and } x \in T\} \]
• Union
  
  \[ S \cup T = \{x \mid x \in S \text{ or } x \in T\} \]
Cardinality

- **Cardinality**
  - A set, $S$, is *finite* if there is an integer $n$ such that the elements of $S$ can be placed in a one-to-one correspondence with \{1, 2, 3, ..., $n$\}
  - in this case we write $|S| = n$

- How many distinct subsets does a finite set on $n$ elements have? There are $2^n$ subsets.

- How many distinct subsets of cardinality $k$ does a finite set of $n$ elements have?
  There are $C(n, k) = \frac{n!}{((n-k)!k!)}$, “$n$ choose $k$” \[ \binom{n}{k} \]}
Sequence

• A group of elements in a specified order is called a sequence.
• A sequence can have repeated elements.
• Sequences are defined by listing or describing their elements in order, enclosed in parentheses.
  • e.g. $S_1 = (a, b, c)$, $S_2 = (b, c, a)$, $S_3 = (a, a, b, c)$
• A sequence is finite if there is an integer $n$ such that the elements of the sequence can be placed in a one-to-one correspondence with $(1, 2, 3, \ldots, n)$.
• If all the elements of a finite sequence are distinct, that sequence is said to be a permutation of the finite set consisting of the same elements.
• One set of $n$ elements has $n!$ distinct permutations.
Tuples and Cross Product

• A tuple is a finite sequence.
  – Ordered pair \((x, y)\), triple \((x, y, z)\), quadruple, and quintuple
  – A \(k\)-tuple is a tuple of \(k\) elements.

• The \textit{cross product} of two sets, say \(S\) and \(T\), is
  \[ S \times T = \{(x, y) \mid x \in S, y \in T\} \]

• \(|S \times T| = |S| \cdot |T|\)

• It often happens that \(S\) and \(T\) are the same set, e.g.
  \[ \mathbb{N} \times \mathbb{N} \]
  where \(\mathbb{N}\) denotes the set of natural numbers,
  \[ \{0, 1, 2, \ldots\} \]
Relations and Functions

• A relation is some subset of a (possibly iterated) cross product.
• A binary relation is some subset of a cross product, e.g. $R \subseteq S \times T$
• e.g. “less than” relation can be defined as
  $\{(x, y) | x \in \mathbb{N}, y \in \mathbb{N}, x < y\}$
• Important properties of relations; let $R \subseteq S \times S$
  – reflexive: for all $x \in S$, $(x, x) \in R$.
  – symmetric: if $(x, y) \in R$, then $(y, x) \in R$.
  – antisymmetric: if $(x, y) \in R$, then $(y, x) \not\in R$
  – transitive: if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.
• A relation that is reflexive, symmetric, and transitive is called an equivalence relation, partition the underlying set $S$ into equivalence classes $[x] = \{y \in S | x R y\}$, $x \in S$
• A function is a relation in which no element of $S$ (of $S \times T$) is repeated with the relation (informal def.)
Analysis Tool: Logic

• Logic is a system for formalizing natural language statements so that we can reason more accurately.
• The simplest statements are called *atomic formulas*.
• More complex statements can be build up through the use of *logical connectives*: \( \land \) “and”, \( \lor \) “or”, \( \neg \) “not”, \( \Rightarrow \) “implies” A \( \Rightarrow \) B “A implies B” “if A then B”
• A \( \Rightarrow \) B is logically equivalent to \( \neg A \lor B \)
• \( \neg (A \land B) \) is logically equivalent to \( \neg A \lor \neg B \)
• \( \neg (A \lor B) \) is logically equivalent to \( \neg A \land \neg B \)
Quantifiers: all, some

- “for all x” $\forall x \ P(x)$ is true iff $P(x)$ is true for all $x$
  - universal quantifier (universe of discourse)
- “there exist x” $\exists x \ P(x)$ is true iff $P(x)$ is true for some value of $x$
  - existential quantifier
- $\forall x \ A(x)$ is logically equivalent to $\neg \exists x(\neg A(x))$
- $\exists x \ A(x)$ is logically equivalent to $\neg \forall x(\neg A(x))$
- $\forall x \ (A(x) \Rightarrow B(x))$
  “For all x such that if $A(x)$ holds then $B(x)$ holds”
Prove: by counterexample, Contraposition, Contradiction

- **Counterexample**
  to prove $\forall x \ (A(x) \Rightarrow B(x))$ is false, we show some object $x$ for which $A(x)$ is true and $B(x)$ is false.
  
  $\neg (\forall x \ (A(x) \Rightarrow B(x))) \iff \exists x \ (A(x) \land \neg B(x))$

- **Contraposition**
  to prove $A \Rightarrow B$, we show $(\neg B) \Rightarrow (\neg A)$

- **Contradiction**
  to prove $A \Rightarrow B$, we assume $\neg B$ and then prove $B$.
  
  $A \Rightarrow B \iff (A \land \neg B) \Rightarrow B$
  
  $A \Rightarrow B \iff (A \land \neg B)$ is false
  
  Assuming $(A \land \neg B)$ is true, and discover a *contradiction* (such as $A \land \neg A$), then conclude $(A \land \neg B)$ is false, and so $A \Rightarrow B$. 
Prove: by Contradiction, e.g.

- Prove \([B \land (B \Rightarrow C)] \Rightarrow C\)
  - by contradiction
- Proof:
  - Assume \(\neg C\)
  - \(\neg C \land [B \land (B \Rightarrow C)]\)
  - \(\Rightarrow \neg C \land [B \land (\neg B \lor C)]\)
  - \(\Rightarrow \neg C \land [(B \land \neg B) \lor (B \land C)]\)
  - \(\Rightarrow \neg C \land [(B \land C)]\)
  - \(\Rightarrow \neg C \land C \land B\)
  - \(\Rightarrow \) False, \textit{Contradiction}
  - \(\Rightarrow C\)
Rules of Inference

- A rule of inference is a *general pattern* that allows us to draw some new conclusion from a set of given statements.
  - If we know {...} then we can conclude {...}
- If \{B and (B \Rightarrow C)\} then \{C\}
  - modus ponens
- If \{A \Rightarrow B and B \Rightarrow C\} then \{A \Rightarrow C\}
  - syllogism
- If \{B \Rightarrow C and \neg B \Rightarrow C\} then \{C\}
  - rule of cases
Two-valued Boolean (algebra) logic

1. There exists two elements in B, i.e. $B=\{0,1\}$
   - there are two binary operations $+$ “or, $\lor$”, $\cdot$ “and, $\land$”
2. Closure: if $x, y \in B$ and $z = x + y$ then $z \in B$
   - if $x, y \in B$ and $z = x \cdot y$ then $z \in B$
3. Identity element: for $+$ designated by 0: $x + 0 = x$
   - for $\cdot$ designated by 1: $x \cdot 1 = x$
4. Commutative: $x + y = y + x$
   - $x \cdot y = y \cdot x$
5. Distributive: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
   - $x + (y \cdot z) = (x + y) \cdot (x + z)$
6. Complement: for every element $x \in B$, there exits an element $x' \in B$
   - $x + x' = 1$, $x \cdot x' = 0$
True Table and Tautologically Implies e.g.

• Show \([B \land (B \implies C)] \implies C\) is a tautology:

\[
\begin{array}{ccc|ccc|cc}
B & C & (B \implies C) & [B \land (B \implies C)] & [B \land (B \implies C)] & \implies & C \\
0 & 0 & 1 & 0 & & & 1 \\
0 & 1 & 1 & 0 & & & 1 \\
1 & 0 & 0 & 0 & & & 1 \\
1 & 1 & 1 & 1 & & & 1 \\
\end{array}
\]

• For every assignment for B and C,
  – the statement is True
Prove: by Rule of inferences

• Prove \([B \land (B \Rightarrow C)] \Rightarrow C\)
  – Proof:
  – \([B \land (B \Rightarrow C)] \Rightarrow C\)
  – \(\Rightarrow \neg [B \land (B \Rightarrow C)] \lor C\)
  – \(\Rightarrow \neg [B \land (\neg B \lor C)] \lor C\)
  – \(\Rightarrow \neg [(B \land \neg B) \lor (B \land C)] \lor C\)
  – \(\Rightarrow \neg [(B \land C)] \lor C\)
  – \(\Rightarrow \neg C \lor C\)
  – \(\Rightarrow \text{True (tautology)}\)
• Direct Proof:
  – \([B \land (B \Rightarrow C)] \Rightarrow [B \land C] \Rightarrow C\)
Analysis Tool: Probability

• Elementary events (outcomes)
  – Suppose that in a given situation an event, or experiment, may have any one, and only one, of k outcomes, \( s_1, s_2, \ldots, s_k \). (mutually exclusive)

• Universe
  The set of all elementary events is called the universe and is denoted \( U = \{s_1, s_2, \ldots, s_k\} \).

• Probability of \( s_i \)
  • associate a real number \( \Pr(s_i) \), such that
  • \( 0 \leq \Pr(s_i) \leq 1 \) for \( 1 \leq i \leq k \);
  • \( \Pr(s_1) + \Pr(s_2) + \ldots + \Pr(s_k) = 1 \)
Event

- Let $S \subseteq U$. Then $S$ is called an *event*, and
- $Pr(S) = \sum_{s_i \in S} Pr(s_i)$
- Sure event $U = \{s_1, s_2, \ldots, s_k\}$, $Pr(U) = 1$
- Impossible event, $\emptyset$, $Pr(\emptyset) = 0$
- Complement event “not $S$” $U - S$, $Pr(\text{not } S) = 1 - Pr(S)$
Conditional Probability

- The conditional probability of an event $S$ given an event $T$ is defined as
- $\Pr(S \mid T) = \frac{\Pr(S \text{ and } T)}{\Pr(T)}$
  $= \frac{\sum_{s_i \in S \cap T} \Pr(s_i)}{\sum_{s_j \in T} \Pr(s_j)}$

**Independent**

- Given two events $S$ and $T$, if
  - $\Pr(S \text{ and } T) = \Pr(S)\Pr(T)$
  - then $S$ and $T$ are stochastically independent, or simply independent.
Random variable and their Expected value

- A random variable is a real valued variable that depends on which elementary event has occurred
  - it is a function defined for elementary events.
  - e.g. \( f(e) = \) the number of inversions in the permutation of \{A, B, C\}; assume all input permutations are equally likely.

- Expectation
  - Let \( f(e) \) be a random variable defined on a set of elementary events \( e \in U \). The expectation of \( f \), denoted as \( E(f) \), is defined as
    \[
    E(f) = \sum_{e \in U} f(e)Pr(e)
    \]
    - This is often called the average values of \( f \).
    - Expectations are often easier to manipulate than the random variables themselves.
Conditional expectation and Laws of expectations

• The *conditional expectation* of \( f \) given an event \( S \), denoted as \( E(f \mid S) \), is defined as
  \[
  E(f \mid S) = \sum_{e \in S} f(e)Pr(e \mid S)
  \]

• *Law of expectations*

• For random variables \( f(e) \) and \( g(e) \) defined on a set of elementary events \( e \in U \), and any event \( S \):
  • \( E(f + g) = E(f) + E(g) \)
  • \( E(f) = Pr(S)E(f \mid S) + Pr(\text{not } S) E(f \mid \text{not } S) \)
Analysis Tool: Algebra

- **Manipulating Inequalities**
- Transitivity: If \((A \leq B)\) and \((B \leq C)\) Then \((A \leq C)\)
- Addition: If \((A \leq B)\) and \((C \leq D)\) Then \((A+C \leq B+D)\)
- Positive Scaling: 
  If \((A \leq B)\) and \((\alpha > 0)\) Then \((\alpha A \leq \alpha B)\)

- **Floor and Ceiling Functions**
- Floor\([x]\) is the largest integer less than or equal to \(x\).
  \[
  \left\lfloor x \right\rfloor
  \]
- Ceiling\([x]\) is the smallest integer greater than or equal to \(x\).
  \[
  \left\lceil x \right\rceil
  \]
Logarithms

– For b>1 and x>0,
  \( \log_b x \) (read “log to the base b of x”) is that real number L such that \( b^L = x \)
– \( \log_b x \) is the power to which b must be raised to get x.

• Log properties: def: \( \lg x = \log_2 x; \quad \ln x = \log_e x \)
  • Let x and y be arbitrary positive real numbers, let a, b any real number, and let b>1 and c>1 be real numbers.
    – \( \log_b \) is a strictly increasing function,
      if \( x > y \) then \( \log_b x > \log_b y \)
    – \( \log_b \) is a one-to-one function,
      if \( \log_b x = \log_b y \) then \( x = y \)
    – \( \log_b 1 = 0; \log_b b = 1; \log_b x^a = a \log_b x \)
    – \( \log_b(xy) = \log_b x + \log_b y \)
    – \( x^{\log y} = y^{\log x} \)
    – change base: \( \log_c x = \frac{(\log_b x)}{(\log_b c)} \)
Series

- A series is the sum of a sequence.
- Arithmetic series
  - The sum of consecutive integers
    \[ \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \]
- Polynomial Series
  - The sum of squares
    \[ \sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \approx \frac{n^3}{3} \]
  - The general case is
    \[ \sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} \]
- Power of 2
  \[ \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \]
- Arithmetic-
- Geometric Series
  \[ \sum_{i=1}^{n} i2^i = (k-1)2^{k+1} + 2 \]
Summations Using Integration

– A function \( f(x) \) is said to be monotonic, or nondecreasing, if \( x \leq y \) always implies that \( f(x) \leq f(y) \).
– A function \( f(x) \) is antimonotonic, or nonincreasing, if \(-f(x)\) is monotonic.

• If \( f(x) \) is nondecreasing then

\[
\int_{a-1}^{b} f(x) \, dx \leq \sum_{i=a}^{b+1} f(i) \leq \int_{a}^{b+1} f(x) \, dx
\]

• If \( f(x) \) is nonincreasing then

\[
\int_{a}^{b+1} f(x) \, dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x) \, dx
\]
Classifying functions by their Asymptotic Growth Rates

- asymptotic growth rate, asymptotic order, or order of functions
  - Comparing and classifying functions that ignores constant factors and small inputs.

- The Sets big oh \( \mathcal{O}(g) \), big theta \( \Theta(g) \), big omega \( \Omega(g) \)

\( \Omega(g) \): functions that grow at least as fast as \( g \)

\( \Theta(g) \): functions that grow at the same rate as \( g \)

\( \mathcal{O}(g) \): functions that grow no faster than \( g \)
The Sets $O(g)$, $Θ(g)$, $Ω(g)$

- Let $g$ and $f$ be a functions from the nonnegative integers into the positive real numbers
- For some real constant $c > 0$ and some nonnegative integer constant $n_0$
  - $O(g)$ is the set of functions $f$, such that $f(n) \leq c \, g(n)$ for all $n \geq n_0$
  - $Ω(g)$ is the set of functions $f$, such that $f(n) \geq c \, g(n)$ for all $n \geq n_0$
  - $Θ(g) = O(g) \cap Ω(g)$
    - asymptotic order of $g$
    - $f \in Θ(g)$ read as “$f$ is asymptotic order $g$” or “$f$ is order $g$”
Comparing asymptotic growth rates

- Comparing \( f(n) \) and \( g(n) \) as \( n \) approaches infinity,
- IF \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \)
- \( < \infty \), including the case in which the limit is 0 then \( f \in O(g) \)
- \( > 0 \), including the case in which the limit is \( \infty \) then \( f \in \Omega(g) \)
- \( = c \) and \( 0 < c < \infty \) then \( f \in \Theta(g) \)
- \( = 0 \) then \( f \in o(g) \) //read as “little oh of \( g \)”
- \( = \infty \) then \( f \in \omega(g) \) //read as “little omega of \( g \)”
Properties of $O(g)$, $\Theta(g)$, $\Omega(g)$

- Transitive: If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
  $O$ is transitive. Also $\Omega$, $\Theta$, $o$, $\omega$ are transitive.
- Reflexive: $f \in \Theta(f)$
- Symmetric: If $f \in \Theta(g)$, then $g \in \Theta(f)$
- $\Theta$ defines an equivalence relation on the functions.
  - Each set $\Theta(f)$ is an equivalence class (complexity class).
- $f \in O(g) \iff g \in \Omega(f)$
- $O(f + g) = O(\max(f, g))$
  similar equations hold for $\Omega$ and $\Theta$
Classification of functions, e.g.

- \( \mathcal{O}(1) \) denotes the set of functions bounded by a *constant* (for large \( n \))
- \( f \in \Theta(n) \), \( f \) is *linear*
- \( f \in \Theta(n^2) \), \( f \) is *quadratic*; \( f \in \Theta(n^3) \), \( f \) is *cubic*
- \( \lg n \in o(n^{\alpha}) \) for any \( \alpha > 0 \), including factional powers
- \( n^k \in o(c^n) \) for any \( k > 0 \) and any \( c > 1 \)
  - powers of \( n \) grow more slowly than any exponential function \( c^n \)

\[
\sum_{i=1}^{n} i^d \in \Theta(n^{d+1}) \quad \sum_{i=1}^{n} \log(i) \in \Theta(n \log(n))
\]

\[
\sum_{i=a}^{b} r^i \in \Theta(r^b) \quad \text{for} \quad r > 0, \quad r \neq 1, \quad \text{by some function of} \quad n
\]
Analyzing Algorithms and Problems

• We analyze algorithms with the intention of improving them, if possible, and for choosing among several available for a problem.

• Correctness
• Amount of work done, and space used
• Optimality, Simplicity
Correctness can be proved!

- An algorithm consists of sequences of steps (operations, instructions, statements) for transforming inputs (preconditions) to outputs (postconditions).
- Proving
  - if the preconditions are satisfied,
  - then the postconditions will be true,
  - when the algorithm terminates.
Amount of work done

- We want a measure of work that tells us something about the efficiency of the method used by the algorithm
- independent of computer, programming language, programmer, and other implementation details.
- Usually depending on the size of the input
- Counting passes through loops
- Basic Operation
  - Identify a particular operation fundamental to the problem
  - the total number of operations performed is roughly proportional to the number of basic operations
- Identifying the properties of the inputs that affect the behavior of the algorithm
Worst-case complexity

– Let $D_n$ be the set of inputs of size $n$ for the problem under consideration, and let $I$ be an element of $D_n$.
– Let $t(I)$ be the number of basic operations performed by the algorithm on input $I$.
– We define the function $W$ by

\[ W(n) = \max \{ t(I) \mid I \in D_n \} \]

– called the worst-case complexity of the algorithm.
– $W(n)$ is the maximum number of basic operations performed by the algorithm on any input of size $n$.

• The input, $I$, for which an algorithm behaves worst depends on the particular algorithm.
Average Complexity

– Let Pr(I) be the *probability* that input I occurs.
– Then the average behavior of the algorithm is defined as

\[ A(n) = \sum_{I \in D_n} Pr(I) \cdot t(I). \]
– We determine \( t(I) \) by analyzing the algorithm,
– but \( Pr(I) \) cannot be computed analytically.

\[ A(n) = \Pr(\text{succ})A_{\text{succ}}(n) + \Pr(\text{fail})A_{\text{fail}}(n) \]

• An element I in \( D_n \) may be thought as a set or equivalence class that affect the behavior of the algorithm. (see following e.g. \( n+1 \) cases)
e.g. Search in an unordered array

- int seqSearch(int[] E, int n, int K)
- 1. int ans, index;
- 2. ans = -1; // Assume failure.
- 3. for (index = 0; index < n; index++)
- 4. if (K == E[index])
- 5. ans = index; // Success!
- 6. break; // Done!
- 7. return ans;
Average-Behavior Analysis e.g.

- \( A(n) = \Pr(\text{succ})A_{\text{succ}}(n) + \Pr(\text{fail})A_{\text{fail}}(n) \)
- There are total of \( n+1 \) cases of \( I \) in \( D_n \)
  - Let \( K \) is in the array as “succ” cases that have \( n \) cases.
  - Assuming \( K \) is equally likely found in any of the \( n \) location, i.e. \( \Pr(I_i | \text{succ}) = \frac{1}{n} \)
  - for \( 0 \leq i < n \), \( t(I_i) = i + 1 \)
  - \( A_{\text{succ}}(n) = \sum_{i=0}^{n-1} \Pr(I_i | \text{succ}) \cdot t(I_i) \)
    - \( = \sum_{i=0}^{n-1} \frac{1}{n} (i+1) = \frac{1}{n} [n(n+1)/2] = \frac{(n+1)}{2} \)
  - Let \( K \) is not in the array as the “fail” case that has \( 1 \) cases, \( \Pr(I | \text{fail}) = 1 \)
    - Then \( A_{\text{fail}}(n) = \Pr(I | \text{fail}) \cdot t(I) = 1/n \)
- Let \( q \) be the probability for the succ cases
  - \( q \left( \frac{n+1}{2} \right) + (1-q) \cdot n \)
Space Usage

• If memory cells used by the algorithms depends on the particular input,
  – then worst-case and average-case analysis can be done.

• Time and Space Tradeoff.
Optimality “the best possible”

- Each problem has inherent complexity
  - There is some *minimum* amount of work required to solve it.
- To analyze the complexity of a problem,
  - we choose a class of algorithms, based on which
  - prove theorems that establish a *lower bound* on the number of operations needed to solve the problem.
- Lower bound (for the worst case)
Show whether an algorithm is optimal?

• Analyze the algorithm, call it A, and found the Worst-case complexity $W_A(n)$, for input of size n.

• Prove a theorem starting that,
  – for any algorithm in the same class of A
  – for any input of size n, there is some input for which the algorithm must perform
  – at least $W_{[A]}(n)$
    (lower bound in the worst-case)

• If $W_A(n) \equiv W_{[A]}(n)$
  – then the algorithm A is optimal
  – else there may be a better algorithm
  – OR there may be a better lower bound.
Optimality e.g.

• Problem
  – Finding the largest entry in an (unsorted) array of n numbers
• Algorithm A
  int findMax(int[] E, int n)
  1. int max;
  2. max = E[0]; // Assume the first is max.
  3. for (index = 1; index < n; index++)
  4.    if (max < E[index])
  5.      max = E[index];
  6. return max;
Analyze the algorithm, find $W_A(n)$

- **Basic Operation**
  - Comparison of an array entry with another array entry or a stored variable.

- **Worst-Case Analysis**
  - For any input of size $n$, there are exactly $n-1$ basic operations
  - $W_A(n) = n-1$
For the class of algorithm \([A]\), find \(W_{[A]}(n)\)

- **Class of Algorithms**
  - Algorithms that can compare and copy the numbers, but do no other operations on them.
- **Finding (or proving) \(W_{[A]}(n)\)**
  - Assuming the entries in the array are all distinct
    - (permissible for finding lower bound on the worst-case)
    - In an array with \(n\) distinct entries, \(n - 1\) entries are not the maximum.
  - To conclude that an entry is not the maximum, it must be smaller than at least one other entry. And, one comparison (basic operation) is needed for that.
  - So at least \(n-1\) basic operations must be done.
  - \(W_{[A]}(n) = n - 1\)

Since \(W_A(n) = W_{[A]}(n)\), algorithm \(A\) is optimal.
Simplicity

• Simplicity in an algorithm is a virtue.
Designing Algorithms

• Problem solving using Computer
• Algorithm Design Techniques
  – divide-and-conquer
  – greedy methods
  – depth-first search (for graphs)
  – dynamic programming
Problem and Strategy A

- **Problem: array search**
  - Given an array $E$ containing $n$ and given a value $K$, find an index for which $K = E[index]$ or, if $K$ is not in the array, return $-1$ as the answer.

- **Strategy A**
  - Input data and Data structure: unsorted array
  - sequential search

- **Algorithm A**
  - int seqSearch(int[] E, int n, int k)

- **Analysis A**
  - $W(n) = n$
  - $A(n) = q \left( \frac{n+1}{2} \right) + (1-q)n$
Better Algorithm and/or Better Input Data

• **Optimality A**
  – for searching an unsorted array
  – \( W[A](n) = n \)
  – Algorithm A is optimal.

• **Strategy B**
  – Input data and Data structure: array sorted in nondecreasing order
  – sequential search

• **Algorithm B.**
  – int seqSearch(int[] E, int n, int k)

• **Analysis B**
  – \( W(n) = n \)
  – \( A(n) = q \left\lfloor \frac{(n+1)}{2} \right\rfloor + (1-q) n \)
Better Algorithm

• **Optimality B**
  – It makes no use of the fact that the entries are ordered
  – Can we modify the algorithm so that it uses the added information and does less work?

• **Strategy C**
  – Input data and Data structure: array sorted in nondecreasing order
  – sequential search:
    as soon as an entry larger than K is encountered, the algorithm can terminate with the answer −1.
Algorithm C: modified sequential search

```c
int seqSearchMod(int[] E, int n, int K)
1. int ans, index;
2. ans = -1; // Assume failure.
3. for (index = 0; index < n; index++)
4.   if (K > E[index])
5.     continue;
6.   if (K < E[index])
7.     break; // Done!
8.   // K == E[index]
9.   ans = index; // Find it
10.  break;
11. return ans;
```
Analysis C

- \( W(n) = n + 1 \approx n \)
- **Average-Behavior**
  - \( n \) cases for success:
    - \( A_{\text{succ}}(n) = \sum_{i=0}^{n-1} \Pr(I_i | \text{succ}) \cdot t(I_i) \)
    - \( = \sum_{i=0}^{n-1} \left( \frac{1}{n} \right) (i+2) = \frac{3 + n}{2} \)
  - \( n+1 \) cases or (gaps) for fail: \(<E[0]<E[1]…E[n-1]>\)
- \( A_{\text{fail}}(n) = \Pr(I_i | \text{fail}) \cdot t(I_i) = \)
- \( \sum_{i=0}^{n-1} \left( \frac{1}{(n+1)} \right) (i+2) + \frac{n}{(n+1)} \)
- \( A(n) = q \left( \frac{3+n}{2} \right) + (1-q) \left( \frac{n}{(n+1)} + \frac{3+n}{2} \right) \)
- \( \approx \frac{n}{2} \)
Let’s Try Again! Let’s divide-and-conquer!

- **Strategy D**
  - compare K first to the entry in the middle of the array
  - -- eliminates half of the entry with one comparison
  - apply the same strategy recursively

- **Algorithm D: Binary Search**
  - Input: E, first, last, and K, all integers, where E is an ordered array in the range first, ..., last, and K is the key sought.
  - Output: index such that E[index] = K if K is in E within the range first, ..., last, and index = -1 if K is not in this range of E
Binary Search

int binarySearch(int[] E, int first, int last, int K)
1. if (last < first)
2. index = -1;
3. else
4. int mid = (first + last)/2
5. if (K == E[mid])
6. index = mid;
7. else if (K < E[mid])
8. index = binarySearch(E, first, mid-1, K)
9. else
10. index = binarySearch(E, mid+1, last, K);
11. return index
Worst-Case Analysis of Binary Search

– Let the problem size be \( n = \text{last} - \text{first} + 1; \ n>0 \)

• Basic operation is a comparison of \( K \) to an array entry
  – Assume one comparison is done with the three-way branch
  – First comparison, assume \( K \neq E[\text{mid}] \), divides the array into two sections, each section has at most \( \text{Floor}[n/2] \) entries.
  – estimate that the size of the range is divided by 2 with each recursive call
  – How many times can we divide \( n \) by 2 without getting a result lest than 1 (i.e. \( n/(2^d) \geq 1 \))?
  – \( d \leq \lg(n) \), therefore we do \( \text{Floor}[\lg(n)] \) comparison following recursive calls, and one before that.
  – \( W(n) = \text{Floor}[\lg(n)] + 1 = \text{Ceiling}[\lg(n + 1)] \in \Theta(\log n) \)
Average-Behavior Analysis of Binary Search

- There are n+1 cases, n for success and 1 for fail
- Similar to worst-case analysis, Let n = 2^d – 1
  \( A_{\text{fail}} = \lg(n+1) \)
- Assuming \( \Pr(I_i | \text{succ}) = 1/n \) for \( 1 \leq i \leq n \)
  - divide the n entry into groups, \( S_t \) for \( 1 \leq t \leq d \), such that \( S_t \) requires \( t \) comparisons (capital S for group, small s for cardinality of S)
  - It is easy to see (?) that (members contained in the group)
    - \( s_1 = 1 = 2^0 \), \( s_2 = 2 = 2^1 \), \( s_3 = 4 = 2^2 \), and in general, \( s_t = 2^{t-1} \)
    - The probability that the algorithm does \( t \) comparisons is \( s_t/n \)
    - \( A_{\text{suc}}(n) = \sum_{t=1}^{d} (s_t/n) t = ((d-1)2^d + 1)/n \)
    - \( d = \lg(n+1) \)
    - \( A_{\text{suc}}(n) = \lg(n+1) – 1 + \lg(n+1)/n \)
- \( A(n) \approx \lg(n+1) – q \), where \( q \) is probability of successful search
Optimality of Binary Search

- So far we improve from $\Theta(n)$ algorithm to $\Theta(\log n)$
  - Can more improvements be possible?
- Class of algorithm: comparison as the basic operation
- Analysis by using decision tree, that
  - for a given input size $n$ is a binary tree whose nodes are labeled with numbers between 0 and $n-1$ as e.g.

```
        4
       / \
      1   7
     /     \
    0  2    5  8
   /   \
  3     6
```

 distances: 0, 1, 2, 3
Decision tree for analysis

• The number of comparisons performed in the worst case is the number of nodes on a longest path from the root to a leaf; call this number p.
• Suppose the decision tree has N nodes
• \( N \leq 1 + 2 + 4 + \ldots + 2^{p-1} \)
• \( N \leq 2^p - 1 \)
• \( 2^p \geq (N + 1) \)
• Claim \( N \geq n \) if an algorithm A works correctly in all cases
  – there is some node in the decision tree labeled \( i \) for each \( i \) from 0 through \( n - 1 \)
Prove by contradiction that N >= n

• Suppose there is no node labeled i for some i in the range from 0 through n-1
  – Make up two input arrays E1 and E2 such that
  – For all j < i, make E1[j] = E2[j] using some key values less than K
  – For all j > i, make E1[j] = E2[j] using some key values greater than K’ in sorted order
  – Since no node in the decision tree is labeled i, the algorithm A never compares K to E1[i] or E2[i], but it gives same output for both
  – Such algorithm A gives wrong output for at least one of the array and it is not a correct algorithm

• Conclude that the decision has at least n nodes
Optimality result

- $2^p \geq (N+1) \geq (n+1)$
- $p \geq \lg(n+1)$

- Theorem: Any algorithm to find $K$ in an array of $n$ entries (by comparing $K$ to array entries) must do at least $\lceil \lg(n+1) \rceil$ comparisons for some input.

- Corollary: Since Algorithm D does $\lceil \lg(n+1) \rceil$ comparisons in the worst case, it is optimal.
Mathematical Induction

Topics to cover:

What is a Proof?
Induction Proofs
Proving Correctness of Procedures: *Proof*

- **What is a Proof?**
  - A Proof is a *sequence* of statements that form a logical argument.
  - Each statement is a complete sentence in the normal grammatical sense.
- **Each statement should draw a new conclusion from:**
  - *axiom*: well known facts
  - *assumptions*: premises of the theorem you are proving or inductive hypothesis
  - *intermediate conclusions*: statements established earlier
- **To** arrive at the last statement of a proof that must be the conclusion of the proposition being proven
Format of Theorem, Proof Format

• A proposition (theorem, lemma, and corollary) is represented as:

  \[ \forall x \in W ( A(x) \implies C(x) ) \]

  for all x in W, if A(x) then C(x)

  – the set W is called world,
  – A(x) represents the assumptions
  – C(x) represents the conclusion, the goal statement
  – \( \implies \) is read as “implies”

• Proof sketches provides outline of a proof
  – the strategy, the road map, or the plan.

• Two-Column Proof Format
  – Statement : Justification (supporting facts)
Induction Proofs

• **Induction proofs** are a mechanism, often the only mechanism, for proving a statement about an infinite set of objects.
  – Inferring a property of a set based on the property of its objects
• Induction is often done *over* the set of natural numbers \( \{0,1,2,\ldots\} \)
  – starting from 0, then 1, then 2, and so on
• Induction is valid over a set, provided that:
  – The set is partially ordered;
    • i.e. an order relationship is defined between some pairs of elements, but perhaps not between all pairs.
  – There is no infinite chain of decreasing elements in the set.
    (e.g. cannot be set of all integers)
Induction Proof Schema

• 0: Prove: $\forall x \in W \ ( A(x) \Rightarrow C(x) )$

• Proof:
  – 1: The Proof is by induction on x, <description of x>
  – 2: The base case is, cases are, <base-case>
  – 3: <Proof of goal statement with base-case substituted into it, that is, C(base-case)>
  – 4: For $x$ greater than <base-case>, assume that $A(y) \Rightarrow C(y)$ holds for all $y \in W$ such that $y < x$.
  – 5: <Proof of the goal statement, C(x), exactly as it appears in the proposition>. 
Induction Proof e.g.

• Prove:
  For all $n \geq 0$,
  $\sum_{i=0}^{n} \frac{i(i+1)}{2} = \frac{n(n+1)(n+2)}{6}$
• Proof: …
Recursion and Recurrence Relations

Topics to cover:

• Recursive Procedures
• Recurrence Equations
• Proving Correctness of Procedures
• Recurrence Equations
• Recursion Trees
Recurrence Equations vs. Recursive Procedures

NOTE: For advanced algorithm development, recursion is an essential design technique.

• Recurrence Equations:
  – defines a function over the natural numbers, say \( T(n) \), in terms of its own value at one or more integers smaller than \( n \).
  – \( T(n) \) is defined inductively.
  – There are base cases to be defined separately.
  – Recurrence equation applies for \( n \) larger than the base cases

• Recursive Procedures:
  – a procedure calls a unique copy of itself
  – converging to a base case (stopping the recursion)
e.g. Fibonacci Function

- **Recurrence Equation**: e.g. Fibonacci Function
  
  - $\text{fib}(0) = 0; \text{fib}(1) = 1;  \quad \text{// base cases}$
  
  - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)  \quad \text{// all } n>2$

- **Recursive Procedure**:

```c
int fib(int n) {
    int f, f1, f2;
    1. if (n < 2) 2.       f = n; \quad \text{// base cases}
    3. else 4.      f1 = fib(n - 1); 5.      f2 = fib(n - 2); 6.      f = f1 + f2;
    7. return f;
} 
```
The Working of Recursive Procedure

- a unique copy for each call to itself
  - individual procedure invocation at run time
  - i.e. activation frame

- e.g. The working of fib(n)
  ```
  main()
  int x = fib(3);
  ```
The Working of Recursive Procedure

main
x :

fib
n : 3
f1 :
f2 :
f :

line: 1

main
x :

fib
n : 3
f1 :
f2 :
f :

line: 4

main
x :

fib
n : 1
f1 :
f2 :

line: 4

main
x :

fib
n : 2
f1 :
f2 :

line: 4

main
x :

fib
n : 2
f1 :
f2 :

line: 4

main
x :

fib
n : 1
f1 :
f2 :

line: 4

main
x :

fib
n : 1
f1 :
f2 :

line: 5

main
x :

fib
n : 1
f1 :

line: 5

main
x :

fib
n : 0
f1 :

line: 7

Design and Analysis of Algorithms
by Dr. Nadia Y. Yousif
Activation Tree

- Each node corresponds to a different procedure invocation, just at the point when it is about to return.
- A preorder traversal visits each activation frame in order of its creation.
Analysis for Algorithm Without Loops

- In a computation without loops, but possible with recursive procedure calls:
  - The time that any particular activation frame is on the top of the frame stack is $O(L)$,
  - where $L$ is the number of lines in the procedure that contains either a simple statement or a procedure call.
  - The total computation time is $\Theta(C)$,
  - where $C$ is the total number of procedure calls that occur during the computation.
Designing Recursive Procedures

• // Think Inductively
• converging to a base case (stopping the recursion)
  – identify some unit of measure (running variable)
  – identify base cases
• assume p solves all sizes 0 through 100
  – assume p99 solve sub-problem all sizes 0 through 99
  – if p detect a case that is not base case it calls p99
• p99 satisfies:
  – 1. The sub-problem size is less than p’s problem size
  – 2. The sub-problem size is not below the base case
  – 3. The sub-problem satisfies all other preconditions of p99 (which are the same as the preconditions of p)
Recursive Procedure Design: An Example

- **Problem:**
  - write a delete(L, x) procedure for a list L
  - which is supposed to delete the first occurrence of x.
  - Possibly x does not occur in L.

- **Strategy:**
  - Use recursive Procedure
  - The size of the problem is the number of elements in list L
  - Use IntList ADT
  - Base cases: ??
  - Running variable (converging number): ??
ADT for IntList

- IntList cons(int newElement, IntList oldList)
  - Precondition: None.
  - Postconditions: If x = cons(newElement, oldList) then
    1. x refers to a newly created object;
    2. x != nil;
    3. first(x) = newElement;
    4. rest(x) = oldList

- int first(IntList aList) // access function
  - Precondition: aList != nil

- IntList rest(IntList aList) // access function
  - Precondition: aList != nil

- IntList nil //constant denoting the empty list.
Recurrence Equation for 
delete (L, x) from list L

Think Inductively

- \( \text{delete (nil, x)} = \text{nil} \)
- \( \text{delete (L, x)} = \text{rest(L)} \quad // x == \text{first(L)} \)
- \( \text{delete (L, x)} = \text{cons(first(L), delete(rest(L), x))} \)
Algorithm for Recursive delete \((L, x)\) from list

IntList delete (IntList L, int x)

IntList newL, fixedL;

if (L == nil)
    newL = L;

else if (x == first(L))
    newL = rest(L);

else
    fixedL = delete (rest (L), x);
    newL = cons(first(L), fixedL);

return newL;
Algorithm for Non-Recursive delete(L, x)

IntList delete (IntList L, int x)
    IntList newL, tempL;
    tempL = L; newL = nil;
    while (tempL != nil && x != first (tempL))       //copy elements
        newL = cons (first (tempL), newL);
        tempL = rest (tempL)
    If (tempL != nil) // x == first (tempL)
        tempL = rest (tempL)              // remove x
    while (tempL != nil)                  // copy the rest elements
        newL = cons (first (tempL), newL);
        tempL = rest (tempL)
    return newL;
Convert a Non-Recursive Procedure to a Recursive Procedure

- Convert a procedure with loop
  - to a recursive procedure without loop

- Recursive Procedure acting like WHILE loop
  While(Not Base Case)
  Setting up Sub-problem
  Recursive call to continue

- The recursive function may need an additional parameter
  - which replaces an index in a FOR loop of the non-recursive procedure.
Transforming Loop into a Recursive Procedure

- Local variable with the loop body
  - give the variable only one value in any one pass
  - for variable that must be updated, do all the updates at the end of the loop body
- Re-expressing a while loop with recursion
  - Additional parameters
    - Variables updated in the loop become procedure input parameters. Their *initial values* at loop entry correspond to the actual parameters in the top-level call of the recursive procedure.
    - Variables referenced in the loop but not updated may also become parameters
      - The recursive procedure begins by mimicking the while condition and returns if while condition is false
        - a break also corresponds to a procedure return
      - Continue by updating variable and make recursive call
Removing While Loop: n Example

- `int factLoop(int n)`
- `int k=1; int f = 1`
- `while (k <= n)`
- `int fnew = f*k;`
- `int knew = k+1`
- `k = knew; f = fnew;`
- `return f;`

```c
int factLoop(int n)
{
    return factRec(n, 1, 1);
}

int factRec(int n, int k, int f)
{
    if (k <= n)
    {
        int fnew = f * k;
        int knew = k + 1
        return factRec (n, knew, fnew)
    }
    return f;
}
```
Removing For Loop: An Example

• Convert the following seqSearch
  – to recursive procedure without loop

int seqSearch (int[] E, int num, int K)
1. int ans, index;
2. ans = -1; // Assume failure.
3. for (index = 0; index < num; index++)
4.   if (K == E[index])
5.     ans = index; // Success!
6.     break; // Done!
7. return ans;
Recursive Procedure Without Loops e.g.

- Call with: `seqSearchRec(E, 0, num, K)`

```
seqSearchRec(E, index, num, K)
1:   if (index >= num)
2:       ans = -1;
3:   else if (E[index] == K) // index < num
4:       ans = index;
5:   else
6:       ans = seqSearchRec(E, index+1, num, K);
7:   return ans;
```

Compare to: `for (index = 0; index < num; index++)`
Analyzing Recursive Procedure using Recurrence Equations

- Let \( n \) be the size of the problem
- Worst-Case Analysis (for procedure with no loops)
- \( T(n) = \)
  - the individual cost for a sequence of blocks
  - add the maximum cost for an alternation of blocks
  - add the cost of subroutine call, \( S( f(n) ) \)
  - add the cost of recursive procedure call, \( T( g(n) ) \)
- e.g. seqSearchRec,
  - Basic operation is comparison of array element, cost 1
  - statement: \( 1: + \max(2:, (3: + \max(4:, (5: + 6:))) + (7:) \)
  - Cost: \( 0 + \max(0, (1 + \max(0, (0+T(n-1)))) ) + 0 \)
- \( T(n) = T(n-1) + 1; \quad T(0) = 0 \)
- \( \Rightarrow T(n) = n; \quad T(n) \in \theta(n) \)
Evaluate Recursive Equation using Recursion Tree

- Evaluate: \( T(n) = T(n/2) + T(n/2) + n \)
  - Work copy: \( T(k) = T(k/2) + T(k/2) + k \)
  - For \( k = n/2 \), \( T(n/2) = T(n/4) + T(n/4) + (n/2) \)
- [size|cost]
Recursion Tree: An Example

- To evaluate the total cost of the recursion tree
  - sum all the non-recursive costs of all nodes
    \[ = \text{Sum (rowSum (cost of all nodes at the same depth))} \]
- Determine the maximum depth of the recursion tree:
  - For our example, at tree depth \( d \)
    - the size parameter is \( n/(2^d) \)
    - the size parameter converging to base case, i.e. case 1
    - such that, \( n/(2^d) = 1 \),
    - \( d = \lg(n) \)
    - The rowSum for each row is \( n \)
- Therefore, the total cost, \( T(n) = n \lg(n) \)
Proving Correctness of Procedures

- Things should be made as simple as possible – but not simpler
  - Albert Einstein

- Proving Correctness of procedures is a difficult task in general; the trick is to make it as simple as possible.
  - No loops is allowed in the procedure!
  - Variable is assigned a value only once!

- Loops are converted into Recursive procedures.
- Additional variables are used to make single-assignment (write-once read many) possible.
  - \( x = y+1 \) does imply the equation \( x = y+1 \) for entire time
General Correctness Lemma

- If all preconditions hold when the block is entered,
  - then all postconditions hold when the block exits
- And, the procedure will terminate!
  - Chains of Inference: Sequence
Proving Correctness of Binary Search, e.g.

```c
int binarySearch (int[] E, int first, int last, int K)
1.   if (last < first)
2.        index = -1;
3. else
4.   int mid = (first + last) / 2
5.   if (K == E[mid])
6.        index = mid;
7. else if (K < E[mid])
8.        index = binarySearch (E, first, mid-1, K)
9. else
10.       index = binarySearch (E, mid+1, last, K);
11. return index
```
Proving Correctness of Binary Search

• Lemma  (preconditions => postconditions)
  – if binarySearch(E, first, last, K) is called, and
    the problem size is n = (last – first + 1),
    for all n >= 0, and
    E[first], …, E[last] are in nondecreasing order,
  – then it returns –1 if K does not occur in E within the range
    first, …, last, and
    it returns index such that K=E[index] otherwise

• Proof
  – The proof is by induction on n, the problem size.
  – The base case in n = 0.
  – In this case, line 1 is true, line 2 is reached, and –1 is
    returned. (the postcondition is true)
Inductive Proof, continue

• For \( n > 0 \), assume that binarySearch\((E, \text{first}, \text{last}, K)\) satisfies the lemma on problems of size \( k \), such that \( 0 \leq k < n \), and first and last are any indexes such that \( k = \text{last} - \text{first} + 1 \)
  
  – For \( n > 0 \), line 1 is false, … mid is within the search range \((\text{first} \leq \text{mid} \leq \text{last})\).
  
  – If line 5 is true, the procedure terminates with index = mid. \((\text{the postcondition is true})\)
  
  – If line 5 is false, from \((\text{first} \leq \text{mid} \leq \text{last})\) and def. of \( n \),
    \[ (\text{mid} - 1) - \text{first} + 1 \leq (n - 1) \]
    \[ \text{last} - (\text{mid} + 1) + 1 \leq (n - 1) \]
  
  – so the inductive hypothesis applies for both recursive calls,
Inductive Proof, continue

– If line 7 is true, … the preconditions of binarySearch are satisfied, we can assume that the call accomplishes the objective.
– If line 8 return positive index, done.
– If line 8 returns –1, this implies that K is not in E in the first … mid-1, also since line 7 is true, K is not in E in range min… last, so returning – 1 is correct (done).
– If line 7 is false, … similar the postconditions are true. (done!)
Design and Analysis of Algorithms

Divide and Conquer

• It is an approach to partition the problem into smaller parts
• find solutions for the parts
• then combine the solutions for the parts into a solution for the whole.
• This approach yields efficient solutions when used recursively.
Divide and Conquer: An Example

**Problem Statement**
- Finding the maximum and minimum elements of a set

**Input**
- A set $S$ with $n$ elements, where $n$ is a power of 2 and $n \geq 2$.

**Output**
- The maximum and minimum of $S$

**Strategy A**
- A procedure to find the maximum and
- A procedure to find the minimum
Divide and Conquer: An Example

**Algorithm A**

```c
int MAX (int[] S, int n)
{
    max = S[0];
    for (int index = 1; index < n; index++)
        if (S[index] > max)
            max = S[index];
    return max;
}

int MIN (int[] S, int n)
{
    min = S[0];
    for (int index = 1; index < n; index++)
        if (S[index] < min)
            min = S[index];
    return min;
}
```
Divide and Conquer: An Example

**Analysis A**
MAX function finds the maximum element in \( n-1 \) comparisons. MIN function finds the minimum element in \( n-1 \) comparisons. The total comparisons is \( 2n - 2 \), for \( n \geq 2 \).

• Can this problem be solved in a better way? Yes.

**Strategy B**
• A recursive procedure using the divide and conquer technique is used.
• Set \( S \) is divided into \( S_1 \) and \( S_2 \), each of size \( n/2 \).
• The maximum and minimum elements are found for each subset by recursive applications of the algorithm.
Divide and Conquer: An Example

Algorithm B
void MaxMin (int[] S, int n)
1. if ( n == 2 )
2. let S = { a, b };
3. return ( MAX (a, b), MIN (a, b) );
else
4. divide S into subsets S1 and S2, each with half the elements
5. (max1, min1) ← MaxMin (S1, n/2);
6. (max2, min2) ← MaxMin (S2, n/2);
7. return (MAX (max1, max2), MIN (min1, min2))
Divide and Conquer: An Example

Analysis B

The basic operation is the comparisons in step 3 and in step 7.

• Let $A(n)$ be the number of comparisons between elements of $S$ required by MaxMin to find the maximum and minimum elements in a set of $n$ elements.

• Clearly, $A(2) = 1$

• If $n > 2$, $A(n)$ is the total number of comparisons used in the two calls of MaxMin on sets of size $n/2$, plus the two comparisons from line 7.
Divide and Conquer: An Example

That is:

\[
A(n) = \begin{cases} 
1, & \text{for } n = 2 \\
2 A(n/2) + 2, & \text{for } n > 2 
\end{cases}
\]  \tag{1}

• The function \( A(n) = \frac{3}{2} n - 2 \) is a solution to recurrence (1).
• This solution satisfies \( n = 2 \).
• If it satisfies (1) for \( n = m \), where \( m \geq 2 \), then

\[
A(2m) = 2 \left( \frac{3m}{2} - 2 \right) + 2 = \frac{3}{2} (2m) - 2
\]
Divide and Conquer: An Example

• Thus it satisfies (1) for $n = 2^m$.
• Thus, by induction on $n$, we have shown that $A(n) = \frac{3}{2} n - 2$ satisfies (1) whenever $n$ is a power of 2.

Optimality
• We can show that at least $\frac{3}{2} n - 2$ comparisons between elements of $S$ are necessary to find the maximum and minimum elements in a set $S$ of $n$ elements.
• Thus algorithm B is optimal with respect to the number of comparisons.

Conclusion:
The divide and conquer approach reduced the number of comparisons by a constant factor.
Divide and Conquer: An Example

That is:
Algorithm Design Technique: Divide and Conquer

– It is often easier to solve several small instances of a problem than one large one.
– This technique is an approach to
  • divide the problem into smaller instances of the same problem
  • solve (conquer) the smaller instances recursively
  • combine the solutions to obtain the solution for original input

General Procedure:
Solve(I)
    n = size(I)
    if (n <= smallsize)
        solution = directlySolve(I);
    else
        divide I into I1, …, Ik.
        for each i in {1, …, k}
            Si = solve(Ii);
        solution = combine(S1, …, Sk);
        return solution;
Divide and Conquer: Recurrence Equation

- We must specify the subroutines
  - `directlySolve`
  - `divide`
  - `combine`
- The smaller instances into which the input is divided is k.
- For any input of size n,
  - let $B(n)$ be the number of steps done by `directlySolve`,
  - let $D(n)$ be the number of steps done by `divide`,
  - let $C(n)$ be the number of steps done by `combine`.
- Then the general form of the recurrence equation that describes the amount of work done by the algorithm is

\[
T(n) = D(n) + \sum_{i=1}^{k} T(\text{size}(I_i)) + C(n) \quad \text{for } n > \text{smallSize}
\]

with base case $T(n) = B(n)$ for $n \leq \text{smallSize}$
Divide and Conquer: Recurrence Equation

- For many divide-and-conquer algorithms, either the divide step or the combine step is very simple, and the recurrence equation for $T$ is simpler than the general form.
- Many algorithms can be designed using this technique.
- We will see later some sorting algorithms:
  - Quicksort and Mergesort are two sorting algorithms that are differ in the way they divide the problem and later combine the solutions.
  - Quicksort is characterized as “hard division, easy combination”
  - Mergesort is characterized as “easy division, hard combination”.
Divide and Conquer: An Example

Problem Statement
• Finding the maximum and minimum elements of a set

Input
• A set $S$ with $n$ elements, where $n$ is a power of 2 and $n \geq 2$.

Output
• The maximum and minimum of $S$

Strategy A
• A procedure to find the maximum and
• A procedure to find the minimum
**Divide and Conquer: An Example**

**Algorithm A**

```c
int MAX (int[] S, int n)
    max = S[0];
    for (int index = 1; index < n; index ++)
        if (S [index] > max)
            max = S [index];
    return max;

int MIN (int[] S, int n)
    min = S[0];
    for (int index = 1; index < n; index ++)
        if (S [index] < min)
            min = S [index];
    return min;
```

Design and Analysis of Algorithms
by Dr. Nadia Y. Yousif
Divide and Conquer: An Example

**Analysis A**
MAX function finds the maximum element in $n-1$ comparisons.
MIN function finds the minimum element in $n-1$ comparisons.
The total comparisons is $2n - 2$, for $n \geq 2$.

- Can this problem be solved in a better way? Yes.

**Strategy B**
- A recursive procedure using the divide and conquer technique is used.
- Set $S$ is divided into $S_1$ and $S_2$, each of size $n/2$.
- The maximum and minimum elements are found for each subset by recursive applications of the algorithm.
Divide and Conquer: An Example

**Algorithm B**
void MaxMin (int[] S, int n)
1. if ( n == 2 )
2. let S = { a, b };
3. return ( MAX (a, b), MIN (a, b) );
else
4. divide S into subsets S1 and S2, each with half the elements
5. (max1, min1) ← MaxMin (S1, n/2);
6. (max2, min2) ← MaxMin (S2, n/2);
7. return (MAX (max1, max2), MIN (min1, min2))
Divide and Conquer: An Example

Analysis B
The basic operation is the comparisons in step 3 and in step 7.

- Let $A(n)$ be the number of comparisons between elements of $S$ required by MaxMin to find the maximum and minimum elements in a set of $n$ elements.

- Clearly, $A(2) = 1$

- If $n > 2$, $A(n)$ is the total number of comparisons used in the two calls of MaxMin on sets of size $n/2$, plus the two comparisons from line 7.
Analysis of the Algorithm

That is:

\[
A(n) = \begin{cases} 
1, & \text{for } n = 2 \\
2A(n/2) + 2, & \text{for } n > 2 
\end{cases}
\]  

(1)

• The function \( A(n) = \frac{3}{2}n - 2 \) is a solution to recurrence (1).
• This solution satisfies \( n = 2 \).
• If it satisfies (1) for \( n = m \), where \( m \geq 2 \), then

\[
A(2m) = 2\left(\frac{3m}{2} - 2\right) + 2 = \frac{3}{2}(2m) - 2
\]
Divide and Conquer: An Example

• Thus it satisfies (1) for $n = 2^m$.

• Thus, by induction on $n$, we have shown that
  $A(n) = 3/2 \cdot n - 2$ satisfies (1) whenever $n$ is a power of 2.

Optimality

• We can show that at least $3/2 \cdot n - 2$ comparisons between elements of $S$ are necessary to find the maximum and minimum elements in a set $S$ of $n$ elements.

• Thus algorithm B is optimal with respect to the number of comparisons.

Conclusion:
The divide and conquer approach reduced the number of comparisons by a constant factor.
QuickSort Algorithm

- Published by C. A. R. Hoare in 1962
- It is still one of the fastest in practice

QuickSort Strategy:
- Rearrange the elements to be sorted so that all the “small” keys precede the “large” keys in the array (the “hard division” part).
- QuickSort sorts the two subranges of “small” and “large” recursively, with the result that the entire array is sorted.
- For an array implementation there is nothing to do in the “combination” step, but in case of list implementation, the “combination” step concatenates the to lists.
Quicksort Algorithm Description

• Let E be the array of elements and let first and last be the indexes of the first and last entries, respectively.

• first = 0 and last = n-1, where n is the number of elements.

• The “divide” process of Quicksort algorithm
  - chooses an element, called the pivot element, whose key is called the pivot, from the subrange that it must sort
  - moves the pivot element to a local variable, leaving a vacancy in the array. Assume that the leftmost element in the subrange is chosen as the pivot element
Quicksort Algorithm Description

- passes the pivot (the key field only) to the Partition subroutine, which rearranges the other elements, finding an index splitPoint such that:
  1- for first ≤ i < splitPoint, E(i).key < pivot
  2- for splitPoint < i ≤ last, E(i).key ≥ pivot

Now there is a vacancy at splitPoint.

- deposits the pivot element in E[splitPoint], which is the correct position.

This completes the “divide” process, and Quicksort continues by calling itself recursively to solve the two subproblems created by Partition.
Quicksort Algorithm Description

- **first**
  - pivot
  - last
  - partition
  - first
  - splitPoint
  - last
  - < pivot
  - pivot
  - >= pivot

Sort recursively by Quicksort

Sort recursively by Quicksort
**Quicksort Algorithm**

- **Input:** Array \( E \) and indexes \( \text{first} \), and \( \text{last} \), such that elements \( E[i] \) are defined for \( \text{first} \leq i \leq \text{last} \).
- **Output:** \( E[\text{first}], \ldots, E[\text{last}] \) is sorted rearrangement of the same elements.

```c
void quickSort (Element [ ] E, int first, int last) {
    if (first < last) {
        Element pivotElement = E[first];
        Key pivot = pivotElement.key;
        int splitPoint = partition (E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort (E, first, splitPoint - 1);
        quickSort (E, splitPoint + 1, last);
    }
}
```
The Partition Part of Quicksort Algorithm

- There are many different strategies that may be used by Partition: they yield algorithms with different advantages and disadvantages.
- The following is the method originally described by Hoare
- **Input:**
  - Array E,
  - pivot, the key around which to partition,
  - indexes first, and last (assume that first < last), such that elements E[i] are defined for first + 1 ≤ i ≤ last and E[first] is vacant.
- **Output:**
  - Let splitPoint be the returned value.
  - The Elements originally in first + 1, …, last are rearranged into two subranges such that
    1- the keys of E[first], … E[splitPoint - 1] are less than pivot
    2- the keys of E[splitPoint + 1], …, E[last] are greater than or equal to pivot.
  - Also, first ≤ splitPoint ≤ last, and E[splitPoint] is vacant.
- **Procedure:** See next slide.
The Partition Procedure

int partition (Element [ ] E, Key pivot, int first, int last)
{
    int low, high;
    low = first;   high = last;
    while (low < high)
    {
        int highVac = extendLargeRegion(E, pivot, low, high);
        int lowVac = extendSmallRegion(E, pivot, low+1, highVac);
        low = lowVac;   high = highVac – 1;
    }
    return low;   // This is the splitPoint.
}

/** postcondition for extendLargeRegion:
 * The right most element in E[lowVac + 1], …, E[high]
 * whose key is < pivot is moved to E[lowVac] and the index from which it
 * was moved is returned. If there is no such element, lowVac is returned.
 */
The Partition Procedure .. (Cont.)

```c
int extendLargeRegion (Element [ ] E, Key pivot, int lowVac, int high);
{
    int highVac, curr;
    highVac = lowVac;   // In case no key < pivot.
    curr = high;
    while (curr > lowVac)
    {
        if (E[curr].key < pivot)
            { E[lowVac] = E[curr];
              highVac = curr;
              break;}
        curr -- ;    // keep looking
    }
    return highVac;
}
```
/** Postcondition for extendSmallRegion: (Exercise) */

int extendSmallRegion (Element [ ] E, Key pivot, int low, int highVac)
{
    int lowVac, curr;
    lowVac = highVac; // in case no key ≥ pivot.
    curr = low;
    while ( curr < highVac )
    {
        if ( E[curr].key ≥ pivot)
            { E[highVac] = E[curr];
              lowVac = curr;
              break;
            }
        curr ++ ; // keep looking
    }

    return  lowVac;
}
Analysis of QuickSort

- **Worst Case:**
  - Partition procedure compares each key to pivot; if there are \( k \) positions in the range of the array it is working on, then it does \( k - 1 \) key comparisons (the first position is vacant).
  - If \( E[\text{first}] \) has the smallest key in the range being split, then \( \text{splitPoint} = \text{first} \), and all the splitting process will result in splitting the range into an empty subrange and a subrange with \( k - 1 \) elements.
  - If pivot is the smallest key each time Partition is called, then the total number of comparisons done is

\[
\sum_{k=2}^{n} (k - 1) = \frac{n(n - 1)}{2}
\]

This is as in Insertion sort (to be discussed later).
Analysis of QuickSort

• **Average Case:**
  - We assume that the key are distinct and that all permutations of the keys are equally likely.
  - Let $k$ be the number of elements in the range of the array being sorted, and let $A(k)$ be the average number of key comparisons done for ranges of this size.
  - Suppose the next time Partition is executed pivot gets put in the $ith$ position in this subrange counting from 0.
  - Partition does $k - 1$ key comparisons, and the subranges to be sorted next have $I$ elements and $k - 1$ elements, respectively.
  - Each possible position for the split point $I$ is equally likely (has probability $1/k$) so, letting $k = n$, we have the recurrence equation:

\[
A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n} (A(i) + A(n-1-i)) \quad \text{for} \quad n \geq 2
\]

\[
A(1) = A(0) = 0
\]
Analysis of QuickSort

• To simplify the equation:
  - The terms of A(n-1-i) run from A(n-1) down to A(0), so their sum is the same as the sum of A(i) terms. Then we drop the A(0) terms, giving

    \[ A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \quad \text{for} \quad n \geq 1 \]

  - Solving this equation by guess or by induction.

    \[ A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \leq n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} ci \ln(i) \]

  We can bound the sum by integration

    \[ \sum_{i=1}^{n-1} ci \ln(i) \leq c \int_{1}^{n} x \ln x \, dx \]

  This yields into:

    \[ \int_{1}^{n} x \ln x \, dx = \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 \]
Analysis of QuickSort

So

\[ A(n) \leq n - 1 + \frac{2c}{n} \left( \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 \right) \]

\[ = cn \ln n + n \left( 1 - \frac{c}{2} \right) - 1 \]

To show that \( A(n) \leq cn \ln n \), it suffices to show that the second and third terms are negative or zero.

The second term is less than or equal to zero for \( c \geq 2 \).

So we can let \( c = 2 \) and conclude that

\( A(n) \leq 2n \ln n \)

Similarly, for \( c < 2 \), we have \( A(n) > cn \ln n \)

Corollary:

The number of comparisons done by Quicksort algorithm on sets of size \( n \) is approximately

\[ 1.386 n \log n \], for large \( n \)
Using Divide and Conquer: Mergesort

- Mergesort Strategy

```
\lfloor(\text{first} + \text{last})/2\rfloor
```

```
\text{first} \quad \rightarrow \quad \text{last}
```

```
\text{Sort recursively by Mergesort} \quad \rightarrow \quad \text{Sort recursively by Mergesort}
```

```
\text{Sorted} \quad \rightarrow \quad \text{Sorted}
```

```
\text{Merge} \quad \rightarrow \quad \text{Sorted}
```

Algorithm: Mergesort

- Input: Array E and indexes first, and Last, such that the elements E[i] are defined for first <= i <= last.
- Output: E[first], …, E[last] is sorted rearrangement of the same elements

- void mergeSort(Element[] E, int first, int last)
  - if (first < last)
    - int mid = (first+last)/2;
    - mergeSort(E, first, mid);
    - mergeSort(E, mid+1, last);
    - merge(E, first, mid, last);
  - return;

- $W(n) = W(n/2) + W(n/2) + W_{\text{merge}}(n) \in \theta(n \log n)$
  - $W_{\text{merge}}(n) = n-1$
  - $W(1) = 0$
Merging Sorted Sequences

• Problem:
  – Given two sequences A and B sorted in nondecreasing order, merge them to create one sorted sequence C

• Strategy:
  – determine the first item in C: It is the minimum between the first items of A and B. Suppose it is the first items of A. Then, rest of C consisting of merging rest of A with B.
Algorithm: Merge

- Merge(A, B, C)
  - if (A is empty)
    - rest of C = rest of B
  - else if (B is empty)
    - rest of C = rest of A
  - else if (first of A <= first of B)
    - first of C = first of A
    - merge (rest of A, B, rest of C)
  - else
    - first of C = first of B
    - merge (A, rest of B, rest of C)
  - return

\[ W(n) = n - 1 \]
Graphs and Graph Traversals

// From Tree to Graph
// Many programs can be cast as problems on graph
Definitions and Representations
Traversing Graphs
Depth-First Search on Directed Graphs
Strongly Connected Components of a Directed Graph
Depth-First Search on Undirected Graphs
Biconnected Components of an Undirected Graph
Problems: e.g. Airline Routes
Problems: e.g. Flowcharts

(a) Flowchart

(b) Directed graph
Problems: e.g. Binary relation

- x is a proper factor of y
Problems: e.g. Computer Networks

(a) A star network
(b) A ring network
Definition: Directed graph

- **Directed Graph**
  - A directed graph, or digraph, is a pair
  - \( G = (V, E) \)
  - where \( V \) is a set whose elements are called vertices, and
  - \( E \) is a set of ordered pairs of elements of \( V \).

- Vertices are often also called nodes.
- Elements of \( E \) are called edges, or directed edges, or arcs.
- For directed edge \((v, w)\) in \( E \), \( v \) is its tail and \( w \) its head;
- \((v, w)\) is represented in the diagrams as the arrow, \( v \rightarrow w \).
- In text we simply write \( vw \).
Definition: Undirected graph

- **Undirected Graph**
  - A undirected graph is a pair 
  - \( G = (V, E) \)
  - where \( V \) is a set whose elements are called vertices, and
  - \( E \) is a set of *unordered* pairs of *distinct* elements of \( V \).

  - Vertices are often also called nodes.
  - Elements of \( E \) are called edges, or undirected edges.
  - Each edge may be considered as a subset of \( V \) containing two elements,
  - \( \{v, w\} \) denotes an undirected edge
  - In diagrams this edge is the line \( v \)--\( w \).
  - In text we simply write \( vw \), or \( wv \)
  - \( vw \) is said to be *incident* upon the vertices \( v \) and \( w \)
Definitions: Weighted Graph

- A weighted graph is a triple \((V, E, W)\)
  - where \((V, E)\) is a graph (directed or undirected) and
  - \(W\) is a function from \(E\) into \(\mathbb{R}\), the reals (integer or rationals).
  - For an edge \(e\), \(W(e)\) is called the weight of \(e\).
Graph Representations using Data Structures

- **Adjacency Matrix Representation**
  - Let $G = (V,E)$, $n = |V|$, $m = |E|$, $V = \{v_1, v_2, \ldots, v_n\}$
  - $G$ can be represented by an $n \times n$ matrix

(a) An undirected graph

(b) Its adjacency matrix

$$
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
$$
Adjacency Matrix for weight digraph

(a) A weighted digraph
(b) Its adjacency matrix
Array of Adjacency Lists Representation

- From
  - to

(a) An undirected graph
(b) Its adjacency matrix

adjVertices

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, nil</td>
</tr>
<tr>
<td>2</td>
<td>1, 3, 4, nil</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 6</td>
</tr>
<tr>
<td>5</td>
<td>6, nil</td>
</tr>
<tr>
<td>6</td>
<td>3, 4, 5, nil</td>
</tr>
<tr>
<td>7</td>
<td>6, nil</td>
</tr>
</tbody>
</table>
Array of Adjacency Lists Representation

(a) A weighted digraph

from -> to, weight
More Definitions

- Subgraph
- Symmetric digraph
- complete graph
- Adjacency relation
- Path, simple path, reachable
- Connected, Strongly Connected
- Cycle, simple cycle
- acyclic
- undirected forest
- free tree, undirected tree
- rooted tree
- Connected component
Traversing Graphs

• Most algorithms for solving problems on a graph examine or process each vertex and each edge.

• **Breadth-first search and depth-first search**
  – are two traversal strategies that provide an efficient way to “visit” each vertex and edge exactly once.

• **Breadth-first search: Strategy (for digraph)**
  – choose a starting vertex, distance \( d = 0 \)
  – vertices are visited in order of increasing distance from the starting vertex,
  – examine all edges leading from vertices (at distance \( d \)) to adjacent vertices (at distance \( d+1 \))
  – then, examine all edges leading from vertices at distance \( d+1 \) to distance \( d+2 \), and so on,
  – until no new vertex is discovered
Breath-first search, e.g.

- e.g. Start from vertex A, at d = 0
  - visit B, C, F; at d = 1
  - visit D; at d = 2
- e.g. Start from vertex E, at d = 0
  - visit G; at d = 1
**Breadth-first search: I/O Data Structures**

**Input:** $G = (V, E)$, a graph represented by an adjacency list structure, `adjVertices`, as described in Section 7.2.3, where $V = \{1, \ldots, n\}; s \in V$, the vertex from which the search begins.

**Output:** A breadth-first spanning tree, stored in the parent array. The parent array is passed in and the algorithm fills it.

**Remarks:** For a queue $Q$, we assume operations of the Queue abstract data type (Section 2.4.2) are used. The array `color[1], \ldots, color[n]` denotes the current search status of all vertices. Undiscovered vertices are white; those that are discovered but not yet processed (in the queue) are gray; those that are processed are black.
Breadth-first search: Algorithm

```java
void breadthFirstSearch(IntList[] adjVertices, int n, int s, int[] parent) {
    int[] color = new int[n+1];
    Queue pending = create(n);
    Initialize color[1], ..., color[n] to white.

    parent[s] = -1;
    color[s] = gray;
    enqueue(pending, s);
    while (pending is nonempty)
        v = front(pending);
        dequeue(pending);
        For each vertex w in the list adjVertices[v]:
            if (color[w] == white)
                color[w] = gray;
                enqueue(pending, w);
                parent[w] = v; // Process tree edge vw.
        // Continue through list.
        // Process vertex v here.
        color[v] = black;
    return;
}```
Breadth-first search: Analysis

- For a digraph having n vertices and m edges
  - Each edge is processed once in the while loop for a cost of $\Theta(m)$
  - Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\Theta(n)$
  - Extra space is used for color array and queue, there are $\Theta(n)$

- From a tree (breadth-first spanning tree)
  - the path in the tree from start vertex to any vertex contains the minimum possible number of edges

- Not all vertices are necessarily reachable from a selected starting vertex
Depth-first search for Digraph

- **Depth-first search: Strategy (for digraph)**
  - choose a starting vertex, distance $d = 0$
  - vertices are visited in order of increasing distance from the starting vertex,
  - examine One edges leading from vertices (at distance $d$) to adjacent vertices (at distance $d+1$)
  - then, examine One edges leading from vertices at distance $d+1$ to distance $d+2$, and so on,
  - until no new vertex is discovered, or dead end
  - then, backtrack one distance back up, and try other edges, and so on
  - until finally backtrack to starting vertex, with no more new vertex to be discovered.
Depth-first search for digraph, e.g. from a tree, remember where we have been

and so on...
Depth-first Search, e.g. trace it, in order

- Vertex status: undiscovered, discovered, finished
- Edge status: exploring, backtrack, checked
Depth-first search tree

- edges classified:
  - tree edge, back edge, descendant edge, and cross edge
Depth-first search algorithm: outline

dfs(G, v)  // OUTLINE
         Mark v as “discovered.”
         For each vertex w such that edge vw is in G:
               If w is undiscovered:
                   dfs(G, w); that is, explore vw, visit w, explore from there
                   as much as possible, and backtrack from w to v.
               Otherwise:
                   “Check” vw without visiting w.
         Mark v as “finished.”
Reaching all vertices

dfsSweep(G) // OUTLINE
Initialize all vertices of G to “undiscovered.”
For each vertex $v \in G$, in some order:
  If $v$ is undiscovered:
    dfs(G, v); that is, perform a depth-first search beginning
    (and ending) at $v$; any vertices discovered during an earlier
    depth-first search visit are not revisited; all vertices visited
    during this dfs are now classified as “discovered.”
Depth-first search algorithm

```c
int dfs(IntList[] adjVertices, int[] color, int v, ...)
    int w;
    IntList remAdj;
    int ans;
1. color[v] = gray;
2. Preorder processing of vertex v
3. remAdj = adjVertices[v];
4. while (remAdj ≠ nil)
5.     w = first(remAdj);
6.     if (color[w] == white)
7.         Exploratory processing for tree edge vw
8.         int wAns = dfs(adjVertices, color, w, ...);
9.         Backtrack processing for tree edge vw. using wAns (like inorder)
10.    else
11.        Checking (i.e., processing) for nontree edge vw
12.        remAdj = rest(remAdj)
13. Postorder processing of vertex v. including final computation of ans
14. color[v] = black;
15. return ans;
```
Strongly Connected Components of a Digraph

- Strongly connected:
  - A directed graph is strongly connected if and only if, for each pair of vertices v and w, there is a path from v to w.

- Strongly connected component:
  - A strongly connected component of a digraph G is a maximal strongly connected subgraph of G.
Strongly connected Components and Equivalence Relations

- Strongly Connected Components may be defined in terms of an equivalence relation, \( S \), on the vertices
  - \( vSw \) iff there is a path from \( v \) to \( w \) and
  - a path from \( w \) to \( v \)
- Then, a strongly connected component consists of one equivalence class, \( C \), along with all edges \( vw \) such that \( v \) and \( w \) are in \( C \).
Condensation graph

- The strongly connected components of a digraph can each be collapsed to a single vertex yielding a new digraph that has no cycles.

- **Condensation graph:**
  - Let $S_1, S_2, \ldots, S_p$ be the strong components of $G$.
  - The condensation graph of $G$ denoted as $G^{\downarrow}$, is the digraph $G^{\downarrow} = (V', E')$,
  - where $V'$ has $p$ elements $s_1, s_2, \ldots, s_p$ and
  - $s_is_j$ is in $E'$ if and only if $i \neq j$ and
  - there is an edge in $E$ from some vertex in $S_i$ to some vertex in $S_j$. 

Design and Analysis of Algorithms
Condensation graph and its strongly connected components

- Condensation Graph is acyclic.

(a) The digraph  (b) Its strong components  (c) Its condensation graph
Algorithm to Find Strongly Connected Component

• Strategy:
  – Phase 1:
  – A standard depth-first search on $G$ is performed,
  – and the vertices are put in a stack at their finishing times
  – Phase 2:
  – A depth-first search is performed on $G^T$, the transpose graph.
  – To start a search, vertices are popped off the stack.
  – A strongly connected component in the graph is identified by the name of its starting vertex (call leader).
The strategy in Action, e.g.
Depth-First Search on Undirected Graphs

- Depth-first search on an undirected graph is complicated by the fact that edges should be explored in one direction only,
- but they are represented twice in the data structure (symmetric digraph equivalence)
- Depth-first search provides an orientation for each of its edges
  - they are oriented in the direction in which they are first encountered (during exploring)
  - the reverse direction is then ignored.
Algorithm for depth-first search on undirected graph

int dfs(IntList[] adjVertices, int[] color, int v, int p, ...)
    int w;
    IntList remAdj;
    int ans;
1. color[v] = gray;
2. Preorder processing of vertex v
3. remAdj = adjVertices[v];
4. while (remAdj <> nil)
5. w = first(remAdj);
6. if (color[w] == white)
7. Exploratory processing for tree edge vw.
8. int wAns = dfs(adjVertices, color, w, v,...)
9. BackTrack processing for tree edge vw using wAns (like inorder)
10. else if (color[w] == gray && w <> p)
11. Checking back edge vw
    // else vw was traversed, so ignore vw.
12. remAdj = rest(remAdj)
13. Postorder processing of vertex v, including final computation of ans
14. color[v] = black;
33. return ans;
Breadth-first Search on Undirected Graph

• Simply treat the undirected graph as symmetric digraph
  – in fact undirected graph is represented in adjacency list as symmetric digraph
• Each edge is processed once in the “forward” direction
  – whichever direction is encountered (explored) first is considered “forward” for the duration of the search
Bi-connected components of an Undirected graph

• Problem:
  – If any *one* vertex (and the edges incident upon it) are removed from a connected graph,
  – is the remaining subgraph still connected?

• Biconnected graph:
  – A connected undirected graph G is said to be biconnected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.

• Biconnected component:
  – A biconnected component of a undirected graph is a maximal biconnected subgraph, that is, a binconnected subgraph not contained in any larger binconnected subgraph.

• Articulation point:
  – A vertex v is an articulation point for an undirected graph G if there are distinct vertices w and x (distinct from v also) such that v is in every path from w to x.
Bi-connected components, e.g.

- Some vertices are in more than one component (which vertices are these?)
Graph Optimization Problems and Greedy Algorithms

- To study many graph optimization problems that can be solved exactly by greedy algorithms.

- **Greedy Algorithms**
  - Make the best choice now!

- **Optimization Problems**
  - Minimizing Cost or Maximizing Benefits
  - Minimum Spanning Tree
    - Minimum cost for connecting all vertices
  - Single-Source Shortest Paths
    - Shortest Path between two vertices
Greedy Algorithms: Make the best choice now!

- Making choices in sequence such that
  - each individual choice is best
    - according to some limited “short-term” criterion,
    - that is not too expensive to evaluate
  - once a choice is made, it cannot be undone!
    - even if it becomes evident later that it was a poor choice
- Make progress by choosing an action that
  - incurs the minimum short-term cost,
  - in the hope that a lot of small short-term costs add up to small overall cost.
- Possible drawback:
  - actions with a small short-term cost may lead to a situation, where further large costs are unavoidable.
Optimization Problems

• Minimizing the total cost or
  Maximizing the total benefits
  – Analyze all possible outcomes and find the best, or
  – Make a series of choices whose overall effect is to achieve
    the optimal.

• Some optimization problems can be solved exactly by greedy
  algorithms
  – Minimum cost for connecting all vertices
    • Minimum Spanning Tree Algorithm
  – Shortest Path between two vertices
    • Single-Source Shortest Paths Algorithm
Minimum Spanning Tree

- A spanning tree for a connected, undirected graph, $G=(V,E)$ is
  - a subgraph of $G$ that is
  - an undirected tree and contains
  - all the vertices of $G$.
- In a weighted graph $G=(V,E,W)$, the weight of a subgraph is
  - the sum of the weights of the edges in the subgraph.
- A minimum spanning tree for a weighted graph is
  - a spanning tree with the minimum weight.
Prim’s Minimum Spanning Tree Algorithm

• Select an arbitrary starting vertex, (the root)
• branches out from the tree constructed so far by
  – choosing an edge at each iteration
  – attach the edge to the tree
    • that edge has minimum weight among all edges that can be attached
  – add to the tree the vertex associated with the edge
• During the course of the algorithm, vertices are divided into three disjoint categories:
  – Tree vertices: in the tree constructed so far,
  – Fringe vertices: not in the tree, but adjacent to some vertex in the tree,
  – Unseen vertices: all others
The Algorithm in action, e.g.

(a) A weighted graph

(b) After selection of the starting vertex

(c) BG was considered but did not replace AG as a candidate.

(d) After AG is selected and fringe and candidates are updated

(e) IF has replaced AF as a candidate.

(f) After several more passes: The two candidate edges will be put in the tree.
Prim's Minimum Spanning Trees: Outline

\texttt{primMST}(G, n) // OUTLINE

Initialize all vertices as \textit{unseen}.
Select an arbitrary vertex \textit{s} to start the tree; reclassify it as \textit{tree}.
Reclassify all vertices adjacent to \textit{s} as \textit{fringe}.
While there are fringe vertices:
   Select an edge of minimum weight between a tree vertex \textit{t} and a
   fringe vertex \textit{v};
   Reclassify \textit{v} as \textit{tree}; add edge \textit{tv} to the tree;
   Reclassify all \textit{unseen} vertices adjacent to \textit{v} as \textit{fringe}. 

Properties of Minimum Spanning Trees

- Definition: Minimum spanning tree property
  - Let a connected, weighted graph $G = (V, E, W)$ be given, and let $T$ be any spanning tree of $G$.
  - Suppose that for every edge $vw$ of $G$ that is not in $T$,
    - if $uv$ is added to $T$, then it creates a cycle
    - such that $uv$ is a maximum-weight edge on that cycle.
  - Then the tree $T$ is said to have the minimum spanning tree property.
Properties of Minimum Spanning Trees

• Lemma:
  – In a connected, weighted graph $G = (V, E, W)$,
  – if $T_1$ and $T_2$ are two spanning trees that have the
    MST property,
  – then they have the same total weight.

• Theorem:
  – In a connected, weighted graph $G = (V, E, W)$
  – a tree $T$ is a minimum spanning tree if and only if
  – $T$ has the MST property.
Correctness of Prim’s MST Algorithm

• Lemma:
  – Let $G = (V, E, W)$ be a connected, weighted graph with $n = |V|$;
  – let $T_k$ be the tree with $k$ vertices constructed by Prim’s algorithm, for $k = 1, \ldots, n$; and
  – let $G_k$ be the subgraph of $G$ induced by the vertices of $T_k$ (i.e., $uv$ is an edge in $G_k$ if it is an edge in $G$ and both $u$ and $v$ are in $T_k$).
  – Then $T_k$ has the MST property in $G_k$.

• Theorem:
  – Prim’s algorithm outputs a minimum spanning tree.
Problem: Single-Source Shortest Paths

- Problem:
  - Finding a minimum-weight path between two specified vertices
  - It turns out that, in the worst case, it is no easier to find a minimum-weight path between a specified pair of nodes s and t than
  - it is to find minimum-weight path between s and every vertex reachable from s. (single-source shortest paths)
Shortest-Path

• **Definition: shortest path**
  – Let P be a nonempty path
  – in a weighted graph $G = (V, E, W)$
  – consisting of k edges $xv_1, v_1v_2, \ldots, v_{k-1}y$ (possibly $v_1 = y$).
  – The *weight* of P, denoted as $W(P)$ is
  – the sum of the weights, $W(xv_1), W(v_1v_2), \ldots W(v_{k-1}y)$.
  – If $x = y$, the empty path is considered to be a path from $x$ to $y$. The weight of the empty path is zero.
  – If no path between $x$ and $y$ has weight less than $W(P)$,
  – then P is called a *shortest path*, or minimum-weight path.
Properties of Shortest Paths

- Lemma: Shortest path property
  - In a weighted graph G,
  - suppose that a shortest path from x to z consist of
  - path P from x to y followed by
  - path Q from y to z.
  - Then P is a shortest path from x to y, and
  - Q is a shortest path form y to z.
Dijkstra’s Shortest-Path Algorithm

- Greedy Algorithm
- weights are nonnegative

dijkstraSSSP(C, n) // OUTLINE
    Initialize all vertices as unseen.
    Start the tree with the specified source vertex s; reclassify it as tree;
    define $d(s, s) = 0$.
    Reclassify all vertices adjacent to s as fringe.
    While there are fringe vertices:
        Select an edge between a tree vertex $t$ and a fringe vertex $v$ such that
        $(d(s, t) + W(tv))$ is minimum;
        Reclassify $v$ as tree; add edge $tv$ to the tree;
        define $d(s, v) = (d(s, t) + W(tv))$.
        Reclassify all unseen vertices adjacent to $v$ as fringe.
The Algorithm in action, e.g.

(a) The graph

(b) An intermediate step

(c) An intermediate step: $CH$ was considered, but not chosen, to replace $GH$ as a candidate.
Correctness of Dijkstra’s Shortest-Path Algorithm

• Theorem:
  – Let $G=(V,E,W)$ be a weighted graph with nonnegative weights.
  – Let $V'$ be a subset of $V$ and
  – let $s$ be a member of $V'$.
  – Assume that $d(s,y)$ is the shortest distance in $G$ from $s$ to $y$, for each $y \in V'$.
  – If edge $yz$ is chosen to minimize $d(s,y)+W(yz)$ over all edges with one vertex $y$ in $V'$ and one vertex $z$ in $V-V'$,
  – then the path consisting of a shortest path from $s$ to $y$ followed by the edge $yz$ is a shortest path from $s$ to $z$.

• Theorem:
  – Given a directed weighted graph $G$ with a nonnegative weights and a source vertex $s$, Dijkstra's algorithm computes the shortest distance from $s$ to each vertex of $G$ that is reachable from $s$. 
Dynamic Programming

• An Algorithm Design Technique
• A Framework to Solve Optimization Problems
• Elements of Dynamic Programming
• Dynamic Programming Version of a Recursive Algorithm
• Developing a Dynamic Programming Algorithm
  - Multiplying a Sequence of Matrices
An Algorithm Design Technique

- Recursive techniques are useful if a problem can be divided into sub-problems with reasonable effort and the sum of the sizes of the sub-problems can be kept small.

- Recall the **Master Theorem**:

  Let $a$, $b$, and $c$ be nonnegative constants. The solution to the recurrence

  $$ T(n) = \begin{cases} 
  b & \text{for } n = 1 \\
  a \ T(n / c) + bn & \text{for } n > 1
  \end{cases} $$

  for $n$ a power of $c$ is

  $$ T(n) = \begin{cases} 
  O(n), & \text{if } a < c \\
  O(n \log n), & \text{if } a = c \\
  O(n^{\log_c a}), & \text{if } a > c
  \end{cases} $$
An Algorithm Design Technique (Cont.)

• If the sum of the sizes of the sub-problems is $an$, for some constant $a > 1$, the recursive algorithm is likely to be polynomial in time complexity.

• If the division of a problem of size $n$ results in $n$ problems of size $n - 1$, then a recursive algorithm is likely to have exponential growth.

• In this case, tabular technique called dynamic programming is more efficient.
A Framework to Solve Optimization Problems

• For each current choice:
  – Determine what sub-problem(s) would remain if this choice were made.
  – Recursively find the optimal costs of those sub-problems.
  – Combine those costs with the cost of the current choice itself to obtain an overall cost for this choice

• Select a current choice that produced the minimum overall cost.
Elements of Dynamic Programming

• Constructing solution to a problem by building it up dynamically from solutions to smaller (or simpler) sub-problems
  – sub-instances are combined to obtain sub-instances of increasing size, until finally arriving at the solution of the original instance.
  – make a choice at each step, but the choice may depend on the solutions to sub-problems

• Principle of optimality
  – the optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.

• Memoization (for overlapping sub-problems)
  – avoid calculating the same thing twice,
  – usually by keeping a table of known results that fills up as sub-instances are solved.
Memoization for Dynamic programming

Version of a Recursive Algorithm e.g.

• Trade space for speed by storing solutions to sub-problems rather than re-computing them.

• As solutions are found for sub-problems, they are recorded in a dictionary, say soln.
  – Before any recursive call, say on sub-problem Q, check the dictionary soln to see if a solution for Q has been stored.
    • If no solution has been stored, go ahead with recursive call.
    • If a solution has been stored for Q, retrieve the stored solution, and do not make the recursive call.
  – Just before returning the solution, store it in the dictionary soln.
Dynamic Programming: An Example

• Problem: The Product of $n$ Matrices

$$M = M_1 \times M_2 \times \ldots \times M_n$$

where each $M_i$ is a matrix with $r_{i-1}$ rows and $r_i$ columns.

• The order in which the matrices are multiplied together can have a significant effect on the total number of operations required to evaluate $M$, no matter what matrix multiplication algorithm is used.

• Example:

Assume that the multiplication of a $p \times q$ matrix by a $q \times r$ matrix requires $pqr$ operations, as in the usual algorithm, and consider the product

$$M = M_1 \times M_2 \times M_3 \times M_4$$

$$\begin{bmatrix}
[10 \times 20] & [20 \times 50] & [50 \times 1] & [1 \times 100]
\end{bmatrix}$$
Dynamic Programming: An Example

- Evaluating $M$ in the order
  \[ M_1 \times (M_2 \times (M_3 \times M_4)) \]
  requires 125,000 operations,
- Evaluating $M$ in the order
  \[ (M_1 \times (M_2 \times M_3)) \times M_4 \]
  requires 2200 operations
- Trying all possible orderings to evaluate the product of $n$ matrices so as to minimize the number of operations is impractical when $n$ is large.
- Dynamic programming provides an $O(n^3)$ algorithm.
Dynamic Programming: An Example

- Let \( m_{ij} \) be the minimum cost of computing \( M_i \times M_{i+1} \times \ldots \times M_j \) for \( 1 \leq i \leq j \leq n \). Clearly,

\[
m_{ij} = \begin{cases} 
0, & \text{if } i = j \\
\min_{i \leq k < j} (m_{ik} + m_{k+1} + r_{i-1} r_k r_j), & \text{if } j > i
\end{cases}
\]

- The term \( m_{ik} \) is the minimum cost of evaluating \( M' = M_i \times M_{i+1} \times \ldots \times M_k \).
- The term \( m_{k+1} \) is the minimum cost of evaluating \( M'' = M_{k+1} \times M_{k+2} \times \ldots \times M_j \).
- The third term is the cost of multiplying \( M' \) by \( M'' \). Note that \( M' \) is a \( r_{i-1} \times r_k \) matrix and \( M'' \) is an \( r_k \times r_j \) matrix.
Dynamic Programming Algorithm

- **Problem**: Compute the minimum cost order of multiplying a string of $n$ matrices $M_1 \times M_2 \times \ldots \times M_n$
- **Input**: $r_0, r_1, \ldots, r_n$, where $r_{i-1}$ and $r_i$ are the dimensions of matrix $M_i$.
- **Output**: The minimum cost of multiplying the $M_i$’s, assuming $pqr$ operations are required to multiply a $p \times q$ matrix by a $q \times r$ matrix.
- **Method**: Recursively define the value of an optimal solution
  - $m[i,j] = 0$ if $i = j$
  - $m[i,j] = \min \{i <= k < j\} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j)$
  - for $1 <= i <= j <= n$
Dynamic Programming Algorithm

MatrixChainOrder(n)
  for (i = 1; i<= n; i++)
    m[i,i] = 0;
  for (s = 2; s<= n; s++)
    for (i = 1; i<= n-s+1; i++)
      { j = i+s-1;
        m[i,j] = inf;
        for (k = i; k< j; k++)
          { q = m[i,k] + m[k+1,j] + Pi-s Pk Pj;
            if q < m[i,j]
              { m[i,j] = q;
                s[i,j] = k
              }
          }
      }
  //At each step, the m[i, j] cost computed depends only on table entries m[i,k] and m[k+1, j] already computed
Construct an Optimal Solution From Computed Information

- **MatrixChainMult(A, s, i, j)**
  - if $j > i$
    - $x = \text{MatrixChainMult}(A, s, i, s[i,j])$
    - $y = \text{MatrixChainMult}(A, s, s[i,j]+1, j)$
    - return $\text{matrixMult}(x, y)$
  - else return $A_i$

- **Analysis:**
  - Time $\Omega(n^3)$ space $\theta(n^2)$
Example on Matrix Multiplication

• Apply the algorithm to the string of 4 matrices:
  \[ M = M_1 \times M_2 \times M_3 \times M_4 \]
  \[
  \begin{bmatrix}
  10 \times 20 \\
  20 \times 50 \\
  50 \times 1 \\
  1 \times 100
  \end{bmatrix}
  \]
  where \( r_0, \ldots, r_4 \) are 10, 20, 50, 1, 100.

• The result of computing the values for the \( m_{ij} \)'s is shown as follows:

<table>
<thead>
<tr>
<th></th>
<th>( m_{11} )</th>
<th>( m_{22} )</th>
<th>( m_{33} )</th>
<th>( m_{44} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_{12} )</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m_{13} )</td>
<td>1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m_{14} )</td>
<td>2200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m_{23} )</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m_{24} )</td>
<td></td>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m_{34} )</td>
<td></td>
<td></td>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>

The minimum number of operations is 2200
Dynamic Programming Version of the fib.

```python
fibDPwrap(n)
    Dict soln = create(n);
    return fibDP(soln, n);

fibDP(soln, k)
    int fib, f1, f2;
    if (k < 2)
        fib = k;
    else
        if (member(soln, k-1) == false)
            f1 = fibDP(soln, k-1);
        else
            f1 = retrieve(soln, k-1);
        if (member(soln, k-2) == false)
            f2 = fibDP(soln, k-2);
        else
            f2 = retrieve(soln, k-2);

    fib = f1 + f2;
    store(soln, k, fib);
    return fib;
```
Development of a Dynamic Programming Algorithm

• Characterize the structure of an optimal solution
  – Breaking a problem into sub-problems
  – whether principle of optimality apply
• Recursively define the value of an optimal solution
  – define the value of an optimal solution based on value of solutions to sub-problems
• Compute the value of an optimal solution in a bottom-up fashion
  – compute in a bottom-up fashion and save the values along the way
  – later steps use the saved values of previous steps
• Construct an optimal solution from computed information