Homework 3, Solution set

1. a. The recursion tree has the following structure.

size of subproblems	number of subproblems	non-recursive cost
$n = 2^{2^k}$	1	n
$2^{2^{k-1}}$	$4\sqrt{n} = 4n/2^{2^{k-1}}$	$2^{2^{k-1}} \cdot 4n/2^{2^{k-1}} = 4n$
$2^{2^{k-2}}$	$4^2 \cdot n/2^{2^{k-2}}$	$2^{2^{k-2}} \cdot 4^2 \cdot n/2^{2^{k-2}} = 4^2 n$
	...	
2^{2^1}	$4^{k-1} \cdot n/2^{2^1}$	$2^{2^1} \cdot 4^{k-1} \cdot n/2^{2^1} = 4^{k-1} n$
$2 = 2^{2^0}$	$4^k \cdot n/2^{2^0} = 4^k \cdot n/2$	$4^k \cdot n/2$

So the total cost is $n + 4n + \ldots 4^{k-1}n + 4^k n/2 = \frac{n}{3}(4^k - 1) + \frac{n}{2}4^k = \frac{5}{6}n(\log n)^2 - \frac{n}{3}$. Check: When $n = 2$, this gives $T(2) = 1$; when $n = 4$, this gives $T(4) = 12$. Both are correct.

b. The recursion tree has the following structure.

size of subproblems	number of subproblems	non-recursive cost
$n = 2^k$	1	$\log n$
2^{k-1}	1	$\log n/2 = (\log n) - 1$
2^{k-2}	1	$(\log n) - 2$
	...	
2^1	1	$\log 2 = 1$
$2^0 = 1$	1	1

So the total cost is $1 + 1 + 2 + 3 + \ldots + \log n = \frac{1}{2}\log n(1 + \log n) + 1$. Check: When $n = 1$, this gives $T(1) = 1$; when $n = 2$, this gives $T(2) = 2$. Both are correct.

2. Note that if A is in sorted order then this is readily done by determining for each i, $1 \leq i \leq n$, a leading finger f_i such that $A[f_i]$ is the largest point within distance d of $A[i]$ (possibly $f_i = i$). Then the total number of pairs of points within distance d of each other is exactly $\sum_{i=1}^{n}(f_i - i)$.

The following pseudo-code carries out this computation, returning the number of pairs of such points in *num_pr*.

NumPairs(A, n, num_pr):
$f = 1$; $num\_pr \leftarrow 0$;
for $1 = 1$ **to** n **do**
  while $f < n$ **and** $A[f+1] - A[i] \leq d$ **do** $f \leftarrow f + 1$
  end while
  $num\_pr \leftarrow num\_pr + (f - i)$
end for

Observe that since all updates to f are increments, this can happen only n times. For each iteration of the **for** loop, the **while** loop tests its condition one more time than the number of increments to f for that iteration of the **for** loop. Therefore the **while** loop iterates $2n$ times in total. Therefore NumPairs runs in time $O(n)$.

To obtain A in sorted order, we begin by sorting A using Merge Sort, which takes $O(n \log n)$ time. Thus the overall runtime is $O(n \log n)$.

One could also do a finger-style calculation while doing the Merge operation in MergeSort, but this is a bit more complicated. One way to do this maintains two leading fingers, one in each of the subarrays being merged.

3. The algorithm first runs the ApproxSelect procedure on its input. If the item $A[i]$ is the r-th smallest item at this point, i.e. if $i - d + 1 = r$, $A[i]$ is returned. Similarly if item $A[k]$ is the r-th smallest item, i.e. if $k - d + 1 = r$, $A[k]$ is returned. Otherwise, the r-th smallest item lies strictly between $A[i]$ and $A[k]$, and consequently a recursive search is performed on $A[i + 1 : k - 1]$. Among these items, we are seeking the $r - (i - d + 1)$-th smallest item, as the $i - d + 1$ smallest items at the left end of A have been removed. Pseudo code follows.

```
Find(A, d, u, r)
    ApproxSelect(A, d, u, r, i, k);
    if r = i − d + 1 then return (A[i]);
    else if r = k − d + 1 then return (A[k]);
    else Find(A, i + 1, k − 1, r − (i − d + 1))
end (* Find *)
```

Notice that there is no need for a separate base case, for when $d = u$, we are guaranteed that $r = 1$, and ApproxSelect(A, d, u, r, i, k) will return the values $i = k = d$. So then the first**if** leads to the return of value $A[d]$.

By assumption the top level call to ApproxSelect takes $O(n)$ time. The two tests take $O(1)$ time. Finally, the recursive call, if it is made, is to a problem of size at most $n/2$. Let $T(n)$ denote the worst case running time for problems of size at most n. Then:

$$T(n) \leq cn + T(n/2) \quad n > 1$$
$$T(1) \leq c$$

for some constant c.

The top node in recursion tree has a non-recursive cost of cn. Its one child node has a non-recursive cost of $cn/2$. The node at depth i has a cost of $cn/2^i$, and the leaf node has a cost of c. This totals $cn(1 + 1/2 + \ldots + 1/2^i + \ldots 1/2^{\log n}) < 2cn$.

Consequently this algorithm runs in $O(n)$ time.

4. Let $0 \le i < n/2$.

$$\begin{aligned} b_i = \sum_{j=0}^{n-1} x^{ij} a_j &= \sum_{j=0,\ j \text{ even}}^{n-1} x^{ij} a_j + \sum_{j=0,\ j \text{ odd}}^{n-1} x^{ij} a_j \\ &= \sum_{k=0}^{n/2-1} x^{i\cdot 2k} a_{2k} + \sum_{k=0}^{n/2-1} x^{i(2k+1)} a_{2k+1} \\ &= \sum_{k=0}^{n/2-1} (x^2)^{ik} a_k^e + x^i \sum_{k=0}^{n/2-1} (x^2)^{ik} a_k^o \\ &= [V_{n/2}(x^2)\boldsymbol{a}^e]_i + x^i [V_{n/2}(x^2)\boldsymbol{a}^o]_i. \end{aligned}$$

Similarly, recalling $x^{nk} = (x^n)^k = 1$,

$$\begin{aligned} b_{n/2+i} = \sum_{j=0}^{n-1} x^{(n/2+i)j} a_j &= \sum_{k=0}^{n/2-1} x^{(n/2+i)2k} a_{2k} + \sum_{k=0}^{n/2-1} x^{(n/2+i)\cdot(2k+1)} a_{2k+1} \\ &= \sum_{k=0}^{n/2-1} x^{nk} (x^2)^{ik} a_k^e + \sum_{k=0}^{n/2-1} x^{nk} x^{n/2} x^i (x^2)^{ik} a_k^o \\ &= \sum_{k=0}^{n/2-1} (x^2)^{ik} a_k^e + x^{n/2+i} \sum_{k=0}^{n/2-1} (x^2)^{ik} a_k^o \\ &= [V_{n/2}(x^2)\boldsymbol{a}^e]_i + x^{n/2} x^i [V_{n/2}(x^2)\boldsymbol{a}^o]_i. \end{aligned}$$

This leads to the following recursive algorithm to compute $V_n(x)\boldsymbol{a}$. If $n = 1$ then set $b_0 = a_0$. Otherwise, recursively compute $V_{n/2}(x^2)\boldsymbol{a}^e$ and $V_{n/2}(x^2)(\boldsymbol{a}^o)$, compute all powers x^k, for $0 \le k \le n-1$ (where we obtain x^{k+1} in $O(1)$ time by computing the product of the already obtained x^k with x), and determine $\boldsymbol{b}$ using the formula from the above equations.

Aside the recursive calls, this requires $O(n)$ time. Consequently, the running time is given by the following recurrence equation, for a suitable constant $c > 0$.

$$\begin{aligned} T(n) &\le 2T(n/2) + cn \quad n \ge 2 \\ T(1) &\le c \end{aligned}$$

and this has solution $T(n) = O(n \log n)$.