

Sorting and Searching

I. Searches

A) Sequential (linear) Search - An algorithm that starts at the beginning of the list and looks at each item in sequence.

ex.1) Sequential search function:

```
public int search(int[ ] arrayEx, int target)           // function heading
{
    int index = 0,
    boolean found = false;
    while ((!found) && (index < arrayEx.length))       // sequential search
    {
        if (target == arrayEx[index])
            found = true;
        else
            index++;
    }
    if (found)
        return index;
    else
        return -1;
}
/* returns the array index if found
   otherwise a negative is returned
   to indicate no match */
```

I. Searches continued

B) Binary Search - This search reduces the range to be searched by approximately one-half after each comparison by searching the middle of an ordered array or its subsequent subarray.

Note: A billion item array could be searched in a maximum of 30 comparisons with this method

ex.1) Binary search function:

```
// Function accepts an array, a range of elements for the search ,
// and the number for which we are searching.
public void binarySearch(int[ ] arrayEx, int lowerbound, int upperbound,
    int target)
{
    int index;
    int compare_count = 1;           // Variable used to count the comparisons
    index = (lowerbound + upperbound)/2; // Calculate initial search position
    while((arrayEx[index] != target) && (lowerbound <= upperbound))
    {
        compare_count++;
        if(arrayEx[index] > target)    // If the value in the search
        {                               // position is greater than the number
            upperbound = index - 1;    // for which we are searching, change
        }                               // upperbound to the search position
        else                            // minus one.
        {                               // else, change lowerbound to search
            lowerbound = index + 1;    // position plus one.
        }
        index = (lowerbound + upperbound) / 2;
    }
    if(lowerbound <= upperbound)
    {
        System.out.println("A binary search found the number in "
            + compare_count + " comparisons.\n");
    }
    else
    {
        System.out.println("Number not found by binary search after "
            + compare_count + " comparisons.\n");
    }
}
```

I. B) Binary Search continued

Given the declaration: `int[] list = {10, 20, 30, 40, 50, 60};`
a binary search is used to search the list for a particular value. In each iteration of the search loop, the index variables first, middle, and last define the range of items being searched. When the search is finished, what is the value of first given the value below that is being searched for?
(Note: The answer should be either 0, 1, 2, 3, 4, or 5.)
(Note: Remember that Java arrays begin at index 0.)

ex.1) Search for the value 20

ex.2) Search for the value 55

ex.3) Search for the value 50

AP Programming - Chapter 18 Lecture

I. B) Binary Search continued (KEY)

index: 0 1 2 3 4 5

Given the declaration: `int[] list = {10, 20, 30, 40, 50, 60};`

ex.1) Search for the value 20

| <u>Target</u> | <u>first lowerbound</u> | <u>last upperbound</u> | <u>middle index</u> | <u>lower + upper</u> |
|---------------|-----------------------------|----------------------------|-------------------------|----------------------|
| 20 | 0 | 5 | $2.5 = 2$ | (30) |
| | 0 | 1 | $0.5 = 0$ | (10) |
| | 1 | 1 | 1 | (20) |

index: 0 1 2 3 4 5

Given the declaration: `int[] list = {10, 20, 30, 40, 50, 60};`

ex.2) Search for the value 55

| <u>Target</u> | <u>first lowerbound</u> | <u>last upperbound</u> | <u>middle index</u> | <u>lower + upper</u> |
|---------------|-----------------------------|----------------------------|-------------------------|----------------------|
| 55 | 0 | 5 | $2.5 = 2$ | (30) |
| | 3 | 5 | 4 | (50) |
| | 5 | 5 | 5 | (60) |
| | 5 | 4 | Stop Searching | |

index: 0 1 2 3 4 5

Given the declaration: `int[] list = {10, 20, 30, 40, 50, 60};`

ex.3) Search for the value 50

| <u>Target</u> | <u>first lowerbound</u> | <u>last upperbound</u> | <u>middle index</u> | <u>lower + upper</u> |
|---------------|-----------------------------|----------------------------|-------------------------|----------------------|
| 50 | 0 | 5 | $2.5 = 2$ | (30) |
| | 3 | 5 | 4 | (50) |

II. Sorts - arranging the components of a list into order.

There are two main types of sorting algorithms:

incremental and *divide-and-conquer* (which we will not study until a later chapter)

A) Incremental - repeatedly passing through the list in order looking for or switching (swapping elements in the list.)

1) Selection Sort - If sorting in ascending order (from smallest to largest) then method 1: the smallest item is moved to the beginning of the list and the process continues with the next smallest item moving to the second position in the list etc.

or

method 2: the largest item is moved to the end of the list and the process continues with the next largest item moving to the next to last position in the list etc.

II. Sorts - Selection sort continued

ex.1) ascending order selection sort (method 1: using minimum value)

```
public class SelectionExample
{
    public static void main(String[ ] args)
    {
        int[ ] nums= {22, 5, 67, 98, 45};
        int index, j, temp, moves, min, minIndex;
        moves = 0;
        for (index = 0; index < 5; index++)
        {
            min = nums[index];
            minIndex = index;
            for (j = index + 1; j < 5; j++)
            if (nums[j] < min)
            {
                min = nums[j];
                minindex = j;
            }
            // perform the switch
            if (min < nums[index])
            {
                temp = nums[index];
                nums[index] = min;
                nums[minIndex] = temp;
                moves++;
            }
        }
        System.out.println("\nThe sorted list, in ascending order, is:");
        for (index = 0; index < 10; ++index)
            System.out.print(nums[index] + " ");
        System.out.println("\n' + moves + " moves were made to sort this list.");
    }
}
```

original order: {22, 5, 67, 98, 45}
first pass: {5, 22, 67, 98, 45}
second pass: {5, 22, 67, 98, 45}
third pass: {5, 22, 45, 98, 67}
fourth pass: {5, 22, 45, 67, 98}

II. Sorts - Incremental continued

- 2) Insertion Sort - Start with a list and make a sorted sublist from it by pulling elements out of the unsorted list and putting them into the sorted list.

ex.1) ascending order insertion sort

```
public class InsertionExample
{
    public static void main(String[ ] args)
    {
        int[ ] nums= {22, 5, 67, 98, 45};
        int nums_length = 5;
        System.out.println("Unsorted Array:");
        display_array(nums, nums_length);
        insertion_sort(nums, nums_length);
        System.out.println("Sorted Array:");
        display_array(nums, nums_length);
    }
    public static void insertion_sort(int[ ] array, int array_length)
    {
        int j, index, key;
        for (j = 1; j < array_length; j++)
        {
            key = array[j];
            for (index = j - 1; (index >= 0) && (array[index] > key); index--)
            {
                array[index + 1] = array[index];
            }
            array[index + 1] = key;    // insert key into proper position
        }
    }
    public static void display_array(int[ ] input_array, int input_size)
    {
        int index;
        for (index = 0; index < input_size; index++)
        {
            System.out.println(input_array[index] + ' ');
        }
        System.out.println("\n\n");
    }
}
```

| | | |
|-------------------------------|---------------------|-------------------------------------|
| original unsorted order: | {22, 5, 67, 98, 45} | sorted sublist: { } |
| first pass unsorted sublist: | {67, 98, 45} | sorted sublist: {5, 22} |
| second pass unsorted sublist: | {98, 45} | sorted sublist: {5, 22, 67} |
| third pass unsorted sublist: | {45} | sorted sublist: {5, 22, 67, 98} |
| fourth pass unsorted sublist: | { } | sorted sublist: {5, 22, 45, 67, 98} |

II. Sorts - Recursive (recursion is discussed in the next chapter)

3) Merge Sort - divides the list into smaller lists by $\frac{1}{2}$ and sorts the smaller lists

Here is a recursive description of how mergesort works:

If there is more than one element in the array Break the array into two halves.
Mergesort the left half. Mergesort the right half. Merge the two subarrays into a sorted array.

Mergesort uses a merge method to merge two sorted pieces of an array into a single sorted array. For example, suppose array $a[0] \dots a[n - 1]$ is such that $a[0] \dots a[k]$ is sorted and $a[k + 1] \dots a[n - 1]$ is sorted, both parts in increasing order.

ex.1)

| | | | | | |
|------|------|------|------|------|------|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |
| 2 | 5 | 8 | 9 | 1 | 6 |

In this case, $a[0] \dots a[3]$ and $a[4] \dots a[5]$ are the two sorted pieces. The method call `merge(a, 0, 3, 5)` should produce the "merged" array:

| | | | | | |
|------|------|------|------|------|------|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |
| 1 | 2 | 5 | 6 | 8 | 9 |

The middle numerical parameter in `merge` (the 3 in this case) represents the index of the last element in the first "piece" of the array. The first and third numerical parameters are the lowest and highest index, respectively, of array `a`.

Here's what happens in mergesort:

1. Start with an unsorted list of n elements.
2. The recursive calls break the list into n sublists, each of length 1.
Note that these n arrays, each containing just one element, are sorted!
3. Recursively merge adjacent pairs of lists. There are then approximately $n/2$ lists of length 2; then, approximately $n/4$ lists of approximate length 4, and so on, until there is just one list of length n .

AP Programming - Chapter 18 Lecture

ex.2) An example of mergesort follows:

| | | | | | |
|----|----|---|---|---|---|
| 5 | -3 | 2 | 4 | 0 | 6 |
| 5 | -3 | 2 | 4 | 0 | 6 |
| 5 | -3 | 2 | 4 | 0 | 6 |
| 5 | -3 | 2 | 4 | 0 | 6 |
| -3 | 5 | 2 | 0 | 4 | 6 |
| -3 | 2 | 5 | 0 | 4 | 6 |
| -3 | 0 | 2 | 4 | 5 | 6 |

Break list **into** n sublists of length 1
then merge into sorted sublists

Note: A merge sort is always more efficient than the selection and in most case also the insertion sort.

Note: The major disadvantage of a merge sort is that it needs a temporary array that is as large as the original array to be sorted. This could be a problem if space is a factor and you are dealing with extremely large arrays.

Chapter 18 Terminology:

sequential search - An algorithm that starts at the beginning of the list and looks at each item in sequence.

binary search - This search reduces the range to be searched by approximately one-half after each comparison.

sorting - Arranging the components of a list into order.

selection (straight selection) sort- This sort makes several passes through the list, each time swapping a component into its proper position.

insertion sort - This sort starts with an empty list and places each new item into its proper position in the list.

merge sort - divides the list into smaller lists by $\frac{1}{2}$ and sorts the smaller lists