Assignment #2 Solutions

1) Let $k$ be a fixed natural number. Consider the family $A_k$ of all arrays $A[1..n]$ satisfying that for every $i \leq n$ there are at most $k$ elements among $A[1..i-1]$ larger than $A[i]$. Show that there exists a constant $C$ such that every array $A[1..n]$ from $A_k$ can be sorted in time $C \cdot n$.

We use Insertion Sort, and note that at stage $i$ there are at most $k$ elements that $A[i]$ will be compared with, and that will be shifted one cell to the right. Thus altogether we have at most $\sum_{i=2}^{n} k = (n-1)k = O(n)$ many steps ($k$ is constant and independent of $n$).

2) Assume that an array $A[1..n]$, consisting of $n$ numbers is sorted (ascending order). Design an algorithm which, given a number $m$, outputs the largest number $1 \leq i \leq n$ such that $A[i] < m$, or 0 if $m \leq A[1]$, and which runs in time $\Theta(lg n)$.

Use binary search; thus, first compare $m$ with $A[n/2]$, and if $A[n/2] < m$ compare $A[3n/4]$ and $m$, etc.

3) Can you use the previous algorithm to make the insertion sort run in time $\Theta(n \lg n)$? What if instead of an array of numbers you have a linked list of numbers?

The answer is no: if we have an array, then searching for the proper place to insert $A[i]$ is fast, but inserting $A[i]$ is slow because we might have to move $i-1$ elements to the right. If we have a linked list, then inserting is fast (constant time) but searching is slow: we must traverse the linked list to find the proper place; binary search does not work with linked lists; see the diagram below.
4) Describe a $\Theta(n \log n)$ algorithm that, given a set $S$ of $n$ integers and another integer $x$, determines whether or not there exist two elements in $S$ whose sum is exactly $x$.

Use Merge Sort to sort the array in $\Theta(n \log n)$ steps (worst case). Then go through the array and for each element $A[i]$ use binary search to determine if $x - A[i]$ is in the array.

5) Determine if $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$ or $f(n) = O(g(n))$ for the following pairs:

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$g(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>$(n - 2 \log(n))(n + \cos(n))$</td>
</tr>
<tr>
<td>$(\log(n))^2$</td>
<td>$\log(n^{\log_2 n}) + 2 \log(n)$</td>
</tr>
<tr>
<td>$n^{(1+\sin(\pi n/2))/2}$</td>
<td>$\sqrt{n}$ (a plot might help)</td>
</tr>
</tbody>
</table>

a) It is easy to see that $\frac{1}{2} n^2 < (n - 2 \log(n))(n + \cos(n)) < 2 n^2$. Thus, $n^2 = \Theta(n - 2 \log(n))(n + \cos(n))$.

b) $\log(n^{\log(n)}) + 2 \log(n) = \log(n) \log(n) + 2 \log(n) = \Theta((\log(n))^2)$

c) Notice that for $n$ divisible by 4, i.e., for $n = 4k$ we have $\sin(\pi n/2) = \sin(2k\pi) = 1$, and that for $n$ of the form $n = 4k + 2$ we have $\sin(\pi n/2) = \sin((2k+1)\pi) = \sin(\pi) = 0$. Thus, for $n$ divisible by 4 we have $n^{(1+\sin(\pi n/2))/2} = n$, and for $n$ of the form $n = 4k + 2$ we have $n^{(1+\sin(\pi n/2))/2} = n^0 = 1$. Since $\sqrt{n} = n^{1/2}$, we have that for arbitrary $C$ there are arbitrarily large $n$ such that $Cn^{(1+\sin(\pi n/2))/2} < \sqrt{n}$ and arbitrarily large $n$ such that $\sqrt{n} < Cn^{(1+\sin(\pi n/2))/2}$. Thus, neither is $n^{(1+\sin(\pi n/2))/2} = O(\sqrt{n})$ nor $\sqrt{n} = O(n^{(1+\sin(\pi n/2))/2})$. Note that this means that there are functions whose asymptotic growth are incomparable. This is quite unlike numbers, that satisfy that for any two numbers $n, m$ either $m \leq n$ or $n \leq m$. 


6) Bubblesort sorts by repeatedly swapping adjacent elements that are out of order.

BUBBLESORT(A)
1. for i ← 1 to length[A]-1 do
2. for j ← length[A] downto i+1 do

a. Justify why Bubble sort works, i.e., explain why the final sequence must be sorted.
b. Implement the Bubblesort and compare its performance with your implementation of Quick Sort and Insertion Sort for randomly generated sequences of 100 numbers, 1000 numbers and 10,000 numbers.

a) Just notice that in the first pass of the first loop (line 1) the smallest element is put into the first place, after the second pass the second smallest etc. It is easy to prove by induction that after the $i^{th}$ pass the $i^{th}$ element is put in its proper place. Thus after length[A]-1 many passes length[A]-1 elements are in its proper place, and thus, automatically so is the last element.

7) Assume you are working for Google and have to design a fast algorithm that, given $n$ pages with their page ranks, it selects $\log_2 n$ pages with the largest possible ranks. Your algorithm must be very fast; it must run in only $O(n)$ steps.

Just abort the HeapSort after $\log_2(n)$ many steps. Since one can build a heap in $O(n)$ many steps and then performs $\log_2(n)$ many times Heapify that each takes less than $\log_2(n)$ many steps, the whole procedure takes $O(n) + O(\log_2(n)^2) = O(n)$ many steps. Notice that $\log_2(n)^2 = O(n)$ because

$$\lim_{x \to \infty} \frac{(\log x)^2}{x} = \lim_{x \to \infty} \frac{d}{dx} \log x = \lim_{x \to \infty} \frac{2 \log x}{x} = \frac{2 \log 1}{1} = \frac{2 \log 2}{1} = 0$$

Thus, for sufficiently large $n$ we have $\log_2(n)^2 < n$. 

8) Again you are a Google employee and have to design an algorithm that, given $k$ sets of $n$ pages each such that each set is sorted by page rank, it merge these sets of pages into a single set (array) of $nk$ pages, also sorted by page rank. Google wants your algorithm to be extremely fast, and to work in only $O(n \log k)$ many steps.

We give two solutions, one uses ideas from the Heap Sort, the other from the Merge Sort.

1. First be build a heap whose elements are these $k$ sorted arrays, according to their maximal element. Notice that the elements of the heap are NOT all elements of all arrays, but arrays themselves (see the picture).

First notice that the top element of the top array in the heap must be larger than any other element of any array, because it is larger than the top element of any array (heap property) and all arrays are sorted, so every element of an array is smaller than the top element of that array. Now start filling a new array of size $nk$ by taking the top element of the top array in the heap and placing it in the new array starting from the rightmost place. After the top element is removed from the top array, the next element of the same array becomes the top element of this array, and the heap property might be violated. We now the heap by reordering the arrays (not by moving the top elements only). This ensures that the top element of every array is larger than every other element of the same array. Once an array is exhausted, we take rightmost leaf array and place it at the top and Heapify. We continue this process until all elements are moved.
2. Organize these arrays as *leaves* of a binary tree, not sorted in any way (not a heap or similar), as shown on the picture:

We now merge all pairs of arrays with $n$ elements that are on the leaves into arrays with $2n$ elements on height 1, merging them in turn into arrays on level 2 with $4n$ elements and so forth, until all arrays are merged into a single array at the root of the tree. The height of a binary tree with $k$ leaves is $\log 2k = \log k + 1$, because a binary tree with $k$ leaves has at most $2k$ nodes in total. At any height the total work is linear in the number of elements i.e., it is always $O(k \cdot n)$. Since the total number of levels is $\log k + 1$ the total number of steps is $O(n \cdot k \cdot \log k)$, as required.

Notice that we could not merge arrays 1 and 2, then merge the resulting array with array 3, and so forth, eventually merging the result of merging arrays 1 through $k-1$ with array $k$. If we did so, the number of steps at stage $i$ i.e., when merging the result of merging arrays 1 through $i$ with the array $i+1$ is $(i+1)n$ because this many elements are involved in merging at stage $i$. Thus, all together we would have

$$\sum_{i=1}^{k-1} n \cdot (i + 1) = n \sum_{i=1}^{k-1} (i + 1) = n \cdot \frac{k(k-1)}{2} = O(nk^2)$$

many steps.