Final #1

Mark all correct answers in each of the following questions. In parts 2.d and 3.a fill in the missing quantities.

1. We are given a graph G with non-negative weights attached to its edges. The graph is disassembled by deleting consecutively its vertices. At each stage, one vertex and all the edges incident to it (which were not deleted in former stages) are deleted. The cost of deleting a vertex is the product of the weights of the edges deleted along with the vertex. (This product is taken as 0 if the vertex has no edges incident to it at the time it is deleted.) The total cost of disassembling the graph is the sum of the costs at all stages. We are looking for an ordering of the vertices so that, when deleting the vertices according to this order, the cost is minimal.

   (a) There exists an ordering for which the total cost is 0 if and only if each vertex of the graph is incident to at least one edge of 0 weight.

   (b) Suppose G is connected (i.e., for each pair of vertices, there is a path connecting them). Then there exists an ordering for which the total cost is 0 if and only if the graph, obtained from G by deleting all edges with positive weights, is still connected.

   (c) For $n \geq 2$, the set of all total costs associated with the $n!$ orderings of V contains at most $n!/2$ distinct numbers.

   (d) Suppose we order V according to the greedy algorithm, as follows. At each stage we delete that vertex whose deletion yields minimal possible cost. We can implement this solution in polynomial time, but it is not necessarily the optimal solution.

(e) Recall the algorithm presented in class for solving the traveling salesman problem in time $O(P(n)2^n)$ (where P is some polynomial). Employing a similar idea, we can solve our problem in time $O(P(n)2^n)$.

(f) In the preceding parts, we assumed that the cost associated with the deletion of a vertex is $\prod_{i=1}^{k} w_i$, where $w_1, w_2, \ldots, w_k$ are the weights of the edges incident to that vertex at the stage it is deleted. Now suppose the cost is $\sum_{i=1}^{k} f(w_i)$, where $f : \mathbf{R} \longrightarrow \mathbf{R}$ is some increasing function. Then there exists a polynomial algorithm to find the optimal ordering of V.

2. We are given an array $\mathbf{v}$ of length n, whose entries are elements of a certain set A. An element $x \in A$ is a *majority element* of $\mathbf{v}$ if more than $n/2$ of the entries of $\mathbf{v}$ are equal to x.

(a) Assume first that a total order relation $\prec$ is defined on A, such that, given $x, y \in A$ one can check in time $O(1)$ whether $x \prec y, x = y$ or $x \succ y$. Then one can find out whether there exists a majority element in $\mathbf{v}$, and if so find this element, as follows: First, sort $\mathbf{v}$. Next, scan $\mathbf{v}$, recording the lengths of the blocks consisting of equal elements. Finally, check whether any of these lengths exceeds $n/2$. The algorithm takes $O(n \log n)$ time.

(b) By finding a majority element of an array, we have in particular sorted at least half of this array. Since sorting an array of length $n/2$ cannot be done in less than $O((n/2)\log(n/2)) = O(n \log n)$ time, it is impossible to find an algorithm which takes $o(n \log n)$ time.

(c) Assume now that we do not have an order relation defined on A. The following algorithm solves the problem in this case: First find out (recursively) whether the first half of the array contains a majority element, say x, and whether the second half of the array contains a majority element, say y. If neither such an x nor such a y exist, $\mathbf{v}$ does not contain a majority element. If x and/or y exist, check directly whether any of them is a majority element.

(d) The algorithm in the preceding part works in time $\Theta(???)$.

(e) An element $x \in A$ *has a sizeable representation* in $\mathbf{v}$ if at least a quarter of the entries of $\mathbf{v}$ are equal to x. Assume again that we have a total order relation on A. By employing (several times) the algorithm for finding the k-th element of a set, we can find in linear time all elements having a sizeable representation in $\mathbf{v}$.

3. Given an array $(x_1, x_2, \ldots, x_n)$ of integers, we want to count the number of possible relations in terms of equality and inequality between all x_i's. For example, if $n = 3$, then we have 13 possibilities:

   (i) $x_1 = x_2 = x_3$.

   (ii) $x_1 = x_2 < x_3$, $x_3 < x_1 = x_2$, $x_1 = x_3 < x_2$, $x_2 < x_1 = x_3$, $x_2 = x_3 < x_1$.

   (iii) $x_1 < x_2 < x_3$, $x_1 < x_3 < x_2$, $x_2 < x_1 < x_3$, $x_2 < x_3 < x_1$, $x_3 < x_1 < x_2$, $x_3 < x_2 < x_1$.

   In general, denote by a_n the number of all possibilities, and by a_{ni} the number of those possibilities in which i of the relations are inequalities and the other $n-1-i$ are equalities. (The splitting above into three classes was done according to the value of i.) Obviously, $a_n = \sum_{i=0}^{n-1} a_{ni}$ for each n. The idea is to employ dynamic programming for calculating the a_{ni}'s. In fact, the a_{ni}'s satisfy the recurrence:

$$a_{ni} = \begin{cases} f(i)a_{n-1,i} + g(i)a_{n-1,i-1}, & 1 \le i \le n-2, \\ a_{n-1,0}(= 1), & i = 0, \\ na_{n-1,n-2}(= n!), & i = n-1. \end{cases}$$

   (a) $f(i) =???, g(i) =???$.

   (b) The calculation of all a_{ki}'s with $0 \le i < k \le n$ (and thereby a_n) from bottom to top using the recursion takes $\Theta(n^3)$ time.

   (c) The number of all a_{ki}'s is approximately $n^2/2$, so that a direct implementation of the formula requires $O(n^2)$ space. However, it is possible to never keep the values of more than n of the a_{ki}'s, and in particular do with $O(n)$ space.

   (d) $a_n = O(P(n)n!)$ for an appropriate polynomial P.

4. (a) We explained in class that it takes an exponential number of steps to calculate the n-th Fibonacci number F_n recursively (directly from the definition), but takes only $O(n)$ steps if we calculate the F_i's successively, starting from $i = 2$, then $i = 3$, and so forth until we get to $i = n$. In fact, this is not correct if we take into account the fact that the Fibonacci numbers grow very fast, and we shall need to store the calculated values as arrays, corresponding to the expansion of the numbers in some base (say, binary). Since the length of a number n is approximately proportional to $\log n$, the number of steps required to calculate F_n in this method is in fact $\Theta(n \log n)$.

   (b) A polynomial will be called (for the purposes of this question) *squarish* if it is of the form $\sum_{k=0}^{L} a_k x^{k^2}$. (For example, the polynomial $2x^{100} - 5x^9 - 17$ is squarish.) We are given two squarish polynomials $P(x) = \sum_{k=0}^{\lfloor\sqrt{n}\rfloor} a_k x^{k^2}$ and $Q(x) = \sum_{k=0}^{\lfloor\sqrt{n}\rfloor} b_k x^{k^2}$, of degrees not exceeding n, and need to find their product $P(x)Q(x) = \sum_{m=0}^{2n} c_m x^m$ (which is non-squarish in general). Consider the following two options for calculating the product (namely, the $2n+1$ coefficients c_m). The first option is to multiply the polynomials in the classical way. That is, we start with all c_m's being 0. Then, for each i and j between 0 and $\lfloor\sqrt{n}\rfloor$ we increase $c_{i^2+j^2}$ by $a_i b_j$. The second option is to use FFT. Due to the special form of the polynomials, it is faster to use the classical method rather than FFT.

   (c) We are given two vectors $\mathbf{u} = (u_1, u_2, \ldots, u_n), \mathbf{v} = (v_1, v_2, \ldots, v_n) \in \mathbf{R}^n$. The coordinates of each vector distinct positive numbers. These coordinates cannot be calculated precisely, but they can be approximated arbitrarily well, and in particular it is possible to decide, for any $1 \le i < j \le n$, whether $u_i < u_j$ or $u_i > u_j$ (and whether $v_i < v_j$ or $v_i > v_j$) in time $O(1)$. It is required to permute the coordinates of each vector, thus obtaining the vectors $\mathbf{u}' = (u_{\sigma(1)}, \ldots, u_{\sigma(n)}), \mathbf{v}' = (v_{\tau(1)}, \ldots, v_{\tau(n)})$, in such a way that the inner product $\langle \mathbf{u}', \mathbf{v}' \rangle = \sum_{i=1}^{n} u_{\sigma(i)} v_{\sigma(i)}$ will be minimal. Even though the vectors are not precisely known, it is possible to find such permutations σ and τ. Moreover, this can be done in time $\Theta(n^2)$, but not in less.

   (d) In the strong pseudo-primality test, the prime 2 plays a unique

role. However, it should not. In fact, consider the following test: We are given integers n and a, with $0 < a < n$ and $\gcd(3a, n) = 1$. Write $n - 1 = 3^t m$, where $\gcd(m, 3) = 1$. If $a^m \not\equiv 1 \pmod{n}$ and $a^{3^l m} \not\equiv -1 \pmod{n}$ for $l = 0, 1, \ldots, t-1$, then output "n is composite", otherwise "undecided". This test is correct, namely, if it returns "n is composite", then indeed n is composite. The only problem is that it is not necessarily the case that, if n is composite, then the test says so for at least half of the possible a's.

(e) Consider the problem, discussed in class, of job scheduling with deadlines. Assume that, unlike the case we discussed, each job i has its own duration T_i. Then it is still true that a set of jobs is acceptable if and only if, when ordering it according to deadlines, we obtain an acceptable ordering.

Solutions

1. The characterization in (a) is false. In fact, let G be the complete graph on 4 vertices v_1, v_2, v_3, v_4, where the weights of (v_1, v_2) and (v_3, v_4) are 0, and all other weights are strictly positive. Then every vertex is incident to a 0 weight edge, but the cost of disassembling the graph is positive for every ordering of the vertices.

   The characterization in (b) is also false. In fact, let G be a path graph on 4 vertices v_1, v_2, v_3, v_4, where v_1 and v_4 are the leaves, the edge (v_2, v_3) is of positive weights and the other edges are of weight 0. The graph is connected, the graph obtained by deleting the 0 weight edges is disconnected, yet by deleting v_2 and v_3 first we can disassemble the graph at 0 cost.

   Any two orderings, differing only in the last two entries, yield the same total cost. Hence there are no more than $n!/2$ possible distinct costs.

   The greedy algorithm fails in general to give the optimal algorithm. For example, let G be the complete graph on 3 vertices, the edge weights

being $1, 1/2$ and $1/4$. The greedy algorithm yields an ordering with total cost $9/8$, whereas the optimum is easily seen to be only $3/4$.

For $S \subseteq V$, Denote by $C(S)$ the minimal possible cost of deleting the vertices in S only. Thus we need to calculate $C(V)$ and find an ordering yielding this value. Now $C(S) = \prod_{u \in V-\{v\}} w(v,u)$ for $S = \{v\}$. (The product is considered as 0 if it is empty. If it is not, then $w(v,u)$ is considered as 1 for non-existing edges (v,u).) We calculate $C(S)$ for increasingly larger sets V using the recursion:

$$C(S) = \min_{v \in S} \left(C(S - \{v\}) + \prod_{u \in V-S} w(v,u) \right).$$

The calculation is carried for (almost) all 2^n subsets of V, and for each of them takes $O(n^2)$ time (which can be lowered to $O(n)$ with little effort). Knowing all the values $C(S)$, it is easy to find an optimal ordering. Thus we can find an optimal ordering in time $O(P(n)2^n)$.

With the cost function of part (f), the total cost associated with any ordering of the vertices is the sum of costs of all edges of the graph. Thus, all solutions are optimal, and in particular we can present an optimal solution in polynomial time.

Thus, (c), (d), (e) and (f) are true.

2. The algorithm suggested in (a) takes $O(n \log n)$ time for the sorting, and then another $O(n)$ for checking whether any of the blocks of the sorted array is longer than half the whole array. Hence it works in $O(n \log n)$ time. The considerations in (b) are false, as they certainly do not apply when there is no majority element in the array. Moreover, the conclusion of those considerations, whereby it is impossible to find an algorithm which takes $o(n \log n)$ time, is also false. In fact, a majority element, if one exists, must be equal to the median of the array. Now we can find the median of the array, and then check if it is a majority element, in linear time. Since a majority element of an array must be a majority element in at least one of the halves of the array, the algorithm in (c) is indeed correct. The algorithm calls itself twice for $n/2$ (if n is even), and then finishes the work in $O(n)$ time. Hence,

denoting by a_n the number of steps it takes, we have $a_n \leq 2a_n + Cn$ for some constant C. Hence $a_n = \Theta(n \log n)$. An element with a sizeable representation, if one exists, must be equal to the element ranking number k in the array for at least one of the following three k's: $k = \lceil n/4 \rceil, k = 2 \cdot \lceil n/4 \rceil, k = 3 \cdot \lceil n/4 \rceil$. Hence by finding these three elements of the array, and checking for each whether it is of sizeable representation, we indeed find all these elements in linear time.

Thus, (a), (c) and (e) are true, and the algorithm in (c) works in $\Theta(n \log n)$ time.

3. Each arrangement of $x_1, x_2, \ldots, x_n$ is obtained from some arrangement of $x_1, x_2, \ldots, x_{n-1}$ by adding x_n at an appropriate place, either as equal to one of the existing sets of equal elements or as a new set. More precisely, any arrangement with $i > 0$ inequalities and $n - 1 - i > 0$ equalities (and thus counted in a_{ni}) is obtained either from an arrangement of $n - 1$ elements with i inequalities and $n - 2 - i$ equalities by adding x_n as equal to one of the $i + 1$ existing groups or from an arrangement of $n - 1$ elements with $i - 1$ inequalities and $n - 1 - i$ equalities by adding x_n as a separate group. Hence for $1 \leq i \leq n - 2$ we have $a_{ni} = (i + 1)a_{n-1,i} + (i + 1)a_{n-1,i-1}$. The number of all a_{ki}'s is approximately $n^2/2$, and the calculation of each in its turn takes $O(1)$ time, so that the whole calculation takes $\Theta(n^2)$ time. It suffices to keep one array of length n. At the end of stage k, places 0 through $k - 1$ of this array contain the values of the a_{ki}'s. At the next stage we fill in the values of the $a_{k+1,i}$'s, starting from $i = k$ and going down to $i = 0$. Since the recursion never uses larger values of the second index, we shall have all required (and updated) data when getting to any calculation.

   For each fixed l, the number of arrangements with l equal pairs and $n - l - 1$ inequalities is $\binom{n}{n-2l,2,2,\ldots,2}(n - l)!$. Since this expression is $\Omega(n^l \cdot n!)$, we have $a_n \geq a_{n,n-l-1} = \Omega(n^l \cdot n!)$ for each l.

   Thus, only (c) is true, and $f(i) = g(i) = i + 1$.

4. (a) False. It is true that the computations become more and more lengthy as we proceed. Since $F_i \approx c\lambda^i$ (where $\lambda = (1+\sqrt{5})/2$), we need $\Theta(\log F_i) = \Theta(i)$ time to find each F_i after F_{i-2} and F_{i-1} are known. Hence the whole computation takes $\sum_{i=2}^{n} \Theta(i) = \Theta(n^2)$ time.

   (b) True. The direct method takes $O(1)$ time for each i and j between 0 and $\lfloor\sqrt{n}\rfloor$, altogether $O(n)$, whereas when using FFT we do not benefit (at least at the last stage) from the special form of the polynomials, so that we still need $O(n \log n)$ time.

   (c) False. According to the conditions, we can sort the arrays, $\mathbf{u}$ according to increasing order and $\mathbf{v}$ according to decreasing order, in $O(n \log n)$ time. We claim that the resulting vectors $\mathbf{u}'$ and $\mathbf{v}'$ satisfy the required optimality condition. The proof is exactly the same as the one we gave for the correctness of the greedy algorithm for the problem of scheduling, when the criterion is minimization of the total time the jobs spend in the system.

   (d) False. As an example, take $n = 7, a = 2$. Then $t = 1, m = 2$. Since $2^2 \not\equiv 1 \pmod 7$ and $2^2 \not\equiv -1 \pmod 7$, the test returns "7 is composite".

   (e) True. Clearly, if there exists an acceptable ordering, then the set is acceptable. Now assume the set is acceptable. We shall prove by induction on the number n of jobs that, by ordering the jobs according to increasing deadlines, we obtain an acceptable ordering. In fact, this is trivial for $n = 1$. Suppose our claim is correct for n jobs, and let an acceptable set of $n + 1$ jobs be given. Take an acceptable ordering of the jobs. We claim that, by postponing the job $n+1$ with the last deadline to be the last, and performing all jobs currently planned to be done after it T_{n+1} time units earlier than planned, we still obtain an acceptable ordering. Indeed, job $n + 1$ will now terminate at a time when some other job was originally planned to terminate. Since job $n + 1$ has the latest deadline, this means that job $n + 1$ will still terminate on time. Regarding the other jobs, since they are now scheduled to be performed earlier, there is certainly no problem. Thus there exists an acceptable ordering in which the job with the last deadline is performed last. Ignoring now job $n + 1$, we have an acceptable

ordering of the first n jobs, which, by the induction hypothesis, can be ordered according to deadlines. This proves the claim.