

Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, as well as correctness. For the data structure question, the efficiency of your solution will also be taken into account.

Analyzing loops-10pts Consider the following iterative algorithm, that uses an $\Theta(I)$ time procedure $proc(I)$, which does not change I .

Algorithm(n : positive integer);

1. begin;
2. $I \leftarrow 1$;
3. While $I \leq n$ do:
4. begin;{while}
5. FOR $J = 1$ TO $\lfloor n/I \rfloor$ do $proc(I)$
6. $I \leftarrow 2 * I$.
7. end;{while}
8. end;

Give a time analysis, up to Θ , for this algorithm.

Let I_t be the value of I after t iterations of the While loop. Since $proc(I)$ doesn't change I , I doubles every loop, so $I_t = 2^t$. So the while loop terminates when $I_t = 2^t > n$, which is when $t > \log n$. Inside the t 'th iteration of the while loop, the FOR loop repeats $n/I_t - 1 \leq \lfloor n/I_t \rfloor \leq n/I_t$ times, each time taking $\Theta(I_t)$ time for a total of between $\Theta(n - I_t)$ and $\Theta(n)$ time each iteration. For the first $\log n - 1$ iterations, $I_t \leq n/2$, and both of these are $\Theta(n)$. Therefore, the total time is $\Theta(n \log n)$.

Correctness proofs Say that a graph is d -dense if every node has at least d edges adjacent to it. The following algorithm strategy determines whether graph G has a non-empty d -dense sub-graph G' . After the algorithm, a correctness proof with some gaps is given. Fill in the gaps to form a complete proof. (The gaps are numbered with Roman numerals. On your exam answers, write the phrases that should be in the gaps, in that order.)

Blanks are filled in in **boldface DenseSubgraph(G: undirected graph, d: positive integer)**

1. While there is a node x of degree $< d$ do:
2. $G \leftarrow G - \{x\}$.
3. If G is empty return False, else return True.

We need to prove that, if G has a d -dense sub-graph G' , then the above algorithm returns **True**, and conversely, that, if **the algorithm returns True**, then G has a d dense subgraph.

Assume that G has a d -dense subgraph $S \subseteq G$. We prove the algorithm by **induction** using the following loop invariant: Let G_t be the remaining graph after t iterations of the while loop. Then $S \subseteq G_t$.

Since after 0 loops, $G_0 = G$, the **base** case $t = 0$ is clear.

For the induction step, assume the hypothesis holds for t , i.e., that $S \subseteq G_t$. We will show it holds for $t + 1$, i.e., that $S \subseteq G_{t+1}$.

Let x be the node deleted in the $t + 1$ 'st iteration. By the definition of the algorithm $deg(x) < d$. By the definition of d -dense, all nodes $y \in S$ have $deg(y) \geq d$. Thus, $x \notin S$, so $S \subseteq G_t - \{x\} = G_{t+1}$. Thus, we have proved our goal. Then, by induction, the invariant holds after any number of iterations.

In particular, if the loop halts after t iterations, $S \subseteq G_t$. Since $S \neq \emptyset$ by the definition of d -dense, $G_t \neq \emptyset$. Thus, at this point, the algorithm returns **True** as we wanted to prove.

Conversely, assume the algorithm returns **True**. Then when the loop halts, we have $G_t \neq \emptyset$. Since the loop has halted, every node x in G_t has $deg(x) \geq d$. Thus, G_t is a **d-dense subgraph** of G , which proves the converse direction.

Data structures and efficient versions of algorithms 10 pts: Union of intervals: We are given a list of (not necessarily disjoint) intervals of the real line, $I_1 = [s_1, f_1] \dots I_n = [s_n, f_n]$, with each $s_i < f_i$. We want to express the union of these intervals as a set of disjoint intervals. A strategy to do this is:

1. Initialize an empty set of intervals. *Disjoint*
2. Initialize S as the smallest start time of an interval, and F as the finish time of this interval.
3. While there is an interval I_j that ends after F ($F < f_j$) do:
 4. IF there is an interval I_j so that $s_j \leq F < f_j$ THEN $F \leftarrow f_j$.
 5. ELSE do:
 6. Add $[S, F]$ to *Disjoint*.
 7. Find the interval I_j with the smallest s_j so that $F < s_j$
 8. $S \leftarrow s_j, F \leftarrow f_j$.
 9. Add $[S, F]$ to *Disjoint*.
10. Return *Disjoint*

Give an efficient implementation of this strategy, specifying pre-processing and data structures used. Give a time analysis for your version.

As a preprocessing step, we can sort the set of intervals by s_i . This takes $O(n \log n)$ time using heapsort or mergesort. The smallest start time of an interval is the first start time in the sorted set, and we initialize F to the first finish time. Since F only increases, we only need to move right in the sorted array to keep finding intervals with $s_j \leq F$ or to find the interval I_j with the smallest s_j so that $F < s_j$.

This leads to the following algorithm:

- Sort I_1, \dots, I_n by non-decreasing s_i , using Heapsort.
- $S \leftarrow s_1; F \leftarrow f_1$.
- *Disjoint* is initially an empty list of intervals.
- FOR $J = 2$ to n do:
 - IF $s_j \leq F$ THEN $F \leftarrow \max(F, f_j)$.
 - ELSE do:
 - Append (S, F) to *Disjoint*;
 - $S \leftarrow s_j$;
 - $F \leftarrow f_j$;
 - Append (S, F) to *Disjoint*.
- Return *Disjoint*.

The first step takes $O(n \log n)$ time. The inside of the FOR loop is constant time, so the rest of the algorithm is $O(n)$. Thus, the first step dominates completely and tyrannically, so the total time is $O(n \log n)$.

Divide-and-Conquer Recurrence: 10 points Consider the following recursive algorithm. Its input is an array of (not necessarily positive) integers. $A[1..n]$ The goal is to find the maximum possible sum of a consecutive sub-sequence $A[i..j]$ of elements of the array, $\max_{1 \leq i \leq j \leq n} \sum_{k=i}^j A[k]$.

MaxConsSum[A[1..n]]

1. IF $n = 0$ return 0.
2. IF $n = 1$ return $\max(0, A[1])$.
3. $S_1 \leftarrow \text{MaxConsSum}[A[1..n/2]]$
4. $S_2 \leftarrow \text{MaxConsSum}[A[n/2 + 1..n]]$
5. $\text{SumLeftFromCenter} \leftarrow 0$

6. $BestSLFC \leftarrow 0$
7. FOR $I = n/2$ down to 1 do:
 8. $SumLeftFromCenter \leftarrow SumLeftFromCenter + A[I]$
 9. IF $BestSLFC < SumLeftFromCenter$ THEN $BestSLFC \leftarrow SumLeftFromCenter$.
10. $SumRightFromCenter \leftarrow 0$
11. $BestSRFC \leftarrow 0$
12. FOR $I = n/2 + 1$ TO n do:
 13. $SumRightFromCenter \leftarrow SumRightFromCenter + A[I]$
 14. IF $BestSRFC < SumRightFromCenter$ THEN $BestSLFC \leftarrow SumRightFromCenter$.
15. $S_3 \leftarrow BestSLFC + BestSRFC$
16. Return $max(S_1, S_2, S_3)$.

Give a recurrence for the time $T(n)$ taken by the above algorithm Use the recurrence to give a time analysis up to order.

For $n \geq 2$, the algorithm always makes two recursive calls, in lines 3 and 4. Each recursive call has an input of size $n/2$. The rest of the algorithm consists of two FOR loops going for $n/2$ steps each, with all operations inside the loops constant time. Thus, the recurrence is $T(n) = 2T(n/2) + \Theta(n)$. Using the Main Recurrence Theorem with $a = 2, b = 2$ and $k = 1$, since $a = 2 = 2^1 = b^k$, we are in the steady state case, and the time is $T(n) \in \Theta(n^k \log n) = \Theta(n \log n)$.