Homework 1 Solutions

---

(1) Compare and contrast the `linearSearch` and `binarySearch` algorithms by searching for numbers 45 and 54 in the following list: $(3, 8, 12, 34, 54, 84, 91, 110)$.
Answer:
—Searching for 45: 45 does not exist in the list. With linear search you have to compare all 8 elements to see this. With binary search, you first check the middle element in position 3 (34), then the element at position 5 (54), then the element at position 4 (34); so binary search checks 3 elements in total, while linear search checks 8.
—Searching for 54: Linear search finds it with 5 comparisons. Binary search needs only 2: first it checks element at position 3 (34), then the element at position 5 (54).

(2) Consider an array A of n elements, $a[0]...a[n-1]$ and the pseudocode for `bubbleSort` discussed in class:

```
for k = 1 to n-1 do
     for i = 0 to n-1-j
            if (A[i] > A[i+1])    swap A[i] and A[i+1]
```

(a) Describe what happens when you bubblesort an array A that is already sorted. How many swaps are performed by the inner loop each time? How many times is the outer loop executed?

Answer: If the array is already sorted, the first iteration of the inner loop performs no swap, and at that point the outer loop should stop. Instead, the outer loop executes n-1 times. The inner loop executes n-1-j times; even though it performs no swaps, it still needs to run n-1-j times. Overall, if the array is already sorted, Bubblesort as written above takes $\Theta(n^2)$ time.
(b) Show how you can change the code so that it exits the loop early if the inner loop performs no swap. Try to add as little extra code as possible. You can change the `if` to a `while`, or, you can look into exiting a loop using `break`.

Answer: Here is one way to do it:

```
for k = 1 to n-1 do
     //initialize this iteration
     swapped = false
     for i = 0 to n-1-j
            if (A[i] > A[i+1])
                   swap A[i] and A[i+1]
                   swapped = true
     //if no swap was performed in this iteration, then we're done
```

```
if swapped== false break
```

In this case, if the array is already sorted, Bubblesort stops after one iteration of the inner loop, in $\Theta(n)$ total time. With this modification, the best case is $\Theta(n)$ while the worst case is still $\Theta(n^2)$. Note that the case when the array is already sorted is just one instance where the running time is improved. Any other scenario where Bubblesort does not need all n-1 passes will be detected by the algorithm above and it will exit early.

(3) You are playing a game where your task is to guess the value of a hidden number that is one of n integers between 0 and $n-1$. For simplicity, we'll asume that n is a power of 2. Each time you make a guess, you are told whether your guess is too high or too low.
One strategy for playing this game is to guess 0, then 1, then 2, then 3, and so on, until hitting the hidden number. How long would it take you to guess the hidden number, in the worst case?
Describe a better strategy for playing this game and analyse it.

Answer: The first strategy take $\Theta(n)$ trials in the worst case to guess the number, since the hidden number can be the very last one we try.
A better strategy is to keep track of the interval where the number can be, and always guess the number in teh middle of the interval. Initially we know that the number x is in $[0, n-1]$. We guess $n-1/2$; if $x > n-1/2$ the interval becomes $[(n-1)/2+1, n-1]$; otherwise the interval becomes $[0, (n-1)/2-1]$. In the worst case this takes $\Theta(\lg n)$ guesses.

(4) Given an array of n real numbers, sketch how you'd find the pair of numbers that are closest in value.

Answer: One way to do this is to compute, for every element $a[i]$, its closest element: that is, the element $a[j]$ such that $|a[j]-a[i]|$ is smallest in value, among all $j = 1..n, j \neq i$. Then find the smallest such pair, across all $i = 1..n$. For one element, finding its closest element takes $\Theta(n)$ time. Doing this for every element i and finding the overall minimum takes $\Theta(n^2)$.
Another solution is to sort the array. The claim is that the pair of elements that are closest in value must be in consecutive indices of the sorted array. You should at least try to argue why this is true. Let's assume that the array is sorted in increasing order.

$$a[0] \leq a[1] \leq a[2] \leq ...$$

This means that $a[i+1]-a[i] \geq 0$ for all i. Now let's look at the difference in value between two elements at distance 2 apart, $a[i+2]$ and $a[i]$:

$$a[i+2]-a[i] = a[i+2]-a[i+1]+a[i+1]-a[i]$$

and since both $a[i+2] - a[i+1] \geq 0$ and $a[i+1] - a[i] \geq 0$ it follows that

$$a[i+2] - a[i] \geq a[i+2] - a[i+1]$$

and

$$a[i+2] - a[i] \geq a[i+1] - a[i]$$

With this property, it now suffices to do one pass over the sorted array and compute differences in value between consecutive elements, and keep track of the largest one. Once the array is sorted, this takes $\Theta(n)$ time. Sorting the array with teh methods we learnt (insertion sort, bubble sort, selection sort) takes $\Theta(n^2)$, so this is not better than the first solution. However we will see later that it is possible to sort in $\Theta(n \lg n)$.

(5) Same problem as above, but find the pair of numbers that are farthest apart in value.

Answer: The observation is that the numbers that are farthest apart are the smallest and teh largest numbers in the array. One way to solve this is to sort the array and then output the first and the last. The complexity of this is the complexity of the sorting algorithm you use.
Another way to do this is to notice that you do not need to sort the array in order to find the smallest and the largest elements. You can find each of them in one pass through the array. This approach therefore runs in $\Theta(n)$ time.