**Question 1: Suppose A is already sorted in increasing order. Prove that the running time of Quicksort on input A is (n2).**

When running quicksort, we will need to scan the entire array as usual, except that the partitioning will generate an array of 0 elements and one of (n-1) elements.

Looking at the following example of an array of 10 elements with A[l] = A[0] = 0 and A[r] =A[9] = 9, we can observe that ‘9’ (the pivot in green) will be successively compared to elements 0 to 8, moving the index k from k = 0 to the position k = 8.
A = {0,1,2,3,4,5,6,7,8,9}

Afterwards, Quicksort will be called recursively on the ‘left’ array, A[0..8] and on the right array A[9..9], which only contains a single element and is therefore already sorted.

The recursive call will repeat (n-1) comparisons until k = 7, and then call Quicksort on the ‘left’ array A[0..7] and on the right array, A[7..7], which once again only contains a single element.

This chain of nested calls on arrays of size (n-1) will have to repeat itself (n-1) times in order to reach the end of the recursion by calling A[0..0].

Looking at the time complexity:

T[n] = time to compare every element in the array to the pivot + time to perform Quicksort on both sub arrays

T[n] = n + T[0] + T[n-1]
T[n] = n + T[n-1]
T[n] = n + ((n-1) + T[n-2])
T[n] = n + (n-1) + (n-2) + T[n-3])
[…]
T[n] = n + (n-1) + (n-2) + […] + 2 + 1 + 0

T[n] = ∑i for i = 0 to i = n
T[n] = n(n-1)/2

Therefore, T[n] = O(n2)

**Question 2 (a) Suppose that A is in increasing order. What is the output?**

If A is sorted the output will be all the element indices of a strictly upper triangular matrix of size n x n:

(1,2), (1,3), (1,4), (…) , (1,n),

 (2,3), (2,4), (…) , (2,n),

 (3,4), (…) , (3,n),

 (…), (…),

 (n-1, n),

So the output will be n(n-1)/2 pairs that satisfy our requirements.

**(b) Now design an algorithm that solves the problem using divide-and-conquer technique. Your algorithm must run in time O(n ln n). (Hint: Review the Mergesort algorithm.)**

*The code presented below was taken from merge\_sort code in Wikipedia before being modified to accommodate this problem*

// First, calculate the maximum number of pairs the array could generate by using the

// relationship found in a):

define n = sizeOf(m)

global variable numberOfPairs = n(n-1)/2

// The ‘sort’ part of the algorithm remains unchanged from the original mergeSort



/\* The ‘merge’ part of the algorithm has boon modified to count the number of valid pairs the array contains. Based on the fact that when presented with 2 sorted array, the smallest element at the head of either array will also be smaller than the remaining of the other array. We can therefore substract sizeOf(array with largest head element) for the number of possible pairs that meet our requirements \*/



**(c) Use the Master Theorem to verify that the running time of your algorithm is O(n ln n).**

Running time:

T[n] = 2\*T[n/2] + n

Using master theorem with a = 2; b = 2, d = 1

We can observe that a = bd

So the teorem yields:

O(ndlogbn) = O(n1log2n) = O(k\*n\*ln(n)) = O(n\*ln(n))

**Question 3**

**(a) How would you represent a ternary heap in an array?**

A ternary heap is represented in an array similarly to a binary heap, with the exception that the relationship between parents and children would be different.

Every parent would have 3 children; a left child, a center child and a right child.

Given that the index of the parent is ‘i’, the indices of the children would be as follow:

Left child of i: 3i

Center child of i: 3i + 1

Right child of i: 3i + 2

To find the parent’s index of a child k, we would have to take the floor of k/3

i.e. parent of k = floor(k/3)

 **(b) What is the height of a ternary heap of n elements? Give a precise answer in terms of n.**

Size = Log3(n)

**(c) Give pseudo-code for Heapify3(A,i). This procedure assumes that the sub-trees rooted at the children of node i are ternary heap, but A[i] might be smaller than some of its children (thus violating the heap property). The procedure makes the sub-tree rooted at node i become a ternary heap by letting the value at A[i] “float down” in the ternary heap which is rooted at the largest child of node i. (This is a modified version of the Heapify(A,i) procedure.)**

We can assume that since leaves have no children, they are already heaps3. The maximum leaves a node can have is 3. So we should start heapify3 at the node i where i = floor(n/3) for a tree of n elements.

Heapify3(A, i){

 l = 3i; // Left child of i

 c = 3i + 1; // Center child of i

 r = 3i + 2; // Right child of i

 // Here we find the largest between the ith element and his 3 children (if they exist)

 if l < heapsize(A) and A[l] > A[i]

 largest <- l;

 if c < heapsize(A) and A[largest] < A[c]

 largest <- c;

 if r < heapsize(A) and A[largest] < A[r]

 largest <- r;

// Finally perform the swap to promote the largest element to the ith node and propagate the // recursive function until the node is a heap or we reach a leave

 If largest != I then

 Swap nodes A[i] <-> A[largest]

 Heapify3(A, largest);

}

**(d) Give pseudo-code for a sorting algorithm that uses Heapify3. The worst-case running time of the algorithm must be \_(n ln n).**

HeapSort3(A){

 Int heapSize = length(A);

 // First build the heap3

 For (i from floor(heapSize/3) down to 1) do{

 Heapify3(A, i);

 }

 For ( i from length(A) down to 2) do{

 Swap A[1] <-> A[i];

 heapSize = HeapSize – 1;

 Heapify3(A, 1);

 }

}

**(e) Verify that the worst-case running time of your algorithm is indeed \_(n ln n).**

Heapify3 will take at most 3 comparisons and at most h swaps as the node trickles down the tree, where h = height of the tree = log3n

So Heapify3 runs in O(log3n) time

Since we need to visit n/3 nodes and run Heapfy3 in order to construct a heap3, building a heap3 will take n/3 \* log3n, so O(n\*log3n)

So running the sorting algorithm will take n\*log3n time to build the heap3, then will call (n-2) times Heapify which can be performed in log3n time, so the entire running time is

T[n] = n\* log3n + (n – 2) log3n

T[n] = n\* log3n

**Question 4 Give an algorithm that, given an array A of n integers in the range 0 to k, preprocesses its input in such a way that any query about how many of the n integers fall into a range [a..b], (i.e., how many i such that a \_ A[i] \_ b) can be answered in time O(1). Explain how such a query is to be answered. Your algorithm should use \_(n + k) preprocessing time.**

We can use a very similar algorithm than ‘counting sort’ presented in class.

Processing algorithm:

Process(A[], k){ // Where A = input array and k = range from 0 to k

// Initialize counting array

For(x from 0 to k)

 Do C[x] <- 0;

// Count elements

For(I = 0 to length(A))

 C[A[i]]++;

}

// End of preprocessing. The array C can now be used to answer the queries in time O(1)

// Here is a non-destructive algorithm to answer the query; looking for the number of integers in the range [a, b], using the array ‘C’ computed in pre-processing and leaves a and b untouched.

Query(a, b, C[]){

 Int k = a;

 Int result = 0;

 While(k <= b){

 result = result + C[k];

 k++;

 }

 return result;

}