ITM 312, Fall 2013

Chapter 8 Lecture Notes

8.1. Introduction to Search Algorithms

a. Search - locate an item in a list of information

Two algorithms we will examine:

b. Linear search – best case: $O(1)$; worst case: $O(n)$; average: $O(\frac{n+1}{2})$

i. Also called the sequential search

ii. Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.

iii. Example:

1. Array `numlist` contains: 17, 23, 5, 11, 2, 29, 3
2. Searching for the value 11, linear search examines 17, 23, 5, and 11
3. Searching for the value 7, linear search examines 17, 23, 5, 11, 2, 29, and 3

iv. Pseudocode:

```
set found to false; set position to -1; set index to 0
while index < number of elts. and found is false
    if list[index] is equal to search value
        found = true
        position = index
        end if
    add 1 to index
end while
return position
```

v. Implementation:

```
int searchList(int list[], int numElems, int value)
{
   int index = 0;       // Used as a subscript to search array
   int position = -1;   // To record position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < numElems && !found)
   {
      if (list[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
return position; // Return the position, or -1
}
```

vi. Flags like `isFound` used to indicate if item was found
    1. No point in continuing to loop when item is already found

vii. Advantages:
    1. Easy algorithm to understand
    2. Array can be in any order

viii. Disadvantages:
    1. Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

c. Binary search – best case: $O(1)$; worst case: $O(\log n)$; average: $O(\log n)$

i. Requires array elements to be in order
    1. No need for comparison of elements (already sorted)

ii. Steps:
    1. Divides the array into three sections:
        a. middle element
        b. elements on one side of the middle element
        c. elements on the other side of the middle element
    2. If the middle element is the correct value, done. Otherwise, go to step 1 using only the half of the array that may contain the correct value.
    3. Continue steps 1 and 2 until either the value is found or there are no more elements to examine

iii. Example

1. Array `numlist2` contains: 2, 3, 5, 11, 17, 23, 29
2. Searching for the value 11, binary search examines 11 and stops (it starts from the middle)
3. Searching for the value 7, linear search examines 11, 3, 5, and stops

iv. Pseudocode:

```
Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal
to last
    Set middle to the subscript half-way between
array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired
value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.
```

v. Implementation:

```
int binarySearch(int array[], int size, int value)
{
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```


vi. Advantages:

1. Much more efficient than linear search. For array of N elements, performs at most $\log_2(n)$ comparisons:
   a. $\log_2(n)$ is the number of times you can divide x in half before you get down to 1 (or less).
   b. For example: If you search for a name in the phone book by peeking in the middle of the book to see if the name is in the first half or the second half, then repeating that operations until you find the right page, $\log_2(x)$ is roughly the number of pages you'll visit
      i. $\log_2(8)$ equals 3, since 2^3 equals 8.
      ii. $\log_{10}(100)$ equals 2, since 10^2 equals 100

vii. Disadvantages:

1. Requires that array elements be sorted

8.2. Introduction to Sorting Algorithms

a. Sort - arrange values into an order:
   i. Alphabetical
   ii. Ascending numeric
   iii. Descending numeric

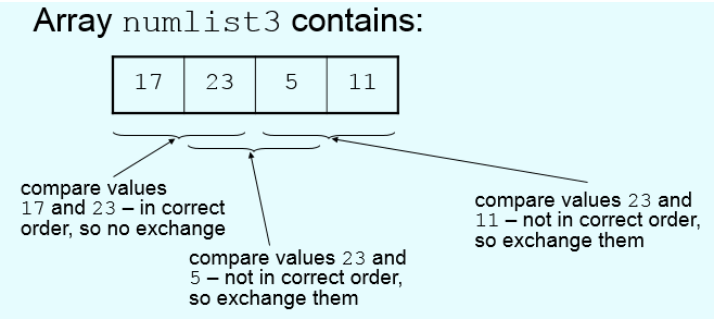
b. Bubble Sort – best case: $O(n)$; worst case: $O(n^2)$; average: $O(n^2)$

i. Steps:

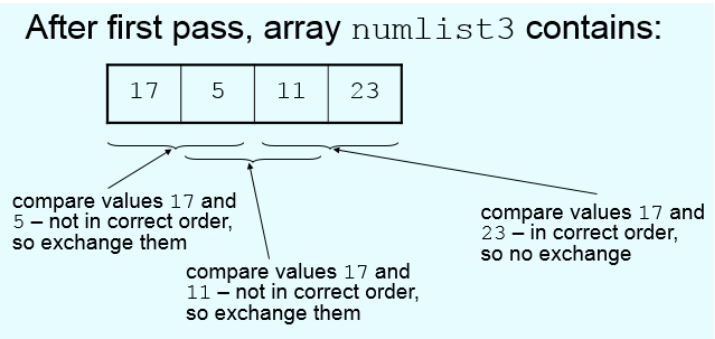
1. Compare 1st two elements
    a. If out of order, exchange them to put in order
2. Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array.
3. Pass through array again, exchanging as necessary
4. Repeat until pass made with no exchanges

ii. Example:

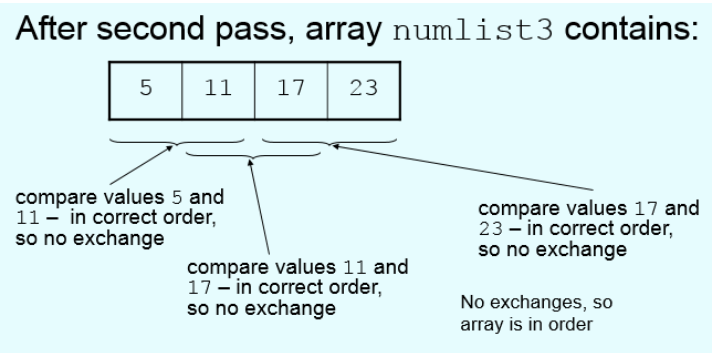
1. First pass:



2. Second pass:



3. Third (last) pass:



iii. Implementation:

```
34 void sortArray(int array[], int size)
35 {
36    bool swap;
37    int temp;
38
39    do
40    {
41       swap = false;
42       for (int count = 0; count < (size - 1); count++)
43       {
44          if (array[count] > array[count + 1])
45          {
46             temp = array[count];
47             array[count] = array[count + 1];
48             array[count + 1] = temp;
49             swap = true;
50          }
51       }
52    } while (swap);
53 }
```

iv. Advantages:

1. Easy to understand and implement

v. Disadvantages:

1. Inefficient (slow for large arrays)

c. Selection Sort – best case: $O(n^2)$; worst case: $O(n^2)$; average: $O(n^2)$

i. Steps:

1. Locate smallest element in array. Exchange it with element in position 0
2. Locate next smallest element in array. Exchange it with element in position 1.
3. Continue until all elements are arranged in order

ii. Example: Array `numlist` contains `{11, 2, 29, 3}`

1. Smallest element is 2. Exchange 2 with element in 1st position in array: `{2, 11, 29, 3}`
2. Next smallest element is 3. Exchange 3 with element in 2nd position in array: `{2, 3, 29, 11}`
3. Next smallest element is 11. Exchange 11 with element in 3rd position in array: `{2, 3, 11, 29} // in order!`

iii. Implementation

```
35 void selectionSort(int array[], int size)
36 {
37    int startScan, minIndex, minValue;
38
39    for (startScan = 0; startScan < (size - 1); startScan++)
40    {
41       minIndex = startScan;
42       minValue = array[startScan];
43       for(int index = startScan + 1; index < size; index++)
44       {
45          if (array[index] < minValue)
46          {
47             minValue = array[index];
48             minIndex = index;
49          }
50       }
51       array[minIndex] = array[startScan];
52       array[startScan] = minValue;
53    }
54 }
```

iv. Advantages:

1. More efficient than Bubble Sort, since fewer exchanges

v. Disadvantages:

1. May not be as easy as Bubble Sort to understand