**Searches and Sorts**

**Part I. Sorts**

**Bubble Sort**

[Bubble Sort](http://quiz.geeksforgeeks.org/bubble-sort/)is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

   int i, j;

   for (i = 0; i < n-1; i++)

       // Last i elements are already in place

       for (j = 0; j < n-i-1; j++)

           if (arr[j] > arr[j+1])

              swap(&arr[j], &arr[j+1]);

**Selection Sort**

1. Assume you have an array *arr* with *n* elements.
2. Find the smallest element in an array *arr*, and swap it with index *arr[0*], the first element.
3. Now find the smallest element in the array from *arr[1]* to *arr[n-1]* and swap it with *arr[1].*
4. Continue this process until just two elements remain to be sorted, arr[n-2] and arr[n-1]. The smaller is placed at *arr[n-2]* and the larger *at arr[n-1]* and the sort is complete.
5. For an array of n elements, the array is sorted after n-1 passes.
6. After the kth pass, the first k elements are in their final sorted position.

**public** **static** **void** selectionSort(**int**[] arr)

{

**for** (**int** i = 0; i < arr.length - 1; i++)

{

**int** index = i;

**for** (**int** j = i + 1; j < arr.length; j++)

{

**if** (arr[j] < arr[index]) index = j;//finds the index of the lowest value //from i+1 to end of array

}

//swaps arr[i] and arr[index] - if already sorted, nothing changed

**int** smallerNumber = arr[index];

arr[index] = arr[i];

arr[i] = smallerNumber;

}

}

**Insertion Sort**

1. Think of the first element in an array as being sorted. You can then think of the array as consisting of two parts, the sorted part and the unsorted part.
2. The idea of an insertion sort is to move elements from the "unsorted" part to the "sorted" part, one at a time and each item is placed in its appropriate location in the sorted part of the array.
3. In order to place a new element into the sorted part of the list, some elements may need to be moved down the array to create a slot.
4. Example
   1. 8146 - We assume 8 is sorted at index 0 to begin with
   2. 1846 - The first pass of the insertion sort looks at index 1 and sorts it with index 0;
   3. 1486 - The second pass sorts index 2 with indices 0 and 1.
   4. 1468 - The third and final pass sorts index 3 with indices 0 to 2 and ends the insertion sort.
5. For an array of n elements, the array is sorted after n-1 passes.
6. After the kth pass, a[0] through a[k] are sorted with respect to one another, but not necessarily in their final sorted positions.
7. Worst case occurs if the array is initially in reverse order, which leads to the maximum comparisons and moves.
8. Best case is if the array is already sorted. Each pass will involve only one comparison and no movement.

**public** **static** **void** insertionSort(**int** array[])

{

**int** n = array.length;

**for** (**int** j = 1; j < n; j++) //assume first element is sorted and rest are unsorted

{

**int** key = array[j]; //value from unsorted to be positioned in the sorted part

**int** i = j-1;

**while** ( (i > -1) && ( array [i] > key ) ) //start at last index of sorted part

{

array [i+1] = array [i];

i--;

}

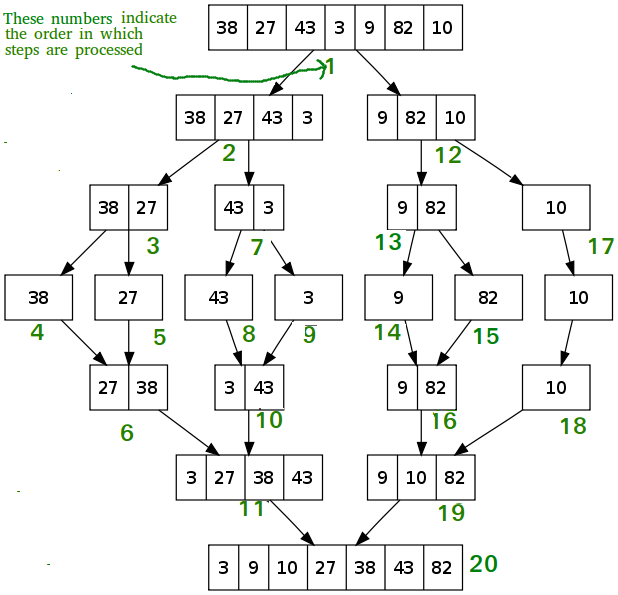
array[i+1] = key;

}

}

**Mergesort**

1. Mergesort is a *recursive* sort and is more efficient than both selection and insertion sorts.
2. Mergesort uses a “divide and conquer” approach.
3. If there is more than one element in an array:
   1. Break the array into two halves
   2. Mergesort the left half
   3. Mergesort the right half
   4. Continue until the broken arrays are size 1
   5. Merge the two subarrays into a sorted array.
4. The major disadvantage of Mergesort is that it needs a temporary array that is as large as the original array to be sorted.
5. Mergesort is not affected by the initial ordering of the elements. Efficiency is practically independent of the order of the values in the array.



**Part II. Searches**

**Sequential Search**

1. Assume you are searching for a key in a list of n elements. A sequential search starts at the first element and compares the key to each element in turn until the key is found or there are no more elements to examine in the list.
2. The best case has the key in the first slot.
3. The worst case occurs if the key is in the last slot or not in the list. All n elements are examined in this situation.
4. On average there will be n/2 comparisons. (n being the number of elements in the array)

**public** **static** **int** sequentialSearch(**int** [] list, **int** key)

{

**for**(**int** x =0; x < list.length; x++)

{

**if**(list[x]==key) **return** x;

}

**return** -1;

}

**Binary Search**

1. Binary search only works with a sorted array and uses a divide and conquer approach.
2. Process: Assume a is an array of integers and key is the value we are searching for within the array. We want to find the index where key is found in the array or have -1 returned if the value isn’t found in the array.
   1. Assume a[low] … a[high] is sorted in ascending order and a method binarySearch returns the index of value we are searching for, which is usually called the key.
   2. If key is not in the array, it returns -1.
3. In the best case, the key is found on the first try. (low + high /2 ) is the key.
4. Worst case, the key is not in the list or it is at either end of the list. Number of recursive calls requires us to take the number of elements in the array and round it up to the next power of 2 and take the exponent. Suppose n =9. 9 rounds up 16 which is 24. Thus you would need 4 comparisons to find the key in a worst case scenario.
5. BinarySearch Logic

**public** **static** **int** binarySearch(**int**[] list, **int** key)

{

**int** low = 0;

**int** high = list.length - 1;

**while** (high >= low)

{

**int** mid = (low + high) / 2;

**if** (key < list[mid]) high = mid - 1;

**else** **if** (key == list[mid]) **return** mid;

**else** low = mid + 1;

}

**return** -1;

}

**Binary Search Example**

int [ ] nums = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79}; (13 elements in nums array)

key = 45 (This is the value whose index we are searching for within the nums array)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Loop #** | **low** | **high** | **mid** | **nums[mid]** | **key is ? than nums[mid]** |
| 1 | 0 | 12 | 6 | 50 | smaller |
| 2 | 0 | 5 | 2 | 7 | larger |
| 3 | 3 | 5 | 4 | 11 | larger |
| 4 | 4 | 5 | 4 | 11 | larger |
| 5 | 5 | 5 | 5 | 45 | found |

**Answer**: The key, 45, is found at index 5 within the nums array.

int [] nums = {65, 103, 192, 312, 351, 363, 369, 435, 452, 753, 813, 837, 850, 857, 860, 876, 900, 939}; (18 elements)

key = 360

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Loop #** | **low** | **high** | **mid** | **nums[mid]** | **key is ? than nums[mid]** |
| 1 | 0 | 17 | 8 | 452 | smaller |
| 2 | 0 | 7 | 3 | 312 | larger |
| 3 | 4 | 7 | 5 | 363 | smaller |
| 4 | 4 | 6 | 5 | 363 | smaller |
| 5 | 4 | 5 | 4 | 351 | larger |
| 6 | 5 | 5 | 5 | 363 | smaller |
| 7 | 5 | 4 | return -1 as *low* is greater than *high* (key is unfound) | | |