CS 2305: Discrete Mathematics for Computing I

Lecture 18

- KP Bhat

Some Example Algorithm Problems

Three classes of problems will be studied in this chapter.

1. *Searching Problems*: finding the position of a particular element in a list.
2. *Sorting problems*: putting the elements of a list into increasing order.
3. *Optimization Problems*: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.
    - An algorithm that quickly produces good, but not necessarily optimal solutions is called a *heuristic*. You will study them in depth in more advanced CS courses.

Searching Problems

Definition: The general *searching problem* is to locate an element x in the list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list.

- The solution to a searching problem is the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) or 0 if x is not in the list.
- For example, a library might want to check to see if a patron is on a list of those with overdue books before allowing him/her to checkout another book.
- We will study two different searching algorithms: linear search and binary search.

Linear Search Algorithm

The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning.

- First compare x with a_1. If they are equal, return the position 1.
- If not, try a_2. If $x = a_2$, return the position 2.
- Keep going, and if no match is found when the entire list is scanned, return 0.

procedure *linear search*(x:integer,
$a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
  $i := i + 1$
if $i \leq n$ **then** *location* := i
else *location* := 0
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

Binary Search 1

Assume the input is a list of items in increasing order.

The algorithm begins by comparing the element to be found with the middle element.

- If the middle element is lower, the search proceeds with the upper half of the list.
- If it is not lower, the search proceeds with the lower half of the list (through the middle position).

Repeat this process until we have a list of size 1.

- If the element we are looking for is equal to the element in the list, the position is returned.
- Otherwise, 0 is returned to indicate that the element was not found.

Binary search algorithm is much more efficient than linear search, but it requires the items to be in sorted order.

Binary Search 2

Here is a description of the binary search algorithm in pseudocode.

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
  $i := 1$ {i is the left endpoint of interval}
  $j := n$ {j is right endpoint of interval}
  while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ **then** $i := m + 1$
    else $j := m$
  if $x = a_i$ **then** *location* $:= i$
  else *location* $:= 0$
  return *location*{location is the subscript i of the term a_i equal to x, or 0 if x is not found}

Binary Search 3

- Starting with the entire range of values, the algorithm successively narrows down the part of the sequence being searched
- At any given stage only the terms from a_i to a_j are under consideration
- The algorithm continues narrowing the part of the sequence being searched until only one term of the sequence remains. When this is done, a comparison is made to see whether this term equals the term we are looking for

Binary Search 4

Example: The steps taken by a binary search for 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. The list has 16 elements, so the midpoint is 8. The value in the 8th position is 10. Since 19 > 10, further search is restricted to positions 9 through 16.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

2. The midpoint of the list (positions 9 through 16) is now the 12th position with a value of 16. Since 19 > 16, further search is restricted to the 13th position and above.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

3. The midpoint of the current list is now the 14th position with a value of 19. Since 19 $\not>$ 19, further search is restricted to the portion from the 13th through the 14th positions .

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

4. The midpoint of the current list is now the 13th position with a value of 18. Since 19> 18, search is restricted to the portion from the 14th position through the 14th.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

5. Now the list has a single element and the loop ends. Since 19=19, the location 14 is returned.

Sorting

To *sort* the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on).

Sorting is an important problem because:

- A nontrivial percentage of all computing resources are devoted to sorting different kinds of lists, especially applications involving large databases of information that need to be presented in a particular order (e.g., by customer, part number etc.).
- An amazing number of fundamentally different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively.
- Sorting algorithms are useful to illustrate the basic notions of computer science.

More than 100 sorting algorithms have been developed till date: this text discusses the following:

- binary, insertion, bubble, selection, merge, quick, and tournament

Bubble Sort 1

Bubble sort makes multiple passes through a list. Every pair of elements that are found to be out of order are interchanged.

- With each iteration the highest remaining number "sinks" to its correct position to the right

https://www.youtube.com/watch?v=xli_FI7CuzA

procedure *bubblesort*(a_1,…,a_n: real numbers with $n \geq 2$)

  for i := 1 **to** $n-1$

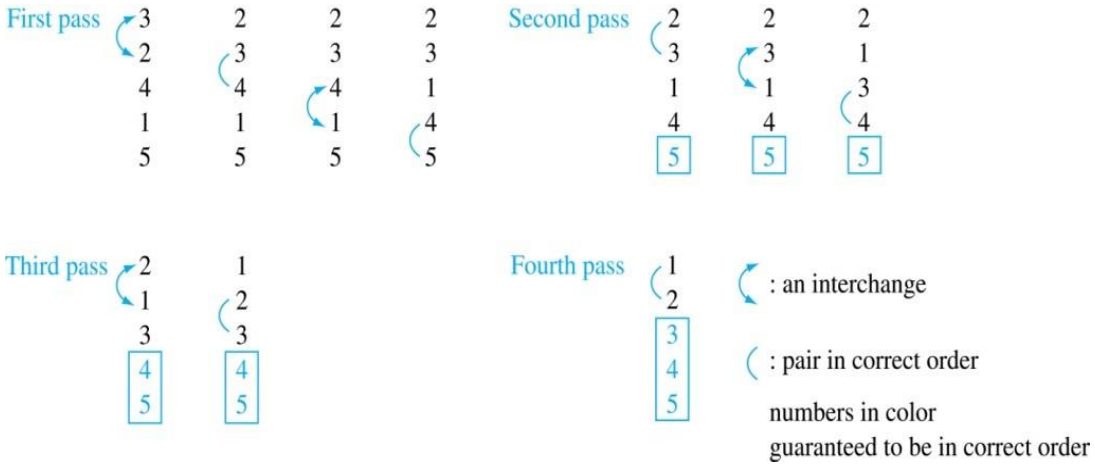
    for j := 1 **to** $n - i$

      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

{a_1,…, a_n is now in increasing order}

Bubble Sort 2

Example: Show the steps of bubble sort with 3 2 4 1 5



At the first pass the largest element has been put into the correct position

At the end of the second pass, the 2nd largest element has been put into the correct position.

In each subsequent pass, an additional element is put in the correct position.

Bubble Sort 3

- Although bubble sort is considered to be an extremely inefficient algorithm, it is possible to slightly tweak the algorithm so as to give bubble sort the ability to handle already sorted lists very efficiently

Bubble Sort 4

```
procedure bubblesort(a1,…,an: real numbers with n ≥ 2)
   swapped := true
   for i := 1 to n− 1
      if(not swapped)
      then
         break
      endif
      swapped := false
      for j := 1 to n − i
         if aj >aj+1
         then
            interchange aj and aj+1
            swapped := true
         endif
      endfor
   endfor
{a1,…, an is now in increasing order}
```

Selection Sort 1

- The selection sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining elements is found and put into the second position. This procedure is repeated until the entire list has been sorted
  - Section 3.1, Exercise 42
  - https://www.youtube.com/watch?v=g-PGLbMth_g
- In general more efficient than bubble sort (lot fewer swaps) unless the list is already sorted

Selection Sort 2

```
procedure selectionsort(a1, a2, …., an)
   for i := 1 to n - 1
      minspot := i
      for j := i + 1 to n
         if aj < aminspot then minspot := j
      interchange aminspot and ai
{the list is now in order}
```

Insertion Sort 1

- The sorting method used by most people for sorting playing cards in their hands
- *Insertion sort* begins with the 2nd element. It compares the 2nd element with the 1st and puts it before the first if it is not larger.
- Next the 3rd element is put into the correct position among the first 3 elements.
- In general, in the j^{th} step of the insertion sort, the j^{th} element of the list is inserted into the correct position in the list of the previously sorted $j - 1$ elements.
- The array forms sorted and unsorted partitions
  - With each iteration the size of the sorted partition increases by 1, while the size of the unsorted partition decreases by 1
- You keep swapping the first element of the unsorted partition with the last elements of the sorted partition until the element is in the correct position in the sorted partition
- https://www.youtube.com/watch?v=JU767SDMDvA

Insertion Sort 2

- An optimized version of the algorithm uses fewer swaps but instead uses a temporary location to store the element that needs to be assigned the correct location
  - Find out the position in the sorted partition where the element under consideration will go and move all subsequent elements in the sorted partition one position to the right, starting from the last element of the sorted partition, thereby creating a “hole” in the sorted partition, where the where the element will go
  - https://www.youtube.com/watch?v=pmDnM9gUxNc&t=25s

Insertion Sort 3

```
for i : 1 to length(A) -1
   j = i
   while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j = j - 1
```

procedure *insertion sort*
(a_1,…,a_n:
real numbers with $n \geq 2$)
for $j := 2$ **to** n
$i := 1$
while $a_j > a_i$
$i := i + 1$
$m := a_j$
for $k := 0$ **to** $j - i - 1$
$a_{j-k} := a_{j-k-1}$
$a_i := m$
{Now a_1,…,a_n is in increasing order}

Insertion Sort 4

Example: Show all the steps of insertion sort with the input:

3 2 4 1 5

i. 2 3 4 1 5 (*first two positions are interchanged*)

ii. 2 3 4 1 5 (*third element remains in its position*)

iii. 1 2 3 4 5 (*fourth is placed at beginning*)

iv. 1 2 3 4 5 (*fifth element remains in its position*)