1. Solve each recurrence below and express $T(n)$ as the simplest $\Theta$ function of $n$.
   
a. $T(n) = 25 \, T(n/5) + n^3$
      $\Theta(n^3)$
   
b. $T(n) = 64 \, T(n/4) + n^3$
      $\Theta(n^3 \lg n)$
   
c. $T(n) = 81 \, T(n/3) + n^3$
      $\Theta(n^4)$
   
d. $T(n) = 2 \, T(n-1) + 1$
      $\Theta(2^n)$

2. We are given an array $A$ that contains $n$ distinct values. We are told that the array $A$ had once been sorted, but then it was rotated by some unknown number of positions. Example: The array below was rotated either 7 positions left or 10 positions right, but assume these rotation distances are unknown to us. Design an efficient divide-and-conquer algorithm that returns the smallest value in $A$. Also write a recurrence for the worst-case running time, and solve the recurrence to obtain $T(n)$.

   \[
   \begin{array}{ccccccccccccccc}
   19 & 23 & 29 & 31 & 37 & 41 & 43 & 47 & 53 & 59 & 2 & 3 & 5 & 7 & 11 & 13 & 17
   \end{array}
   \]

   Smallest ($A[\,], \, low, \, high$) {
      \hspace{1cm} // observe the similarity to binary search
      if (low==high) return $A[low]$;
      mid = (low+high)/2;
      if ($A[mid]<A[high]$) return Smallest ($A, \, low, \, mid$);
      else return Smallest ($A, \, mid+1, \, high$);
   }

   $T(n) = T(n/2) + 1$
   $T(n) = \Theta(lg \, n)$
3. First explain the details of how the $\Theta(n)$-time deterministic selection algorithm chooses the pivot value. Next suppose that we use these same steps to choose the pivot value in the quicksort algorithm. Write a recurrence for the worst-case running time of this version of quicksort, and solve the recurrence to obtain $T(n)$.

To choose the pivot value:
Partition the $n$ elements into $n/5$ groups with 5 elements per group.
Find the median element of each group.
Let $M$ denote the list of these $n/5$ medians.
Find the median of $M$ by calling select.
Let this median of $M$ be the pivot value.

$$T(n) \leq T(7n/10) + T(3n/10) + n \quad \text{[} T(n) \leq T(3n/4) + T(n/4) + n \text{ is acceptable too] }$$
$$T(n) = \Theta(n \log n)$$

4. Among Bin sort, Bubble sort, Counting sort, Heap sort, Insertion sort, Merge sort, Quick sort, Radix sort, Selection sort: Which algorithm performs best in each case? (Justify your answers.)

a. The input array is already sorted.

   Insertion sort or Bubble sort, $\Theta(n)$ time.

b. The input array is in random order, and average case time is most important.

   Quick sort, $\Theta(n \log n)$ average time. Can possibly also justify Heap sort or Merge sort.

c. Suppose that exchanges of the items in the array are very, very expensive. In other words, which method does the fewest “swaps” of the elements of the array?

   Selection sort, $\Theta(n)$ swaps.

d. The worst case running time is the most important such as for a real time system.

   Heapsort or Merge sort, $\Theta(n \log n)$ time.

e. The input array consists of integers in the range 1...$n^k$.

   Radix sort with radix $n$, $\Theta(nk)$ time.

f. The input array consists of bits (0s and 1s).

   Counting sort or Bin sort, $\Theta(n)$ time.
5. Give an $O(N)$ time algorithm to find the $\sqrt{N}$ values that are the closest (numerically) to the median value. Assume that you are given an unordered list $A$ of length $n$.

To simplify the presentation, here we assume there are no duplicate values:

// Solution 1:
median = Select (A, (N+1)/2);
high = Select (A, (N+1)/2 + $\sqrt{N}$);
low = Select (A, (N+1)/2 – $\sqrt{N}$);
array $B[2\sqrt{N}+1]$, $D[2\sqrt{N}+1]$;
$B$ = the $(2\sqrt{N} + 1)$ elements of $A$ in the range from low to high;
for $k$ = 0 to $2\sqrt{N}$
    $D[k] = |B[k] – median|$;
sort $B$ using $D$ values as keys;
return the first $\sqrt{N}$ values in $B$;

// Solution 2:
median = Select (A, (N+1)/2);
array $D[N]$, $B[\sqrt{N}]$;
for $k$ = 0 to $N – 1$
    $D[k] = |A[k] – median|$;
diff = Select ($D$, $\sqrt{N}$);
$B$ = the $\sqrt{N}$ elements of $A$ in the range from (median – diff) to (median + diff);
return $B$;

6. Some sorting algorithms, such as heap sort and in-place quick sort, are not usually stable. Explain a general method that will make any sorting algorithm stable.

let $A$ = input array of size $N$;
array $B[N]$;
for $k$ = 0 to $N – 1$
    $B[k] = pair (A[k], k)$;
Sort ($B$); // stability does not matter here because $B$ contains no duplicate pairs
for $k$ = 0 to $N – 1$
    $A[k] = B[k].first$;