Matthew A. Krieger

September 20, 2015

CS 253: Data & File Structures

Homework I

Neli P. Zlatareva, Ph.D.

Homework I

**Problem I**

1. **Bubble Sort:**

|  |  |  |
| --- | --- | --- |
| Bubble Sort | | |
| Case: | Comparisons: | Exchanges: |
| Best Case: | 1,999 | 0 |
| Worst Case: | 1,999,000 | 1,999,000 |
| Average Case: | 1,997,541 | 1,005,445 |
| Trial Run #1: | 1,997,289 | 1,012,879 |
| Trial Run #2: | 1,997,725 | 1,001,113 |
| Trial Run #3: | 1,998,622 | 1002096 |
| Trial Run #4: | 1,996,722 | 1027694 |
| Trial Run #5: | 1,997,347 | 983444 |

**Best Case: O(N)**

**Average Case: O(N2)**

**Worst Case: O(N2)**

Bubble sort is simple, inefficient, and rarely used. It is an O(n2) algorithm.which makes it inefficient when dealing with large sets of data.

The algorithm is relatively simple:

a) compare each pair of adjacent elements from the beginning of an array and swap them if they are in reverse order.

b) Repeat step 1 if at least 1 swap occurred.

Sorting stops when no more swaps can be made.

When used on average & worse case scenarios the bubble sort’s complexity is O(n2). That being said, the bubble sort is adaptive. For almost sorted arrays it is estimated at O(n). It is very important to check if the array is sorted on every step in order to make the bubble sort adaptive.

1. **Insertion Sort**

|  |  |  |
| --- | --- | --- |
| Insertion Sort | | |
| Case: | Comparisons: | Exchanges: |
| Best Case: | 1,999 | 0 |
| Worst Case: | 1,999,000 | 1,999,000 |
| Average Case: | 998,864 | 996,873 |
| Trial Run #1: | 995,966 | 993,972 |
| Trial Run #2: | 975,311 | 973,320 |
| Trial Run #3: | 1,007,213 | 1,005,224 |
| Trial Run #4: | 1,027,316 | 1,025,327 |
| Trial Run #5: | 988,514 | 986,524 |

Insertion sort is a an O(n2) sorting algorithm but when dealing with a sorted array – it performs better. Usually, the insertion sort is applied towards small arrays (ie. Deck of cards). In insertion sort, the array is divided into a sorted section and an unsorted section. At the start, the sorted section contains the first element and the sorted section contains the rest. With every step, the algorithm removes the first element of the unsorted section & places it in the appropriate location in the sorted section. The algorithm stops when there are no more elements in the unsorted section.

**Best Case: O(N)**

**Average Case: O(N2)**

**Worst Case: O(N2)**

1. **Selection Sort**

|  |  |  |
| --- | --- | --- |
| Selection Sort | | |
| Case: | Comparisons: | Exchanges: |
| Best Case: | 1,999,000 | 0 |
| Worst Case: | 1,999,000 | 1000 |
| Average Case: | 1,999,000 | 1,991 |
| Trial Run #1: | 1,999,000 | 1,993 |
| Trial Run #2: | 1,999,000 | 1,992 |
| Trial Run #3: | 1,999,000 | 1,988 |
| Trial Run #4: | 1,999,000 | 1,990 |
| Trial Run #5: | 1,999,000 | 1,993 |

Selection Sort is an O(n2) algorithm, which makes it very inefficient when dealing with large amounts of data. Despite being inefficient, it is simple to implement.

In selection sort, the array is divided into two parts (sorted & unsorted). At the start of the algorithm, the sorted section has no elements and the unsorted section contains every element. With every iteration, the algorithm searches the unsorted section for the minimum element & moves it to the beginning of the sorted section. Every step decreases the unsorted elements by one. As a result, the number of steps in the outer loop of selection sort is equal to the number of elements. Therefore, n + (n-1) + (n-2) + … +1 makes the selection sort O(n2) when dealing with number of comparisons. The algorithm stops when the unsorted section becomes empty.

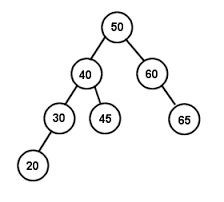
**Best case: O( N2 )**

**Average case: O( N2 )**

**Worst case: O( N2 )**

**Problem II**

1. Binary Search (Recursive):



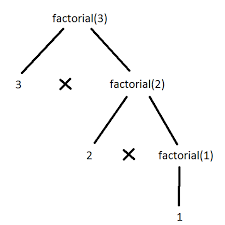
In binary search we know that each time we call our method it will calculate, and compare, the mid point in our data set with the key that we are searching for. Assuming the initial mid point value is not equal to our key, the search will then call itself again (recursively), using the midpoint as either the new lower or upper boundary, based on the results of its previous iteration. This means that, in the worst case, binary search efficiency is O( log2 N ). This is because binary search continuously calls upon a new lower and upper boundary, thus diving its search range in half each time until reaching the target value. Therefore, binary search will run in a number of steps proportional to the logarithm of the length of the list being searched each time.

**Best case: O( 1 )**

**Average case: O( log2 (n) - 1 )**

**Worst case: O( log2 n )**

2. Factorial:



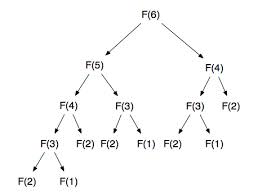
When computing the Nth factorial (recursively) the program must first calculate all previous values down to 1. This causes the efficiency of this recursive call to decrease (linearly) as its input increases. However, unlike Fibonacci's number, this recursive call only needs to call upon itself once because the second value is predetermined to be the input value for the method itself. Therefore this is exponentially more efficient than Fibonacci's number, and even more efficient than binary search as well.

**Best case: O( 1 )**

**Worst case (recursive): O( 2N )**

**Worst case (iterative): O(n)**

3. Fibonacci:



Fibonacci's number has very unique characteristics because, recursively, the method N will call (N-1) and (N-2). This efficiency is better represented, in its worse case, as 2N. This is the least efficient algorithm that we examined in this homework because, for each number calculated, it needs to calculate all the previous numbers more than once. As we can see from just a recursive call of (6) this algorithm expands exponentially based on its input. Alternatively, this can also be called iteratively to actually increase it to a linear efficiency of N. This is because, when using iteration, you no longer need to calculate every previous number more than once. Instead you can start from the bottom up, thereby only having to calculate each value one time.

**Best case: O( 1 )**

**Worst case (recursive): O( 2N )**

**Worst case (iterative): O(n)**

**Source Code:**