Solution of First Homework

Han Wu, Yu Chen

1. **COVID-19 Risk Detection.** (12 points) Each of n customers spends some time in a cafe shop. For each $i = 1, \ldots, n$, user i enters the shop at time a_i and leaves at time $b_i \geq a_i$. You are interested in the question: how many distinct pairs of customers are ever in the shop at the same time? (Here, the pair (i, j) is the same as the pair (j, i)). Example: Suppose there are 5 customers with the following entering and leaving times:

Customer	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

Then, the number of distinct pairs of customers who are in the shop at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$.

(a) (3 pts) Suppose a customer c_i is reported as a suspected case, can you give an algorithm to retrieve all the potential inflicted customers in $O(n)$-time?
Answer: We can enumerate every customers except c_i and check whether their time intervals are overlapped. Algorithm 1 shows the details.

Algorithm 1 inflictedCheck

```
1: procedure INFLICTEDCHECK(List C = [c_1, ..., c_n] of n customers, costomer c_i who is reported as a suspected case)   ▷ Find potential inflicted customers in C
2:     cnt ← 0
3:     for each c_j ∈ C do
4:         if c_i is equal to c_j || c_j.a > c_i.b || c_j.b < c_i.a then   ▷ a: enterTime, b: leavTime
5:             continue   ▷ c_i and c_j is equal or not overlap
6:         cnt++
7:     return cnt
```

(b) (4 pts) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above, there is a straightforward algorithm that takes about[1] n^2 time to compute the number of pairs of customers who are ever in the shop at the same time. Give this algorithm and explain why it takes time about n^2.
Answer: We can do this task by bruteforce. Details are shown in Algorithm 2.

[1]Formally, "about" here means $\Theta(n^2)$, but you can be informal about this.

Algorithm 2 findOverlappedInterval(BruteForce)

```
1: procedure FINDOVERLAPPEDINTERVAL(List C = [c_1, ..., c_n] of n customers)   ▷ Find all overlapped intervals by bruteforce
2:     cnt ← 0
3:     for each c_i ∈ C do
4:         for each c_j ∈ C do
5:             if c_i is equal to c_j || c_j.a > c_i.b || c_i.b > c_j.a then   ▷ a: enterTime, b: leavTime
6:                 continue                                                   ▷ c_i and c_j is equal or not overlap
7:             cnt++
8:     return cnt
```

(c) (5 pts) Give an $O(n\log(n))$-time algorithm to do the same task and analyze its running time. (**Hint:** consider sorting relevant events by time).
Answer: We can sort the intervals by their start times, then we have to methods to count the number of pairs which are overlapped.

1) use binary search to find the smallest index j where interval c_j is not overlapped with current interval $c_i(i < j)$ in $O(log(n))$ time. Algorithm 3 shows the details. Time Complexity: $O(n\log(n))$.

2) count the number of customers who are in the cafe shop at a certain time interval. Algorithm 4 shows the details. Time Complexity: $O(n)$.

Algorithm 3 findOverlappedInterval(Sorting+BinarySearch)

```
1: procedure FINDOVERLAPPEDINTERVAL(List C = [c_1, ..., c_n] of n customers)   ▷ Find all overlapped intervals by a cleverer method
2:     Sort each c_i ∈ C by its start time
3:     cnt ← 0
4:     for each c_i ∈ C do
5:         j ← binarySearch(C, i, c_i.b)
6:         cnt+ = j − i − 1
7:     return cnt
8: procedure BINARYSEARCH(C, i, target)   ▷ Find minimum index in C[i + 1 :] where the interval's enterTime is greater than target
9:     l ← i + 1, r ← C.length
10:    while l < r do
11:        mid ← (l + r)/2
12:        if c_mid.a ≤ target then
13:            l = mid + 1
14:        else
15:            r = mid
16:    return l
```

Algorithm 4 findOverlappedInterval(Sorting+Counting)

```
1:  procedure FINDOVERLAPPEDINTERVAL(List C = [c_1, ..., c_n] of n customers)   ▷ Find all overlapped intervals by a cleverer method
2:      Sort each c_i ∈ C by its start time
3:      List S = [s_1, ..., s_n] ← C                    ▷ A list of start times of each interval
4:      List E = [e_1, ..., e_n] ← C                    ▷ A list of end times of each interval
5:      cnt ← 0
6:      cur ← 0
7:      curEnd ← 0
8:      j ← 1
9:      for i = 1 to n do
10:         while s_i > e_j do
11:             cnt+ = cur − 1
12:             cur − −
13:             j + +
14:         cur + +
15:     while j ≤ n do
16:         cnt+ = cur − 1
17:         cur − −
18:         j + +
19:     return cnt
```

2. **Proof of correctness.** (6 points) Consider the following Selection Sort algorithm that is supposed to sort an array of integers. Provide a proof that this algorithm is correct. (**Hint**: you may need to use more than one loop invariant.)

```
# Sorts an array of integers.
Sort(array A):
    for i = 1 to A.length:
        minIndex = i
        for j = i + 1 to A.length:
            if A[j] < A[minIndex]:
                minIndex = j
        Swap(A[i], A[minIndex])

# Swaps two elements of the array.  You may assume this function is correct.
Swap(array A, int x, int y):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp
```

Answer: For convenience, we add each line of the selection sort algorithm a row number, as shown in Algorithm 5.
Then, we will use two loop invariants to prove that selection sort is correct.

- First loop invariant: between lines 4 and 5, $A[minIndex]$ is the smallest element among $A[i : j)$.
  - To prove this loop invariant, we need to prove following two things.
  - *Initialization*: the loop invariant is true at the beginning of the loop. The beginning of the loop is when $j = i+1$ and $minIndex = i$, so the loop invariant states that $A[i]$ is the smallest element among $A[i : (i + 1)) = A[i]$ which is true.
  - *Maintenance*: the body of the loop (including the increment of the loop index j) preserves

Algorithm 5 SelectionSort

```
1: procedure SORT(A)
2:     for i = 1 to A.length do
3:         minIndex = 1
4:         for j = i + 1 to A.length do
5:             if A[j] < A[minIndex] then
6:                 minIndex = j
7:         Swap(A[i], A[minIndex])
8:     return cnt
```

the invariant. We know that between lines 4 and 5, $A[minIndex]$ is the smallest among $A[i : j)$. The body of the loop checks if $A[j]$ is smaller than $A[minIndex]$ and if so sets $minIndex$ to j, so between lines 5 and 6 we know that $A[minIndex]$ is the smallest among $A[i : (j + 1))$. When j is incremented, the loop rolls around, the loop invariant will still be true between lines 4 and 5.

- Therefore, we can conclude that the invariant is true after the loop is over, which happens when $j = A.length + 1$. In this case, substituting $j = A.length + 1$ into the loop invariant, we know that on line 7 $A[minIndex]$ is the smallest among $A[i : A.legnth + 1)$.

- Second loop invariant: between lines 2 and 3, the entries $A[1 : i)$ are the top-$(i - 1)$ smallest elements of A in sorted order.
  - To prove this loop invariant, we need to prove following two things.
  - *Initialization*: the loop invariant is true at the beginning of the loop. The beginning of the loop is when $i = 1$ so the invariant states that $A[1 : 1)$ are the top-0 smallest elements in sorted order. This is true since the slice is empty.
  - *Maintenance*: By the previous argument, we know that after the inner loop, $A[minIndex]$ is the smallest among $A[i :]$. By swapping it with $A[i]$, we now know that after line 7, $A[1 : i]$ are the top-i smallest elements in sorted order. When the loop rolls around, the loop invariant will therefore be maintained.
  - Therefore, we can conclude that the invariant is true after the loop is over, which happens when $i = A.length + 1$. By substituting $i = A.length + 1$ into the loop invariant, we know that $A[1 : A.length + 1)$ are the top-$A.length$ smallest elements in sorted order, i.e. the list A is sorted.

3. **Needlessly complicating the issue.** (12 points)

   (a) (3pts) Give a linear-time (that is, an $O(n)$-time) algorithm for finding the minimum of n values (which are not necessarily sorted).
   Answer: Details are shown in Algorithm 6.

Algorithm 6 findMinimum

```
1: procedure FINDMINIMUM(List A = [a_1, ..., a_n] of n items)
2:     min ← −1                          ▷ Initialize a variable to store the minimum value
3:     for a ∈ A do
4:         min = Min(a, min)             ▷ Update minimum value
5:     return min
```

   (b) (3pts) Argue that any algorithm that finds the minimum of n items must do at least n operations in the worst case.
   Answer: To see this, we will use a proof by contradiction. Suppose that there exist an algorithm

F with only $k(< n)$ operations works well on List A. Without loss of generality, we assume that these k operations are for a_1 to a_k. We create a new list A' which forms by the same items in A but we execute $swap(a_{MinValueIndex(A)}, a_n)$. So that $F(A') > a_n$. This is a contradiction.

Now consider the following recursive algorithm to find the minimum of a set of n items.

Algorithm 7 `findMinimum`

```
1: procedure FINDMINIMUM(List A = [a_1, ..., a_n] of n items)
2:     if n=1 then
3:         return _____
4:     A_1 = A[0 : n/2]
5:     A_2 = A[n/2 : n]
6:     return min(findMinimum(A_1), findMinimum(A_2) )
```

(c) (3pts) Fill in the blank in the pseudo-code: what should the algorithm return in the base case? Briefly argue that the algorithm is correct with your choice.
Answer: return $A[1]$. To see this, we'll use a proof by *Loop Invariant*.

Proof. **Loop Invariant**: At the end of the *findMinimum*(A), it returns the minimum value in A.
Initialization: When the size k of the list is 1, the function *findMinimum*(A) returns the first element $A[1]$ of the list and it's also the smallest element in list A. So the *Loop Invariant* holds.
Maintenance: Assume that the loop invariant holds at the recursive calls in function *findMinimum*(A). So that the function *findMinimum*$(A[0, n/2])$ returns the minimum value among the first $n/2$ elements, and the function *findMinimum*$(A[n/2, n])$ returns the minimum value among the last $n/2$ elements. At the last line of the function *findMinimum*(A), it returns the smaller of two numbers. So the return value of the function *findMinimum*(A) satisfies the loop invariant. In other words, the return value is the minimum value of the list A.
Termination: The function terminates when the input list size is 1.
Now the loop invariant gives: The return value of the function *findMinimum*(A) holds the minimum value in A. Therefore the algorithm is correct. □

(d) (3pts) Analyze the running time of this recursive algorithm. How does it compare to your solution in part (a)?
Answer: $T(n) = 2T(n/2) + O(1)$. We apply the master theorem with $a = b = 2$ and with $d = 0$. We have $a > b^d$, so the running time is $O(n)$.

4. **Recursive local-minimum-finding.** (12 points)

(a) Suppose A is an array of n integers (for simplicity assume that all integers are distinct). A *local minimum* of A is an element that is smaller than all of its neighbors. For example, in the array $A = [1, 2, 0, 3]$, the local minima are $A[1] = 1$ and $A[3] = 0$.

i. (2 points) Design a recursive algorithm to find a local minimum of A, which runs in time $O(\log(n))$.
Answer: We can use binary search to find a local minimum. Details are shown in Algorithm 8.

Algorithm 8 findLocalMinimum

```
1: procedure FINDLOCALMINIMUM(List A = [a_1, ..., a_n] of n numbers)   ▷ Find local minimum in A
2:     Append ∞ to the head and tail of A, now A's length is n + 2
3:     return findLocalMinimum(A, 1, n)
4: procedure FINDLOCALMINIMUM(A, i, j)                                 ▷ Find local minimum in A[i : j]
5:     if i == j then
6:         return A[i]                                                 ▷ Base case
7:     mid ← (i + j)/2
8:     if A[mid − 1] ≥ A[mid] && A[mid] ≤ A[mid + 1] then
9:         return A[mid]                                               ▷ Element in mid is a local minimum
10:    else if A[mid − 1] ≥ A[mid] then
11:        return findLocalMinimum(A, mid + 1, j)                      ▷ Search on the right handside
12:    else
13:        return findLocalMinimum(A, i, mid − 1)                      ▷ Search on the left handside
```

ii. (2 points) Prove formally that your algorithm is correct.
Answer: Recursion invariant: there must be a local minimum in $A[L : R]$.

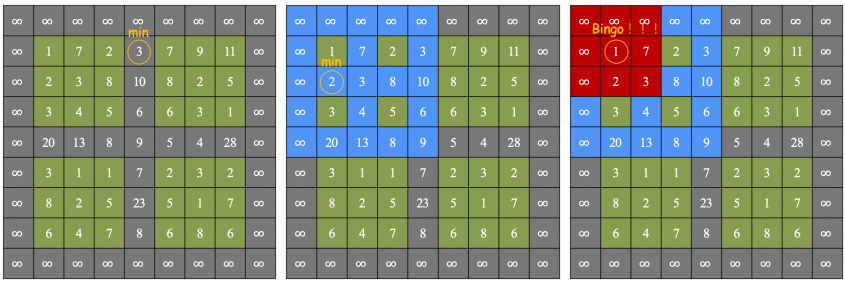
- To prove this recursion invariant, we need to prove following two things.
- *Initialization*: the recursion invariant is true at the beginning of the recursion. The beginning of the recursion is when $L = 1$ and $R = n$, so the recursion invariant states that there must be a local minimum in $A[1 : n]$ which is true.
- *Maintenance*: Each recursion preserves the invariant. Between lines 8 and 9, if $A[mid]$ is the local minimum, then we find the answer. If $A[mid]$ is not a local minimum, there must be a neighbor that is smaller than it. Thus, the local minimum must exist in the half range which contains the smaller neighbor. Thus, the invariant is preserved.
- Therefore, we can conclude that the invariant is true after the recursion is over, which happens when $L = R$ or $A[mid]$ is the local minimum. If $L = R$, then $A[L : R] = A[L]$. Obviously, local minimum of an array with one element is the element itselt.

iii. (2 points) Formally analyze the runtime of your algorithm.
Answer: $T(n) = T(\frac{n}{2}) + O(1)$. Apply master theorem with $a = 1$, $b = 2$ and $d = 0$. Because $a = b^d$, the time complexity is $O(n^d log(n)) = O(log(n))$.

(b) Let G be a square $n \times n$ grid of integers. A *local minimum* of A is an element that is smaller than all of its neighbors (diagonals do not count). For example, in the grid

$$G = \begin{bmatrix} 5 & 6 & 3 \\ 6 & 1 & 4 \\ 3 & 2 & 3 \end{bmatrix}$$

some of the local minima are G[1][1]=5 G[2][2] = 1.



(a) First Reccusion (b) Second Reccusion (c) Last Reccusion

Figure 1: An example to illustrate the algorithm 9.

i. (2 points) Design a recursive algorithm to find a local minimum in $O(n)$ time (you can assume that all integers are distinct).
Answer: We can use divide and conquer to find a local minimum in an $n \times n$ matrix. Algorithm 9 roughly shows the procedure. And to make it more intuitive, we use an example to illustrate the algorithm. As show in Figure 1(a), in the first iteration, we find the minimum number of the gray area. We find that 3 is the minimum. However, it's not a local minimum, because its neighbor 2 is less than it. So we continue to search on the left-top quadrant, which is marked by blue in Figure 1(b). And we find that the minimum number is 2. However, it's still not a local minimum because its neighbor 1 is less than it. So we continous to search on the left-top quadrant, which is marked by red in Figure 1(c). And finally we get the answer.
There are two things need to pay attention to:

- When decreasing the scale of the problem, we need to include the boundaries, e.g. from Figure Figure 1(a) to 1(b), we need to include the gray boundary.
- We cannot just find the middle element of the matrix and then determine the area of the subproblem because it's insufficient to determine which quadrant we can continue to search in next recursion. For example, in Figure 1(a), we select 9 in the first recursion. But what quadrant shall we choose in the next recursion?

Algorithm 9 findLocalMinimum

```
1: procedure FINDLOCALMINIMUM(An n × n matrix G)        ▷ Find local minimum in G
2:     Append ∞ to the boundary of G, now G becomes an (n + 2) × (n + 2) matrix.
3:     Find the minimum element min among boundary and middle row and column.
4:     if min is local min then
5:         return min
6:     else
7:         Find the smaller neighbor and search in the cooresponding quadrant.
```

ii. (2 points) We are not looking for a formal correctness proof, but please explain why your algorithm is correct.
Answer: Recursion invariant: minimum element in the boundary and middle row and column never increase as we continue to next recursion. Thus, with the size of matrix decreases, we can finally find a local minimum.

iii. (2 points) Give a formal analysis of the running time of your algorithm.
Answer: $T(n) = T(\frac{n}{2}) + c \cdot n$. Apply master theorem with $a = 1$, $b = 2$ and $d = 1$. Because $a < b^d$, the time complexity is $O(n)$.

5. **Probability refresher** (4 points)

(a) (1 point) What is the cardinality of the set of all subsets of $\{1, 2, ..., n\}$? **[We are expecting a mathematical expression along with one or two sentences explaining why it is correct.]**
Answer: There are n items in the whole set, and each item can be chosen into a subset or not, so there are total of 2^n subsets.

(b) (1 point) Suppose we choose a subset of $\{1, ..., n\}$ uniformly at random: that is, every set has an equal probability of being chosen. Let X be a random variable denoting the cardinality (that is, the size) of a set randomly chosen in this way. Calculate the expected value of X and show your work. **[We are expecting a mathematically rigorous argument establishing your answer. Your solution should not include summation signs.]**
Answer: $Pr\{X = k\} = \frac{C_n^k}{2^n}$
$E(X) = \sum_{k=0}^{n} k \cdot Pr\{X = k\} = \sum_{k=0}^{n} k \cdot \frac{C_n^k}{2^n} = \frac{1}{2^n} \cdot \sum_{k=0}^{n} k \cdot C_n^k$
Because $k \cdot C_n^k = k \cdot \frac{n!}{k!(n-k)!} = \frac{n!}{(k-1)!(n-k)!} = n \cdot \frac{(n-1)!}{(k-1)!(n-k)!} = n \cdot C_{n-1}^{k-1}$
We can get $E(X) = \frac{n}{2^n} \cdot \sum_{k=1}^{n} C_{n-1}^{k-1} = \frac{n}{2^n} \cdot \sum_{k=0}^{n-1} C_{n-1}^{k} = n \cdot \frac{2^{n-1}}{2^n} = \frac{n}{2}$

(c) (2 points) Let $rand(a, b)$ return an integer uniformly at random from the range $[a, b]$. Each call to $rand(a, b)$ is independent. Consider the following function:

```
f(k,n):
    if k <= 1:
        return rand(1,n)
    return 2 * f(k/2, n)
```

What is the expected value and variance of $f(k, n)$? Assume that k is a power of 2. Please show your work. **[In addition to your answer, we are expecting a mathematical derivation along with a brief (1-2 sentence) analysis of what the pseudocode above does in order to explain why your derivation is the right thing to do.]**
Answer: Given the condition that $k = 2^c$, then $f(k, n)$ will return $2^c \cdot rand(1, n) = k \cdot rand(1, n)$. The expected value and variance of uniform ditribution $U(a, b)$ is $\frac{a+b}{2}$ and $\frac{(b-a)^2}{12}$ respectively. So the expected value and variance of $f(k, n) \sim U(k, kn)$ is $\frac{k+kn}{2} = k \cdot \frac{1+n}{2}$ and $\frac{(kn-k)^2}{12} = k^2 \cdot \frac{(n-1)^2}{12}$ respectively.

6. **Fun with Big-O notation.** (6 points; 1 point each) Mark the following as `True` or `False`. Briefly but convincingly justify all of your answers, using the definitions of $O(\cdot)$, $\Theta(\cdot)$ and $\Omega(\cdot)$. **[To see the level of detail we are expecting, the first question has been worked out for you.]**

(a) $n = O(n \log(n))$.
Answer: This statement is **True**. By the definition of O, there must exist constants n_0 and c such that for all $n > n_0$, $n \leq c * n \log n$. Therefore, $c \geq 1/\log n$. We set $c = 1, n_0 = 2$ such that $n \leq n \log n$ $(n \geq 2)$ as required.

(b) $n^{1/\log(n)} = \Theta(1)$.
Answer: This statement is **True**. By the definition of Θ, there exist positive constants c_1, c_2 and n_0 such that for all $n > n_0$, $0 \leq c1 \leq n^{\frac{1}{\log n}} \leq c_2$. Therefore, $c = 2$. We set $c_1 = c_2 = 2, n_0 = 2$ such that they satisfied the the definition of $\Theta(1)$.

(c) If

$$f(x) = \begin{cases} 5^n & if \quad n < 2^{1000} \\ 2^{1000} n^2 & if \quad n \geq 2^{1000} \end{cases}$$

and $g(n) = \frac{n^2}{2^{1000}}$, then $f(n) = O(g(n))$.
Answer: This statement is **True**. By the definition of O, there must exist constants n_0 and c such that for all $n > n_0$, $f(n) \leq c * g(n)$. We try to set $n_0 = 2^{1000}$, so we got $2^{1000} \leq \frac{c*n^2}{2^{1000}} \Rightarrow c \geq 2^{2000}$. Therefore, we set $n_0 = 2^{1000}$ and $c = 2^{2000}$ such that they satisfied the the definition of $O(g(n))$.

(d) For all possible functions $f(n), g(n) \geq 0$, if $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
Answer: This statement is **False**. Here, we give a counterexample. Let $f(n) = 2\log n$, $g(n) = \log n$, we know that $2\log n \leq c * \log n$ (Note that $c \geq 2$), therefore $f(n) = O(g(n))$. However, $2^{2\log n} = 2^{\log n^2} = n^2$, while $2^{g(n)} = 2^{\log n} = n$. We know that n^2 is not $O(n)$. Therefore, $2^{f(n)} \neq O(2^{g(n)})$.

(e) $5^{\log\log(n)} = O(log(n)^2)$.
Answer: This statement is **False**. To see this, we will use a proof by contradiction. Suppose that, as per the definition of $O()$, there must exist constants n_0 and $c > 0$ so that for all $n > n_0$, $5^{\log\log(n)} \leq c*\log(n)^2 \Rightarrow \log\log n*\log 5 \leq \log c+2\log\log n$. Choose $n = \max\{2^{2^{\log c/(\log 5-2)}}, n_0\}+1$. Then $n \geq n_0$, but we have $n > 2^{2^{\log c/(\log 5-2)}}$, which implies that $\log\log n*\log 5 > \log c+2\log\log n$. This is a contradiction.

(f) $n = \Theta(100^{\log(n)})$.
Answer: This statement is **False**. To see this, we will use a proof by contradiction. Suppose that, as per the definition of Θ, there exist positive constants c_1, c_2 and n_0 such that for all $n > n_0$, $0 \leq c_1 * 100^{\log n} \leq n \leq c_2 * 100^{\log n}$. We know that, $n^7 = (2^7)^{\log n} \leq (10^{10})^{\log n}$. Therefore, the definition requires that $n \geq c_1 * n^7 \Rightarrow c_1 \leq \frac{1}{n^6}$. Choose $n = \max\{c_1^{-1/6}, n_0\} + 1$. Then $n \geq n_0$, but we have $n > c_1^{-1/6}$, which implies that $c_1 > \frac{1}{n^6}$. This is a contradiction.

7. **Fun with recurrences.** (6 points; 1 point each) Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$, and give a short justification. Be aware that some parts might be slightly more involved than others. Unless otherwise stated, assume $T(1) = 1$. [**To see the level of detail expected, we have worked out the first one for you.**]

(a) $T(n) = 2T(n/2) + 3n$.
Answer: We apply the master theorem with $a = b = 2$ and with $d = 1$. We have $a = b^d$, and so the running time is $O(n\log n)$

(b) $T(n) = 3T(n/4) + \sqrt{n}$.
Answer: We apply the master theorem with $a = 3, b = 4$ and with $d = \frac{1}{2}$. We have $a > b^d$, and so the running time is $O(n^{\log_4^3})$

(c) $T(n) = 7T(n/2) + \Theta(n^3)$.
Answer: We apply the master theorem with $a = 7, b = 2$ and with $d = 3$. We have $a < b^d$, and so the running time is $O(n^3)$

(d) $T(n) = 4T(n/2) + n^2\log n$.
Answer: $O(n^2\log n)$

$$\begin{aligned}
T(n) &= 4T(n/2) + n^2\log n \\
&= 4(4T(n/4) + (n/2)^2\log(n/2)) + n^2\log n \\
&= 16T(n/4) + n^2\log(n/2) + n^2\log n \\
&= \dots \\
&= (4^k)T(n/(2^k)) + n^2(\log(n/k) + \dots + \log(n/4) + \log(n/2) + \log n) \\
&\quad (\ if\ n/(2^k) = 1 \Rightarrow n = 2^k \Rightarrow k = \log n\ and\ T(1) = 1\ ) \\
&= n^2T(1) + n^2log((n/(2^k)) * \dots * (n/(2^2)) * (n/(2^1)) * n) \\
&= n^2T(1) + n^2log((n/(2^{\log n})) * \dots * (n/(2^2)) * (n/(2^1)) * n) \\
&= n^2 + n^2(\log(2^{\log n}))\ (Using\ geometric\ series) \\
&= O(n^2\log n)
\end{aligned}$$

(e) $T(n) = 2T(n/3) + n^c$, where $c \geq 1$ is a constant (that is, it doesn't depend on n).
Answer: We apply the master theorem with $a = 2, b = 3$ and with $d = c \geq 1$. We have $a < b^d$, and so the running time is $O(n^c)$

(f) $T(n) = 2T(\sqrt{n}) + 1$, where $T(2) = 1$.
Answer: $O(2^{\log \log n})$

$$
\begin{aligned}
T(n) &= 2T(n^{1/2}) + 1 \\
&= 2(2T(n^{1/4}) + 1) + 1 \\
&= 2^2 T(n^{1/4}) + 2 + 1 \\
&= \dots \\
&= 2^k T(n^{\frac{1}{2^k}}) + 2^{k-1} + \cdots + 1 \\
&= 2^k T(n^{\frac{1}{2^k}}) + 2^k - 1 \\
&\quad (n^{\frac{1}{2^k}} = 2 \Rightarrow k = \log \log n \ and \ T(2) = 1) \\
&= 2^{\log \log n} T(2) + 2^{\log \log n} - 1 \\
&= O(2^{\log \log n})
\end{aligned}
$$