**Problem 1 [Jason Pazis]**

We will make the inductive hypothesis (we actually used the Master theorem case 1) that $T(m) = \Theta(m^{\log_2 3})$ for all $m < n$.

In order to prove our claim we will first show that $T(n) = \Omega(n^{\log_2 3})$ by assuming that $T(m) \geq m^{\log_2 3}$ for all $m < n$:

$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &\geq 3\frac{n^{\log_2 3}}{2^{\log_2 3}} + n \\ &= 3\frac{n^{\log_2 3}}{3} + n \\ &= n^{\log_2 3} + n \\ &\geq n^{\log_2 3} \end{aligned}$$

which proves that $T(n) = \Omega(n^{\log_2 3})$. Now we will show that $T(n) = O(n^{\log_2 3})$ by assuming that $T(m) \leq m^{\log_2 3} - 2m$ for all $m < n$:

$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &\leq 3\left(\frac{n^{\log_2 3}}{2^{\log_2 3}} - 2\frac{n}{2}\right) + n \\ &= 3\frac{n^{\log_2 3}}{3} - 3n + n \\ &= n^{\log_2 3} - 2n \\ &\leq n^{\log_2 3} - 2n \end{aligned}$$

which proves that $T(n) = O(n^{\log_2 3})$ and thus we have proven that $T(n) = \Theta(n^{\log_2 3})$.

We can see by simple inspection that for every 2 "input units" the cost is quadrupled, or said differently for every one "input unit" the cost doubles. Thus it is natural to make the hypothesis $T(n) = \Theta(2^n)$. We will now prove our claim by induction, first for $\Omega$ by assuming that $T(m) \geq 2^m$ for all $m < n$:

$$\begin{aligned} T(n) &= 4T(n-2) + 2 \\ &\geq 4 \times 2^{n-2} + 2 \\ &= 2^2 \times 2^{n-2} + 2 \\ &= 2^n + 2 \\ &\geq 2^n \end{aligned}$$

which proves that $T(n) = \Omega(2^n)$. Assuming $T(m) \leq 2^m - 1$ for all $m < n$ we also have:

$$\begin{aligned} T(n) &= 4T(n-2) + 2 \\ &\leq 4 \times (2^{n-2} - 1) + 2 \\ &= 2^2 \times 2^{n-2} - 2 \\ &= 2^n - 2 \\ &\leq 2^n - 1 \end{aligned}$$

which proves that $T(n) = O(2^n)$ and thus we have that $T(n) = \Theta(2^n)$.

**Problem 2** [**Jason Pazis**]

We will begin by expanding the recurrence:

$$\begin{aligned} T(n) &= 2T(n/2) + n\log n \\ &= 4T(n/4) + n\log\frac{n}{2} + n\log n \\ &= 8T(n/8) + n\log\frac{n}{4} + n\log\frac{n}{2} + n\log n \end{aligned}$$

It is easy now to see that we have $\log n$ levels costing $n\log(n-i)$ each:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n} n\log(n-i) \\ &= n\sum_{i=0}^{\log n} \log(n-i) \\ &= \Theta(n\log^2 n) \end{aligned}$$

Once again we will begin by expanding the recurrence:

$$\begin{aligned} T(n) &= 2T(n/2) + n/\log n \\ &= 4T(n/4) + 2\frac{n/2}{\log(n/2)} + n/\log n \\ &= 4T(n/4) + \frac{n}{\log(n-1)} + n/\log n \end{aligned}$$

It is easy to see that the recursion will have $\log n$ levels where each one will cost $\frac{n}{\log(n-i)}$. Thus the sum will be:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n} \frac{n}{\log(n-i)} \\ &= n \sum_{i=0}^{\log n} \frac{1}{\log(n-i)} \\ &= n \sum_{i=1}^{\log n} \frac{1}{i} \\ &= \Theta(n \log \log n) \end{aligned}$$

If we try to visualize the recurrence we notice that as we progress the new terms generated vanish very quickly. Therefore we will make the hypothesis that $T(n) = \Theta(n \log n)$.

In order to prove our claim we will first show that $T(n) = \Omega(n \log n)$ by assuming that $T(m) \geq m \log m$ for all $m < n$:

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \log n \\ &\geq \sqrt{n} \sqrt{n} \log \sqrt{n} + n \log n \\ &= n \log \sqrt{n} + n \log n \\ &\geq n \log n \end{aligned}$$

which proves that $T(n) = \Omega(n \log n)$. Now we will show that $T(n) = O(n \log n)$ by assuming that $T(m) \leq 2m \log m$ for all $m < n$:

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \log n \\ &\leq \sqrt{n} 2\sqrt{n} \log \sqrt{n} + n \log n \\ &= n \log n + n \log n \\ &\leq 2n \log n \end{aligned}$$

which proves that $T(n) = O(n \log n)$ and thus we have proven that $T(n) = \Theta(n \log n)$.

**Problem 3** [**Matt Matlock**]

Let $I$ be a set of $n$ intervals of the form $[a_1, b_1], ..., [a_n, b_n]$ on the real line. Consider the following algorithm:

```
GREEDY(I):

(1)   l ← (min{a_1, a_2, a_3, ..., a_n} − 1)
(2)   p ← {}
(3)   SORT I by finish points.
(4)   FOR i = 1 to n
(5)       IF a_i > l
(6)           APPEND(p, b_i)
(7)           l ← b_i
(8)       END IF
(9)   END FOR
```

In this algorithm, in step 1, we set $l$ to a number before any start point of any interval, then make a $p$ and empty list to hold our points. Next, we sort our intervals by their *finish point*. Then for each interval (in sorted order). (5) We check to see if the current start point is after the last point we added. If it is, then we need a new point, so (6) adds the point and (7) sets our new last point $l$ to $b_i$. Finally we return $p$.

The running time of this algorithm is, explicitly $n \log n + 2n$. That is, $n \log n$ to sort, $n$ to find the minimum start time, and $n$ to run the for loop. This yields a running time of $\mathcal{O}(n \log n)$.

The algorithm is also correct.

To see this, consider any optimal solution $\mathcal{O}$. We will show that this solution cannot be any better than ours. First, order the points of this solution from left to right, and consider a solution from our algorithm $\mathcal{A}$. Take the first point of $\mathcal{O}$. This point cannot occur after the first interval endpoint in our left to right ordering, because otherwise $\mathcal{O}$ would not cover this interval. In $\mathcal{A}$ our first point occurs exactly on this endpoint. Now, removing this first point from both $\mathcal{O}$ and $\mathcal{A}$, and removing all intervals covered by these points in each solution respectively, we have now generated a subproblem of this covering problem to which $\mathcal{O}$ is still an optimal solution. We can iteratively apply this argument until we have finished, thus demonstrating that we must have covered all of the intervals in at MOST the number of points in the optimal solution $\mathcal{O}$.

**Problem 4 [Jason Pazis]**

The first step in our algorithm will be to find all possible palindromes. We will be able to do that in $O(n^2)$ time (where $n$ is the length of our string S) by incrementally filling the lower half of a $2D$ matrix P with $n \times n$ entries. A 1 in the $i$th column and $j$th row will denote that there exists a palindrome between the $i$th and $j$th letters of our input string. Initially we will initialize our matrix to all zeros. We will start working from the primary diagonal to find palindromes of odd length, and from the diagonal just below that for palindromes of even length. At each step we will subtract 1 from $i$ and add 1 to $j$ moving towards the lower right corner of the matrix and set $P(i, j)$ to 1 if $S(i) = S(j)$. If at some point $S(i) \neq S(j)$ we will terminate the current traversal since at

this point it is clear we will not be able to find more palindromes that have the current substring in their middle. That way we are able to decide on whether a string from $i$ to $j$ is a palindrome incrementally, basing our decision on whether the string from $i+1$ to $j-1$ is a palindrome and whether $S(i) = S(j)$. More formally our algorithm is stated:

```
Initialize P to 0's
For i=k to n
  i=k, j=k
  while S(i) == S(j)
    P(i,j) = 1
    i--, j++
  endWhile
endFor
For i=k to n-1
  i=k, j=k+1
  while S(i) == S(j)
    P(i,j) = 1
    i--, j++
  endWhile
endFor
```

We will now use matrix P and the principle of Dynamic Programing to find the minimum number of palindromes in $O(n^2)$ time.

If $c_i, \ldots, c_n$ is the last palindrome included in the optimal solution the cost of the optimal solution is (note that each palindrome incurs a cost of 1 and we define $OPT(0) = 0$):

$$OPT(n) = 1 + OPT(i-1)$$

It is now straightforward to define the recurrence

$$OPT(j) = \min_{1 \leq i \leq j} (1 + OPT(i-1)) \, for \, i, j | P(i,j) = 1$$

and this way the substring $s_i, \ldots, s_n$ is used in an optimum solution for the subproblem, if and only if the minimum is obtained using index $i$ and $s_i, \ldots, s_n$ constitutes a valid palindrome.

Now we can build up the solutions $OPT(i)$ in order of increasing $i$.

Palindromes(n)
  $OPT(0) = 0$
  For $j = 1$ to $n$
    $OPT(j)$ = $\min_{1 \leq i \leq j} (1 + OPT(i-1)) \, for \, i, j | P(i,j) = 1$
  endFor
  return $OPT(n)$

The for loop will run n times with each iteration taking at most n steps each step requiring $O(1)$ time for a total running time of $O(n^2)$. In order to return the sequence of palindromes we can use the following $O(n^2)$ procedure for $j = n$ tracing back through the array $OPT$ to compute an optimum partition into palindromes:

Find-Palindromes(j)
  If $j = 0$ then
    Output nothing
  Else
    Find an i that minimizes $1 + OPT(i-1)$ under the constraint P(i,j)=1
    Output the palindrome $(s_i, \ldots, s_j)$ and the result of
      Find-Palindromes(i-1)
  endIf

Of course in order to be more efficient we can cache the results of the $\min$ operator on each iteration of the Palindromes algorithm and use them to save time. In any case since all three steps (finding all palindromes, finding the cost of the optimal solution, and outputting the optimal solution) of our algorithm require $O(n^2)$ time, the total complexity will be $O(n^2)$.

**Problem 5 [Alan Guo]**

The probability that a pair of elements is swapped is $\frac{1}{2}$. Indeed, suppose we have a list of elements $a_1, \ldots, a_n$. For any pair $(a_i, a_j)$, there is a bijection between the permutations in which $a_i$ and $a_j$ are inverted and the permutations in which they are not (the bijection is induced by the reversing permutation). Hence half of the permutations have $a_i, a_j$ inverted and the other half do not.

The number of swaps Bubble Sort makes is equal to the number of inverted pairs in the list, since each adjacent swap made reduces the number of inverted pairs by 1. We claim the expected number of inverted pairs in a list of $n$ elements is $\frac{n^2-n}{4}$. There are $\binom{n}{2}$ possible pairs given $n$ elements. Suppose in a certain permutation, $k$ of the pairs are inverted. Then there exists a permutation, namely the reverse one, which contains $\binom{n}{2} - k$ inverted pairs. Hence we may group the $n!$ distinct permutations into pairs with $\binom{n}{2}$ inverted pairs between them. Therefore the expected number of inverted pairs, and hence the expected number of swaps Bubble Sort makes, is equal to $\frac{1}{2}\binom{n}{2} = \frac{n^2-n}{4}$.