Lecture 1
March 16, 2009

Slides in this course were prepared partly by S. Ar, Y. Moses, A. Tal and T. Tamir
- Thanks to all.

http://www.ee.technion.ac.il/courses/044268
Problem Solving by Computer

Input

↓

manipulation

↓

output
What is an Algorithm?

input ↓

algorithm

output
Algorithm

{input} 2,1,12,7
↓↓↓↓↓↓↓↓↓↓
{data manipulated} 1,2,7,12
↓↓↓↓↓↓↓↓↓↓
{output}
Data
Survivor
Solving a problem

**Data**: information being analyzed.
  e.g.: numbers, words, songs, movies

**Algorithm**: A computational procedure for solving a problem

**Data structure**: The way the data is organized
The Sorting Problem

**Input:** A sequence of \( n \) numbers \( \langle a_1, a_2, ..., a_n \rangle \)

**Output:** A permutation (reordering) \( \langle b_1, b_2, ..., b_n \rangle \) of the input sequence such that \( b_1 \leq b_2 \ldots \leq b_n \)

**Note:** the same elements appear in \( \langle a_1, a_2, ..., a_n \rangle \) and in \( \langle b_1, b_2, ..., b_n \rangle \)

**Example:**

- **Input:** \( \langle 31, 41, 59, 26, 41, 58 \rangle \)
- **Output:** \( \langle 26, 31, 41, 41, 58, 59 \rangle \)
Bubble Sort

- Pass over the sequence
- Compare every pair of adjacent elements
- Switch if they are out of order
- Until there is a pass with no switches
Bubble Sort

31 41 59 26 41 58

26 31 41 59 41 58

26 31 41 41 59 58

26 31 41 41 58 59
Bubble Sort

Bubble-Sort (A)

1. for i ← 1 to length[A] - 1
2. for j ← length[A] downto i + 1
Insertion Sort

Pass over sequence, picking elements 1 by 1

For each element

  insert it into correct place in a new sequence of already sorted elements

Until last element inserted into place
Insertion Sort
### Insertion Sort

<table>
<thead>
<tr>
<th>Sorted</th>
<th>NOT Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5  2  4  6  1  3</td>
</tr>
<tr>
<td>5</td>
<td>2  4  6  1  3</td>
</tr>
<tr>
<td>2  5</td>
<td>4  6  1  3</td>
</tr>
<tr>
<td>2  4  5</td>
<td>6  1  3</td>
</tr>
<tr>
<td>2  4  5  6</td>
<td>1  3</td>
</tr>
<tr>
<td>1  2  4  5  6</td>
<td>3</td>
</tr>
<tr>
<td>1  2  3  4  5  6</td>
<td></td>
</tr>
</tbody>
</table>
Pseudocode

• In the lectures, algorithms will be presented using pseudocode.
  – This is very common in the computer science literature
  – Pseudocode is usually easily translated into real code

• Pseudocode should also be used for algorithms and programs in the homework
How is Data Represented?

**Numbers:** Binary? Ascii? Decimal?

**Sequences:** Arrays? Linked lists?

• There are many options

• Implementing the operations: Swap/Insert etc. depends on the chosen representation
  – Efficiency of the implementation also depends on the choice
Insertion Sort (using Arrays)

Insertion-Sort(A)

1. for i ← 2 to length[A]
2. key ← A[i]
3. j ← i−1
4. while j>0 and A[j]>key
6. j ← j−1
7. A[j+1] ← key
Defining Computational Tasks: Abstract Data Types

**Abstract Data Type**: A mathematical definition of objects, with operations defined on them

**Example**:
- **Objects**: Sets of integers
- **Operations**: Union(S,T)
  - Intersection(S,T)
  - Set_Difference(S,T)
  - member(x,S)
  - make-set(x)
Data Structures vs. Algorithms

**Data Structures**
Represent objects of the ADT

**Algorithms**
Manipulate the data structures to implement the operations of the ADT
Elementary Data Structures

- Arrays
- Lists
- Stacks
- Queues
- Trees

In some languages these are basic data types – in others they need to be implemented.
Examples

• Basic Types
  – integer, real (floating point), Boolean (0,1), character

• Arrays
  – $A[0..99]$ : integer array
    
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
    |---|---|---|---|---|---|---|---|
    | 2 | 1 | 3 | 3 | 2 | 9 | 9 | 6 |
  
    … … … 10

  – $B[0..99]$ : array of images
    
    | 0 | 1 | 2 |
    |---|---|---|
    | ![Image 1](image1.png) | ![Image 2](image2.png) | ![Image 3](image3.png) |
    … … 99
Examples of Lists

- List of numbers
- Representing a sparse, high-order polynomial
- List of names
- List of lists
- List of arrays
Elements of an ADT

\{key, \ldots, \ldots, \ldots, \ldots\}
Typical List Operations

- Search\((S,k)\)
- Minimum\((S)\)
- Maximum\((S)\)
- Successor\((S,x)\)
- Predecessor\((S,x)\)
- Create\((S)\)
- Insert\((S,x)\)
- Delete\((S,x)\)
Linked List as an ADT

\[ L: \]

\[ \text{head} \rightarrow \ldots \rightarrow | | \]

- head\([L]\) - pointer to the head of the list
- next\([x]\) - a pointer to the next element in \( L \)
  or \( \text{NIL} \) if \( x \) is the tail of \( L \)
- Search\((L,k)\) - look for element with key \( k \) in \( L \)
- Insert\((L,x)\) - add \( x \) to \( L \)
- Delete\((L,x)\) - delete \( x \) from \( L \)
Linked List - Implementation

• Using Pointers

• Using Arrays
Searching a Linked List

List-Search(L,k)

\[ x \leftarrow \text{head}[L] \]

\[ \text{while } (x \neq \text{NIL}) \text{ and } (\text{key}[x] \neq k) \]

\[ \text{do } x \leftarrow \text{next}[x] \]

\[ \text{return}(x) \]
Inserting into a Linked List

Inserting a new element at the head of a list:

\[\text{List-Insert}(L,x)\]

\[
\begin{align*}
\text{next}[x] & \leftarrow \text{head}[L] \\
\text{if} & \quad \text{head}[L] \neq \text{NIL} \\
\text{then} & \quad \text{prev}[\text{head}[L]] \leftarrow x \\
\text{head}[L] & \leftarrow x \\
\text{prev}[x] & \leftarrow \text{NIL}
\end{align*}
\]
Deleting from a Linked List

**List-Delete** $(L, x)$

- **if** $\text{prev}[x] \neq \text{NIL}$
  - then $\text{next}[	ext{prev}[x]] \leftarrow \text{next}[x]$
- **else** $\text{head}[L] \leftarrow \text{next}[x]$

- **if** $\text{next}[x] \neq \text{NIL}$
  - then $\text{prev}[	ext{next}[x]] \leftarrow \text{prev}[x]$
More Lists

Doubly Linked List

Circular Linked List
Array as an ADT

**Array:** A mapping from an index set, such as \{0, 1, 2, ..., n\}, into a cell type

- `create(A, n)`
- `value(A, i)`
- `put(A, v, i)` or $A[i] \leftarrow v$
Arrays - "Implementation"

\[ A[i] \text{ - the } i\text{-th element of } A \]

(as a basic data type in the programming language)
Arrays - Implementation

\[\text{access}(A, i) = \{ \text{put}(A, i), \text{get}(A, i) \}\]

(arrays implemented by linked lists)