Tutorial: Fundamental Search & Sorting Algorithms

This tutorial assumes prior exposure to the fundamentals of computer programming (in either a structured or object oriented language, like C, C++, Java…), as well as an understanding of computational complexity and data structure (covered in a separate tutorial).

1 Introduction

Suppose you were working on an assembly line. A conveyor belt would bring down a constant flow of plaques with people's names. As these name plaques came in your job was to place them in the first available slot on the shelf. After a while, your boss asks you to retrieve plaques with certain people's names. *What do you?*

Well one approach would is to go through every single one until you find the names requested. But think about that for a moment… If there are a 100 names, it is conceivable that every time a new name is requested, you might have to search through a 100 names.

That sounds annoying, but maybe not too bad. What if it was a million or billions of names? That would be awful, right?

Now say these requests from your boss were quite frequent. You could continue to brutally search through the shelf sequentially…. Or you could realize that you could order them (say alphabetically), making the search much faster.

This process is analogous to receiving a (possibly disorganized) array of data with some underlying order, and then trying to either brute force search it, or conversely organize it for efficiency.

Two significant considerations in the algorithms to be presented are the computational complexity and memory complexity. Computational complexity is equivalent to how many items you would have to manipulate to either organize them, or to find the appropriate item. The memory complexity is analogous to the amount of extra (shelf) space needed to organize everything. In the case of computational complexity, it is possible that the array is already sorted, it is also possible it is organized in fashion that is completely inconvenient (using our example, the item we would be searching for always happens to be the last item). For this reason, we offer a worst case, best case and mean/average computational complexity.

2 Sorting Algorithms

Before we proceed, we will assume all data comes in an array $Arr(i)\ or\ A()$, where i is the index (starting at 0) and it is of size N (that is, that there are N number of entries to be sorted). Below is the list of algorithms we will look at in detail.

Algorithm	**Best Case**	**Mean**	**Worst Case**	**Memory**
Insertion Sort	n	n^2	n^2	1
Selection Sort	n^2	n^2	n^2	1
Merge Sort	$n\log n$	$n\log n$	$n\log n$	n
Quicksort	$n\log n$	$n\log n$	n^2	$\log n$ (or n)
Bubble Sort	n	n^2	n^2	1

2.1 Selection Sort

Concept

The general idea behind Selection Sort is:

- Find the smallest value/lowest ordered in the Arr (the array) and put it in Arr[0]
- Find the next smallest value/lowest ordered and put it in Arr[1]
- Repeat the process until all element are ordered

In general, we state it as follows:

- Use an outer loop from $i = 0\ to\ i = N - 1$
- Each time around, use a nested loop (from $j = i + 1\ to\ j = N - 1$) to find the next ordered item
- Swap the order item with the value currently occupying Arr[i]

Source Code

```
public static void selectionSort(Comparable[] A) {
  int j, k, minIndex;
  Comparable min;
  int N = A.length;

  for (k = 0; k < N; k++) {
     min = A[k];
     minIndex = k;
     for (j = k+1; j < N; j++) {
        if (A[j].compareTo(min) < 0) {
      min = A[j];
      minIndex = j;
  }
     }
     A[minIndex] = A[k];
     A[k] = min;
  }
}
```

Complexity

Now we need to consider the complexity. Well the outer loop executes N-1 times. The inner loop executes a different number of times each time around the outer loop. However, we can observe that the 1st iteration of the outer loop executes N-1 times, and the 2nd N-2 times, and so on, until the last index (Nth) runs 0 times.

This entire process is a sum $N - 1 + N - 2 + N - 3 + \cdots + 3 + 2 + 1 + 0 = \sum_{i=1}^{N} N - i$, which is $O(N^2)$. This is not particularly efficient, and worse, this worst case scenario occurs even if the array is already sorted.

Example of Selection Sort

We are given an array Arr = {64, 25, 12, 22, 11}, using selection sort, we wish to order the array (increasing).

For index = 0, we set our current min as 64 (min = Arr{0}). We then proceed to look at index 1 through 4. We notice index 1 (Arr(1) ={25}) is less than 64. So we swap and set our new min as 25 {25, 64, 12, 22, 11}. We proceed to index 2, and notice 12 is less than 25, so we swap again {12, 64, 25, 22, 11}. Index 3 is 22, which is not less, so we do nothing. We then proceed to index 4, which is 11 and less than 12… So we swap, and get {11, 64, 25, 12, 22}

We then proceed to index 1. This time we are evaluating index 2 through 4. We repeat the process and get {11, 12, 64, 25, 22} after this iteration.

The next iteration starts at index 2. This time we are evaluating index 3 and 4, and we get {11, 12, 22, 64, 25}.

Finally, we reach index 3, and we only have to compare index 3 and 4, and we get our ordered array of Arr = {11, 12, 22, 25, 64}.

2.2 Insertion Sort

Concept

The general idea behind insertion sort is:

- Extract the first two items in the original array, and place them in the correct relative order
- Access the third item, and insert it in the appropriate order relative to the first two
- Access the fourth item, and insert it in the appropriate order relative to the first three
- Repeat this pattern until all points are ordered

In algorithmic form, we state it as follows:

- Use an outer loop from $i = 1\ to\ i = N - 1$
- Use a nested loop to choose the next index, and determine its position relative to the already sorted data.
- Insert the current point in the appropriate position
- Repeat the process until completion.

Also, note that in order to insert an item into its place in the (relatively) sorted part of the array, it is necessary to move some values to the right to make room (this is a relatively inefficient process in arrays).

Source Code

```
public static void insertionSort(Comparable[] A) {
  int k, j;
  Comparable tmp;
  int N = A.length;

  for (k = 1; k < N, k++) {
     tmp = A[k];
     j = k - 1;
     while ((j > = 0) && (A[j].compareTo(tmp) > 0)) {
         A[j+1] = A[j]; // move one value over one place to the right
         j--;
     }
     A[j + 1] = tmp; // insert kth value in correct place relative to
previous
                     // values
  }
}
```

Complexity

The inner loop can execute a different number of times for every iteration of the outer loop. In the worst case, the 1st iteration of the outer loop results in one execution of the inner loop, and likewise increase for each iteration (ie. The 2nd iteration of the outer loop can require two executions of the inner loop… and so on).

This gives us a sum of $\sum_{i=1}^{N-1} i$, which is still on the order of $O(N^2)$. Due to its structure, Selection Sort is very quick in small data sets (outperforming quicksort and many others), but inefficient in large sets.

Example of Insertion Sort

Suppose we are given an Array = {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the key index under consideration is underlined and the previous step was in bold.

Iteration 1: {**3** <u>7</u> 4 9 5 2 6 1} 3 and 7 are already in order. So do nothing.

Iteration 2: {3 **7** <u>4</u> 9 5 2 6 1} 4 is less than 7, but greater than 3. So we insert 4 into index 1, and move 7 to index 2 giving us {3 4 7 9 5 2 6 1}

Iteration 3: {3 **4** 7 <u>9</u> 5 2 6 1} 3,4,7,9 are already in order. So do nothing

Iteration 4: {3 4 7 **9** <u>5</u> 2 6 1} 5 is less than 7 and 9, so we insert it indo index 2, and shift everything afterwards by 1 to the right giving us {3 4 5 7 9 2 6 1}

Iteration 5: {3 4 **5** 7 9 <u>2</u> 6 1} 2 is less than all preceding entries, so we put it in index 0, and shift all over by 1 to the right giving us {2 3 4 5 7 9 6 1}

Iteration 6: {**2** 3 4 5 7 9 <u>6</u> 1} 6 is less than 7 and 9, thus we put in front and shift 7 and 9 by one index to the right giving us {2 3 4 5 6 7 9 1}

Iteration 7: {2 3 4 5 **6** 7 9 <u>1</u>} 1 is less than all preceding entries, so we put it in the zero index and shift all over by 1 to the right giving us 1 2 3 4 5 6 7 9.

2.3 Bubble Sort

Concept

Starting from the beginning of the list, this algorithm compares every adjacent pair, swap their position if they are not in the right order. After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

Complexity

A Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. An advantage of bubble sort is that it is able to detect if the list is already sorted (ie. An excellent best case scenario of $O(n)$).

Example of Insertion Sort

Suppose we are given the list {6 5 3 1 8 7 2 4}

It would be processed as follows: {5 6 3 1 8 7 2 4} -> {5 3 6 1 8 7 2 4} -> {5 3 1 6 8 7 2 4} -> {5 3 1 6 8 7 2 4} -> {5 3 1 6 8 7 2 4} -> {5 3 1 6 7 8 2 4} -> {5 3 1 6 7 2 8 4} -> {5 3 1 6 7 2 4 8}

{3 5 1 6 7 2 4 8}-> {3 1 5 6 7 2 4 8}-> {3 1 5 6 7 2 4 8}-> {3 1 5 6 7 2 4 8}-> {3 1 5 6 2 7 4 8}-> {3 1 5 6 2 4 7 8}-> {3 1 5 6 2 4 7 8}

{3 1 5 6 2 4 7 8} -> {1 3 5 6 2 4 7 8}-> {1 3 5 6 2 4 7 8}-> {1 3 5 6 2 4 7 8}-> {1 3 5 2 6 4 7 8} -> {1 3 5 2 4 6 7 8} -> {1 3 5 2 4 6 7 8} -> {1 3 5 2 4 6 7 8}

{1 3 5 2 4 6 7 8} -> {1 3 5 2 4 6 7 8} -> {1 3 2 5 4 6 7 8} -> {1 3 2 4 5 6 7 8} -> {1 3 2 4 5 6 7 8} -> {1 3 2 4 5 6 7 8}

{1 3 2 4 5 6 7 8} -> {1 2 3 4 5 6 7 8} Sorted

2.4 Merge Sort

Concept

The general idea behind merge sort is (a divide and conquer approach):

- Divide the array into two halves
    - Recursively, sort the left half
    - Recursively, sort the right half
- Merge the two sorted halves.

The base case for the recursion is when the array to be sorted is of length 1 -- then it is already sorted, so there is nothing to do. We should note that the merge step requires an auxiliary array to avoid overwriting values. The sorted values are then copied back from the auxiliary array to the original array.

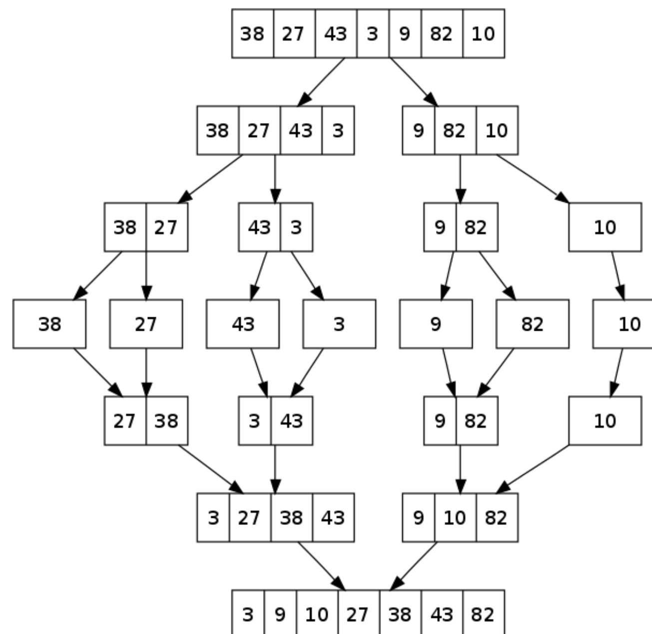
Complexity

This divide and conquer type approach can be conceptually thought of as a tree type structure with height $O(\log N)$. The total work done at each level of this tree is on the order $O(N)$ due to the merge step (excluding recursive calls).

Step 1 (finding the middle index) is $O(1)$, and this step is performed once in each call (once at the top level, twice at the seconds, and so on, down to a total of $\frac{N}{2}$ times at the penultimate level). So for any one level, the total amount of work done for step 1 is at most $O(N)$.

For each individual call, step 4 (merging the sorted half-graphs) takes time proportional to the size of the part of the array to be sorted by that call. So for an entire level, the time is proportional to the sum of the sizes at that level. This sum is always N. Thus, the time for merge sort involved $O(N)$ work done at each level of the recursive call. Since there are $O(\log N)$ levels, the worst case is $O(N \log N)$.

Example of Insertion Sort



Source Code

```
public static void mergeSort(Comparable[] A) {
  mergeAux(A, 0, A.length - 1); // call the aux. function to do all the
work
}

private static void mergeAux(Comparable[] A, int low, int high)
{
  // base case
     if (low == high) return;

  // recursive case
  // Step 1: Find the middle of the array (conceptually, divide it in
half)
     int mid = (low + high) / 2;
  // Steps 2 and 3: Sort the 2 halves of A
     mergeAux(A, low, mid);
     mergeAux(A, mid+1, high);

  // Step 4: Merge sorted halves into an auxiliary array
     Comparable[] tmp = new Comparable[high-low+1];
     int left = low;    // index into left half
     int right = mid+1; // index into right half
     int pos = 0;       // index into tmp

     while ((left <= mid) && (right <= high)) {
  // choose the smaller of the two values "pointed to" by left, right
  // copy that value into tmp[pos]
  // increment either left or right as appropriate
  // increment pos
  ...
     }
     // here when one of the two sorted halves has "run out" of values,
but
     // there are still some in the other half; copy all the remaining
values
     // to tmp
     // Note: only 1 of the next 2 loops will actually execute
       while (left <= mid) { ... }
       while (right <= high) { ... }

     // all values are in tmp; copy them back into A
       arraycopy(tmp, 0, A, low, tmp.length);
}
```

2.5 Quicksort

Concept

Quick sort (like merge sort) is a divide and conquer algorithm. The idea is to partition into (ideally equal size) subarrays. The subarrays would be (recursively) sorted. In ideal circumstances we would like to put exactly half the values in each partition, but this would require calculating the median, which is expensive. Thus, a pivot is arbitrarily selected, and we divide the "halves" by putting those less than the pivot in one group, and those greater in the other.

The algorithm is outlined as follows:

- Choose a pivot value.
- Partition the array (put all value less than the pivot in the left part of the array, then the pivot itself, then all values greater than the pivot).
- Recursively, sort the values less than the pivot.
- Recursively, sort the values greater than the pivot.

Although, the approach could be implemented such that it works all the way down to a single item, in practice it more practical to revert to a sort like insertion sort, which is efficient for small sets (order of 20 items).

Concept: Choosing a Pivot

Choosing a pivot is a challenge. Ideally, there would be an even distribution between partitions; the more uneven the distribution, the worse the runtime.

One approach is to use the first value, however, if the array is already sorted this leads to the worst possible runtime. In this case, after partitioning, the "smaller than pivot" (or left) partition will be empty and the other partition full. In other words, we waste $O(N)$ in comparisons, and this will cause $O(N)$ in recursive calls to be made, which will make the entire sort work in $O(N^2)$

Another option is to use a random-number generator (or a bound random number generator if we have some knowledge on the range of values in our array).

Alternatively, schemes like "median-of-three", where the median value of 3 randomly selected points (choosing the first, last and mid index… median(Arr[0], Arr[N], Arr[N/2]). This will avoid the worst case scenario if already sorted.

Source Code

Here's the actual code for the partitioning step (the reason for returning a value will be clear when we look at the code for quick sort itself):

```
private static int partition(Comparable[] A, int low, int high) {
// precondition: A.length >= 3

    int pivot = medianOfThree(A, low, high); // this does step 1
    int left = low+1; right = high-2;
    while ( left <= right ) {
        while (A[left].compareTo(pivot) < 0) left++;
        while (A[right].compareTo(pivot) > 0) right--;
        if (left <= right) {
            swap(A, left, right);
            left++;
            right--;
        }
    }
    swap(A, left, high-1); // step 4
    return right;
}
```

After partitioning, the pivot is in A[right+1], which is its final place; the final task is to sort the values to the left of the pivot, and to sort the values to the right of the pivot. Here's the code for quick sort (so that we can illustrate the algorithm, we use insertion sort only when the part of the array to be sorted has less than 3 items, rather than when it has less than 20 items):

```
public static void quickSort(Comparable[] A) {
    quickAux(A, 0, A.length-1);
}

private static void quickAux(Comparable[] A, int low, int high) {
    if (high-low < 2) insertionSort(A, low, high);
    else {
        int right = partition(A, low, high);
  quickAux(A, low, right);
  quickAux(A, right+2, high);
    }
}
```

Complexity

On average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare.

3 Search

Consider searching for a given value V in an array of size N. There are two basic approaches: sequential search and binary search.

3.1 Sequential

Sequential search involves looking at each value in turn (i.e., start with the value in array[0], then array[1], etc). The algorithm quits and returns true if the current value is v; it quits and returns false if it has looked at all of the values in the array without finding v. Here's the code:

```
public static boolean sequentialSearch(Object[] A, Object v) {
    for (int k = 0; k < A.length; k++) {
        if (A[k].equals(v)) return true;
    }
    return false;
}
```

If the values are in sorted order, then the algorithm can sometimes quit and return false without having to look at all of the values in the array: v is not in the array if the current value is greater than v. Here's the code for this version:

```
public static boolean sortedSequentialSearch(Comparable[] A, Comparable
v) {
// precondition: A is sorted (in ascending order)
    for (int k = 0; k < A.length; k++) {
        if (A[k].equals(v)) return true;
  if (A[k].compareTo(v) > 0) return false;
    }
    return false;
}
```

The worst-case time for a sequential search is always O(N).

3.2 Binary

When the values are in sorted order, a better approach than the one given above is to use binary search. The algorithm for binary search starts by looking at the middle item x. If x is equal to v, it quits and returns true. Otherwise, it uses the relative ordering of x and v to eliminate half of the array (if v is less than x, then it can't be stored to the right of x in the array; similarly, if it is greater than x, it can't be stored to the left of x). Once half of the array has been eliminated, the algorithm starts again by looking at the middle item in the remaining half. It quits when it finds v or when the entire array has been eliminated.

Here's the code for binary search:

```
public static boolean binarySearch(Comparable[] A, Comparable v) {
// precondition: A is sorted (in ascending order)
    return binarySearchAux(A, 0, A.length - 1, v);
}

private static boolean binarySearchAux(Comparable[] A, int low, int
high, int v) {
// precondition: A is sorted (in ascending order)
// postcondition: return true iff v is in an element of A in the range
//                A[low] to A[high]
    if (low > high) return false;
    int middle = (low + high) / 2;
    if (A[middle].equals(v)) return true;
    if (v.compareTo(A[middle]) < 0) {
        // recursively search the left part of the array
        return binarySearchAux(A, low, middle-1, v);
    }
    else {
        // recursively search the right part of the array
        return binarySearchAux(A, middle+1, high, v);
    }
}
```

The worst-case time for binary search is proportional to log2 N: the number of times N can be divided in half before there is nothing left. Using big-O notation, this is O(log N). Note that binary search in an array is basically the same as doing a lookup in a perfectly balanced binary-search tree (the root of a balanced BST is the middle value). In both cases, if the current value is not the one we're looking for, we can eliminate half of the remaining values.

Sources:

[1] http://pages.cs.wisc.edu/~bobh/367/SORTING.html

[2] https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/#introduction-and-reveiew

[3] https://en.wikipedia.org/wiki/Sorting_algorithm