CS 430 – Fall 2024
Introduction To Algorithms
Homework #1

1a) SelectionSort (array):

```
for i = 1 to array.length - 1
    min_idx = i
    for j = i + 1 to array.length
        if array[j] < array[min_idx]
            min_idx = j
    swap array[i] with array[min_idx]
```

1b) The loop invariant for the outer loop is that at the start of each iteration i, the subarray from array[1] to array[i - 1] contains the i - 1 smallest elements in sorted order. These elements are the smallest elements in the original array and do not change.

1c) After placing the smallest n − 1 elements in their correct positions, the final element at array[n] has to be the largest, since there is no smaller element left to swap. So, we can conclude that the last element is already correctly placed.

1d) Best and Worst Case: $\Theta(n^2)$. The algorithm always performs the same number of comparisons, regardless of whether the input is already sorted or not. Even in the worst case, the algorithm still makes the same number of comparisons.

1e) Selection Sort is not stable. A sorting algorithm is considered stable if equal elements retain their relative order. Since Selection Sort involves swapping elements, it can change the relative order of equal elements if one of the equal elements is selected and swapped from a later position in the array.

2) In the worst-case scenario, each recursive call to binary search divides the input array size by half, continuing this until the size of the array is 1 (base case).

Method 1: Array is passed by a pointer Time = $\Theta(1)$
Recurrence: The array is passed by pointer, which takes constant time $\Theta(1)$. The only cost is the recursive call itself, which divides the problem size n by 2 at each step.

$T(n) = T(n/2) + 2$
$T(n) = \Theta(\log N)$

Method 2: Array is passed by copying Time = $\Theta(N)$

Recurrence: Here, each time the array is passed to the recursive function, the entire array is copied, which takes Θ(n). Thus, the recurrence relation becomes:

$T(n) = T(n/2) + N$

$T(n) = T(n/2) + cN$
$T(n) = 2cN + T(n/4)$
$T(n) = 3cN + T(n/8)$
$T(n) = \sum_{i=0}^{logn-1} (2^i cN/2^i)$
$T(n) = cN \log n$
$T(n) = \Theta(N \log N)$

Method 3: Array is passed by copying only the subrange that might be accessed
Time = Θ(q - p + 1) = Θ(n)
Recurrence: In this case, only the relevant subarray A[p…q] is copied. The size of the subarray is n = q − p + 1.

$T(n) = T(n/2) + \Theta(n)$
$T(n) = \Theta(N)$

3)
Base Case (n = 3):
For n = 3, the recurrence gives:

$T(3) = 9$

According to the proposed solution $T(n) = n^2$, for n = 3:

$T(3) = 3^2 = 9$

Thus, the base case holds.

Inductive Hypothesis:
Assume that for some $k \geq 1$, the recurrence holds for $n = 3^k$

Inductive Step:
We need to show that the recurrence holds for $n = 3^{k+1}$
By the recurrence relation:

$$T(3^{k+1}) = 6T(3^{k+1}/3) + \frac{1}{3}(3^{k+1})^2$$

$$T(3^{k+1}) = 6T(3^k) + \frac{1}{3}(3^{k+1})^2$$

Using the inductive hypothesis $T(3^k) = 9^k$

$$T(3^{k+1}) = 6 \cdot 9^k + \frac{1}{3}(9^{k+1})$$

$$T(3^{k+1}) = 6 \cdot 9^k + 3(9^k)$$
$$T(3^{k+1}) = (6 + 3)9^k$$
$$T(3^{k+1}) = (9)9^k = 9^{k+1}$$

Thus, we have shown that $T(3^{k+1}) = 9^{k+1}$, which proves that the solution to the recurrence is $T(n) = n^2$, when n is an exact power of 3

4a) $T(n) = T(n - 1) + n$ with $T(1) = O(1)$
Solution:

$T(n) = T(n - 1) + n$
$T(n - 1) = T(n - 2) + (n - 1)$
$T(n - 2) = T(n - 3) + (n - 2)$

Continuing this pattern, after k steps:

$T(n) = T(n - k) + n + (n - 1) + (n - 2) + \cdots + (n - k + 1)$

If we expand all the way to T(1), we get:

$T(n) = T(1) + (n + (n - 1) + (n - 2) + \cdots + 2 + 1)$

The sum of the first n integers is:

$T(n) = O(1) + (n(n + 1)/2) = O(n^2)$

Thus, the solution is:

$T(n)=O(n^2)$

7a) Each of the n/k sublists has length k, and each of these sublists is sorted using insertion sort. The time complexity in the worst case for insertion sort on a list of length k is $\Theta(k^2)$. Thus, sorting one sublist of length k takes$\Theta(k^2)$, and since there are n/k sublists, the total time to sort all sublists is $T = (n/k) \cdot \Theta(k^2) = \Theta(nk)$

7b) After sorting the n/k sublists using insertion sort, we need to merge them. There are n/k sublists, and merging m sublists takes $\Theta(n \log m)$ time. Thus, merging n/k sublists of total length n takes: $T = \Theta(n \log (n/k))$

7c) The total running time of the modified algorithm is given by: $T(n) = \Theta(nk) + \Theta(n \log(n/k))$
We want to find the largest value of k such that the running time remains asymptotically equal to the standard merge sort, which runs in $\Theta(n \log n)$.
To satisfy this condition, k cannot grow faster than log n asymptotically, if it does then because of the nk term, the algorithm will run at worse asymptotic time than $\Theta(n \log n)$
Assume, $k = \Theta(\log n)$

$\Theta(nk + n \log(n/k))$
$= \Theta(nk + n \log n - n \log k)$
$= \Theta(n \log n + n \log n - n \log(\log n))$

$= \Theta(2n \log n - n \log(\log n))$
$= \Theta(n \log n)$

7d) In practice, k should be chosen as a small constant value because insertion sort is faster for small inputs due to its low constant factors, even though it has worse asymptotic performance. Modern computer architectures are optimized for small input sizes, and insertion sort often outperforms merge sort due to better cache utilization and lower overhead.