CS60003 Algorithm Design and Analysis, Autumn 2010–11

End-Semester Examination

Maximum marks: 100 November 21, 2010 (FN) Total time: 3 hours

---

Roll no: ____________ **Name:** ______________________________

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. The knapsack problem discussed in the class is an optimization problem. Consider the following decision version of the knapsack problem. Given n objects $O_1, O_2, \ldots, O_n$ with respective weights $w_1, w_2, \ldots, w_n$ and with respective profits $p_1, p_2, \ldots, p_n$, and given a knapsack of capacity C and a profit bound P, decide whether there exists a subcollection $O_{i_1}, O_{i_2}, \ldots, O_{i_k}$ of the given objects such that $\sum_{j=1}^{k} w_{i_j} \leqslant C$ (knapsack capacity cannot be exceeded) and $\sum_{j=1}^{k} p_{i_j} \geqslant P$ (at least a profit of P can be made).

(a) Prove that the decision version of the knapsack problem can be solved in polynomial time if and only if the optimization version of the knapsack problem can be solved in polynomial time. **(10)**

Solution [If] Let M be a polynomial-time algorithm for solving the maximization problem. Using M, we determine the maximum profit P^*, and return *true* if and only if $P^* \geqslant P$.

[Only if] Let D be a polynomial-time algorithm for solving the decision problem. We invoke D multiple times with separate profit bounds P in order to determine the maximum profit P^*. Initially, we start with $L = 0$ and $R = \sum_{i=1}^{n} p_i$, since we definitely know that P^* must lie between these two values. We compute $P = \lfloor (L+R)/2 \rfloor$, and call D with this profit bound P. If D returns *true*, we conclude that P^* is between P and R, so we set $L = P$. On the other hand, if D returns *false*, we set $R = P - 1$, since P^* must be smaller than P. This binary search procedure is repeated until we have $L = R$. We output this value ($L = R$) as P^*.

The total number of invocations of D is $\mathrm{O}(\log \sum_{i=1}^{n} p_i)$ which is $\mathrm{O}(\log(np_{\max}))$. Since each invocation runs in polynomial time, the total running time is polynomial in n and $\log p_{max}$.

(b) Prove that the decision version of the knapsack problem is NP-Complete.

(Hint: You may use the partition problem which, given positive integers $a_1, a_2, \ldots, a_n$ with $A = \sum_{i=1}^{n} a_i$, decides whether there exists a subcollection $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ with $\sum_{j=1}^{k} a_{i_j} = A/2$.) **(10)**

Solution Clearly, the decision version of the knapsack problem is in NP.

In order to prove its NP-hardness, we reduce PARTITION to it. Let $a_1, a_2, \ldots, a_n$ be an input instance for PARTITION with $A = \sum_{i=1}^{n} a_i$.

We consider n objects $O_1, O_2, \ldots, O_n$ such that the weight of O_i is $w_i = 2a_i$ and the profit of O_i is $p_i = 2a_i$. Finally, we take the knapsack capacity $C = A$ and the profit bound $P = A$. Clearly, this reduction can be done in polynomial time.

Suppose that $\sum_{j=1}^{k} a_{i_j} = A/2$ for some subcollection $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of $a_1, a_2, \ldots, a_n$. But then $\sum_{j=1}^{k} w_{i_j} = 2 \times (A/2) \leqslant C$ and $\sum_{j=1}^{k} p_{i_j} = 2 \times (A/2) \geqslant P$, that is, the objects $O_{i_1}, O_{i_2}, \ldots, O_{i_k}$ satisfy the capacity constraint and the profit bound.

Conversely, suppose that the objects $O_{i_1}, O_{i_2}, \ldots, O_{i_k}$ satisfy $\sum_{j=1}^{k} w_{i_j} \leqslant C$ and $\sum_{j=1}^{k} p_{i_j} \geqslant P$. These, in turn, imply that $\sum_{j=1}^{k} 2a_{i_j} \leqslant A$ and $\sum_{j=1}^{k} 2a_{i_j} \geqslant A$, that is, $\sum_{j=1}^{k} a_{i_j} = A/2$. Therefore, the integers $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ satisfy the requirement of the PARTITION problem.

2. A cut in an undirected graph $G = (V, E)$ is a partition of V in two (disjoint) subsets S, T. Define by $E(S, T)$ the set of all edges of G with one endpoint in S and the other in T. The MAX-CUT problem is an optimization problem that determines a cut S, T for which the size of the set $E(S, T)$ (the number of cross edges) is as large as possible.

Recall that in the Ford-Fulkerson algorithm, we have dealt with *minimum* cuts in order to solve the dual problem of maximizing network flow. The Ford-Fulkerson algorithm is not truly polynomial-time, but has variants that run in polynomial time in the input size. The MAX-CUT problem (more correctly, a suitable decision version of this problem), on the other hand, is NP-Complete (you are not asked to prove this).

Prof. Myopia proposes the following approximation algorithm for solving the MAX-CUT problem.

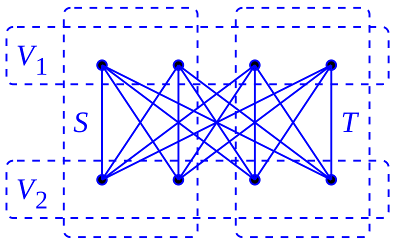
```
1. Start with an arbitrary partition $S, T$ of $V$.
2. Repeat the following two steps until no further vertex movement is possible:
    (a) For each vertex $v \in S$, check whether the cut $(S - v, T + v)$ has more cross edges than $(S, T)$;
        and if so, delete $v$ from $S$ and include $v$ in $T$.
    (b) For each vertex $v \in T$, check whether the cut $(S + v, T - v)$ has more cross edges than $(S, T)$;
        and if so, delete $v$ from $T$ and include $v$ in $S$.
3. Return $S, T$.
```

(a) Prove that Prof. Myopia's algorithm runs in polynomial time (in the input size). **(5)**

Solution First, note that Prof. Myopia's algorithm terminates, since each vertex movement strictly increases the size of $E(S, T)$. Let $n = |V|$ and $m = |E|$. Prof. Myopia's algorithm does not require more than m vertex movements. Each vertex movement can be completed in $O(n^2)$ time under any standard representation of G (like the adjacency-matrix representation). In fact, checking whether $|E(S - v, T + v)| > |E(S, T)|$ (or $|E(S + v, T - v)| > |E(S, T)|$) requires looking at the neighbors of v only. Moreover, a vertex suitable for shifting is to be found out among n candidates. To sum up, Prof. Myopia's algorithm can be implemented to run in $O(mn^2)$ time.

(b) Prove or disprove: Prof. Myopia's algorithm outputs the optimal solution for bipartite graphs. **(5)**

Solution *False.* Consider the following bipartite graph (the complete bipartite graph $K_{4,4}$). Suppose that we start with the partition S, T as shown. Migration of any vertex from S or T to the other part cannot increase $E(S, T)$, since each vertex has two neighbors in S and two neighbors in T too. Thus, Prof. Myopia's algorithm reports this locally maximum solution for which $|E(S, T)| = 8$. On the other hand, $|E(V_1, V_2)| = |E| = 16$.



(c) Prove that the approximation ratio of Prof. Myopia's algorithm is $1/2$. **(5)**

Solution Suppose that at some point of time, S, T satisfy $|E(S,T)| < m/2$. Let c_i be the number of cross edges incident upon the i-th vertex v_i, and b_i the number of non-cross edges incident upon v_i. Clearly, $b_i + c_i = d_i$ (the degree of v_i). By the degree-sum formula,

$$2m = \sum_{i=1}^{n} d_i = \sum_{i=1}^{n} b_i + \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} b_i + 2|E(S,T)| < \sum_{i=1}^{n} b_i + m,$$

that is, $\sum_{i=1}^{n} b_i > m$, that is, $\sum_{i=1}^{n} b_i > \sum_{i=1}^{n} c_i$. This implies that there must exist (at least) one vertex v_i for which $b_i > c_i$. Shifting v_i to the other part increases the number of cross edges by $b_i - c_i > 0$. To sum up, Prof. Myopia's algorithm stops after (not necessarily immediately after) $|E(S,T)| \geqslant m/2$. On the other hand, an optimal cut S^*, T^* evidently satisfies $|E(S^*,T^*)| \leqslant m$. Thus, $|E(S,T)|/|E(S^*,T^*)| \geqslant 1/2$.

(d) Demonstrate that this approximation ratio is tight (suggest an *infinite* family of graphs). **(5)**

Solution The construction of Part (b) can be generalized. Consider the complete bipartite graph $K_{2n,2n}$ for any $n \geqslant 1$. The optimal cut is V_1, V_2 for which $|E(V_1,V_2)| = 4n^2$. On the other hand, Prof. Myopia's algorithm may start with S consisting of exactly n vertices from V_1 and exactly n vertices from V_2. It is easy to see that this is a local maximum with $|E(S,T)| = 2n^2$.

3. The subset-sum problem (SSP) decides whether a given collection of positive integers $a_1, a_2, \ldots, a_n$ has a subcollection whose elements add up to a given positive integer t. We proved that SSP is an NP-Complete problem. Recall also that an algorithm is called *pseudo-polynomial-time*, if its running time is a polynomial in the size of the *unary* representation of the input. We call an NP-Complete problem *weakly NP-Complete* if it admits a pseudo-polynomial-time algorithm. Prove that SSP is weakly NP-Complete.

(Hint: The knapsack problem might help you. Also note that the unary size of $a_1, a_2, \ldots, a_n$ is $n + \sum_{i=1}^{n} a_i$.) **(20)**

Solution Let $a_1, a_2, \ldots, a_n$ be the integers and t the target sum in an instance of SSP. Let $A = \sum_{i=1}^{n} a_i$. The unary size of this input is $\mathrm{O}(A + n)$ (assume that $t \leqslant A$). Thus, we need to have an algorithm with running time polynomial in both n and A. Here goes one.

Build an $(n+1) \times (A+1)$ table T such that $T(i, j)$ is the decision of SSP on $a_1, a_2, \ldots, a_i, j$. The table is populated in the row-major order. The zeroth row is initialized as

$$T(0, j) = \begin{cases} 1 & \text{if } j = 0, \\ 0 & \text{if } j > 0. \end{cases}$$

Here, $i = 0$ means there are no input integers a_i, so the only sum achievable is 0.

Subsequently, for $i \geqslant 1$, we consider two cases. If $j < a_i$, then we cannot include a_i to achieve a sum of j, that is, a sum of j is achievable if and only if the first $i - 1$ integers have a subcollection of sum j. On the other hand, if $j \geqslant a_i$, then we have a choice of including a_i in a subcollection. If we include a_i, we check whether the remaining sum $j - a_i$ can be achieved by a subcollection of the first $i - 1$ integers. If we do not include a_i, then the sum j itself has to be achieved by a subcollection of $a_1, a_2, \ldots, a_{i-1}$. To sum up, we have

$$T(i, j) = \begin{cases} T(i-1, j) & \text{if } j < a_i, \\ T(i-1, j-a_i) \ \text{ OR } \ T(i-1, j) & \text{if } j \geqslant a_i. \end{cases}$$

Finally, we return $T(n, t)$ as the output.

This algorithm needs to compute $\leqslant (n+1)(A+1)$ entries in the table T with each entry requiring only $\mathrm{O}(1)$ time (better $\mathrm{O}(\log n + \log A)$ time, since i can be as large as n and j as large as A). It follows that the running time is polynomial in both n and A, as desired.

4. Consider the problem of finding the i-th smallest element in an array A of n integers. Ms. Lucky proposes the following randomized algorithm to solve this problem. She chooses a (uniformly) random element x of A. She then uses the partitioning algorithm of Quick Sort on A with respect to the pivot x. Suppose that x is placed in the k-th position after the partitioning (counting starts from 1). If $k = i$, the algorithm returns x. If $k > i$, then a recursive call is made on the smaller subarray (of size $k - 1$) and with the same i. Finally, if $k < i$, then a recursive call is made on the larger subarray (of size $n - k$) with i replaced by $i - k$. Deduce that the expected running time of Ms. Lucky's algorithm is $O(n \log n)$. (Notice that this running time may depend upon i (in addition to n). In your calculations, you may suitably ignore this dependence.) **(20)**

Solution Let $T(n)$ denote the expected running time of Ms. Lucky's algorithm on an array of size n. Since the pivot is chosen randomly, each of the n possible values $(1, 2, \ldots, n)$ of k is equally likely, that is, of probability $1/n$. If $k = i$ (a case with probability $1/n$), the algorithm stops after returning the pivot. This case takes constant time. If $k > i$, a recursive call is made on a subarray of size $k - 1$. Finally, if $k < i$, the recursive call is on a subarray of size $n - k$. For $n \geqslant 2$, it then follows that

$$\begin{aligned} T(n) &\leqslant \frac{1}{n} + \frac{1}{n}\Big(T(i) + T(i+1) + \cdots + T(n-1)\Big) + \\ & \quad \frac{1}{n}\Big(T(n-i+1) + T(n-i+2) + \cdots + T(n-1)\Big) + cn. \end{aligned}$$

Here, the term cn on the right side stands for the running time of the partitioning phase. Let us also take

$$T(1) = 1.$$

For $n \geqslant 3$, we then have

$$nT(n) - (n-1)T(n-1) \leqslant 2T(n-1) - T(n-i) + c(2n-1) \leqslant 2T(n-1) + c(2n-1),$$

that is,

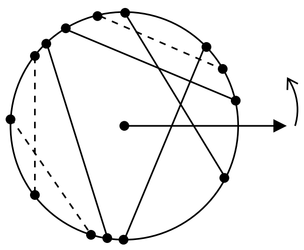
$$\begin{aligned} \frac{T(n)}{n+1} &\leqslant \frac{T(n-1)}{n} + c\left[\frac{2n-1}{n(n+1)}\right] \\ &= \frac{T(n-1)}{n} + c\left[\frac{3}{n+1} - \frac{1}{n}\right] \\ &\leqslant \frac{T(n-2)}{n-1} + c\left[3\left(\frac{1}{n+1} + \frac{1}{n}\right) - \left(\frac{1}{n} + \frac{1}{n-1}\right)\right] \\ &\leqslant \frac{T(n-3)}{n-2} + c\left[3\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1}\right) - \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2}\right)\right] \\ &\leqslant \cdots \\ &\leqslant \frac{T(2)}{3} + c\left[3\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{4}\right) - \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3}\right)\right]. \end{aligned}$$

This implies that

$$T(n) \leqslant \left(\frac{n+1}{3}\right) T(2) + 3c + c(n+1)\left[3\left(H_n - \frac{1}{1} - \frac{1}{2} - \frac{1}{3}\right) - \left(H_n - \frac{1}{1} - \frac{1}{2}\right)\right].$$

Since $T(2)$ is a constant and H_n is $\Theta(\log n)$, the expected running time of Ms. Lucky's algorithm is $O(n \log n)$.

5. You are given a set of n chords in a circle. Each chord may be viewed as an opaque piece of string. Your task is to determine which chords are visible (fully or partially) from the center. An example is given below. The dotted chords are the only chords that are not visible (not even partially) from the center.



(a) Propose an $O((n+h)\log n)$-time ray-sweep algorithm for solving this problem, where h is the total number of intersections of the given chords. Clearly describe the events in your algorithm and how they are handled. Assume that the chords are in general position, that is, no two of them share an endpoint, and no three of them are concurrent. **(10)**

Solution A ray emanating from the center of the circle makes a full sweep of 360^0 starting from the horizontal right position. A chord is active at a position of the ray, if the ray intersects the chord. We maintain two data structures as usual.

The *sweep ray information* S stores the list of all chords that are currently active. This list is kept sorted in accordance with their distances from the center along the sweeping ray.

The *event queue* Q stores the endpoints of the chords, that are yet to be encountered. Q should also store the intersection of the pair of chords C_i, C_j provided that both are active and are consecutive along the ray and has an intersection point lying after the current location of the ray. Q is kept sorted in the increasing order of the angle from the initial position of the ray.

S is initialized to the list of active chords at the beginning position of the sweeping ray. Q is initialized by the $2n$ endpoints of the chords and the intersection points of consecutive active chords (provide that the intersection exists and lies in a future position of the sweeping ray).

The following three events are handled. In all these cases, an event is deleted from Q after it is handled.

Enter chord C_i: C_i is inserted in S. At this point, C_i is farthest from the center among all active chords. However, if C_i is the *only* active chord, it is reported as visible. If not, let C_s be the second farthest active chord. The intersection point $\cap(C_s, C_i)$ (if it exists and is a future event) is inserted in Q.

Leave chord C_i: Delete C_i from S. Just before this deletion, C_i was the farthest active chord from the center. Consequently, nothing else needs to be done.

Intersection of chords C_i, C_j: Swap the positions of C_i and C_j in S. Assume that C_i was nearer to the center than C_j before this swapping. If C_i happened to be the closest active chord from the center (before the swapping), then C_j is reported as visible. Let C_s, C_t be the neighbors of the pair C_i, C_j on S (one or both of these neighbors may be non-existent). The intersection points $\cap(C_s, C_i)$ and $\cap(C_j, C_t)$ are deleted from Q (if they were in Q at all), and the intersection points $\cap(C_s, C_j)$ and $\cap(C_i, C_t)$ are added to Q, if it is appropriate to do so.

(b) Mention relevant data structures that your algorithm uses (like the organization of the event queue and the sweep-ray information). **(5)**

Solution Q must support arbitrary insertions and deletions, and may be organized as a height-balanced binary search tree (like an AVL tree). S must support insertions and deletions of maximum, arbitrary swaps of consecutive elements, and checks for minimum, so S may again be realized as a height-balanced BST.

(c) Deduce that the running time of your algorithm is $\mathrm{O}((n+h)\log n)$. **(5)**

Solution Initialization of Q and S can be done in $\mathrm{O}(n \log n)$ time ($\mathrm{O}(n)$ insertions in the height-balanced BST's.) There are exactly n enter chord events, n leave chord events and h intersection events. Each event involves $\mathrm{O}(1)$ insertions and deletions in two BST's. The BST realizing S is of maximum size n, whereas the BST realizing Q is of maximum size $n + (n-1)$, indicating that each tree insertion or deletion can be done in $\mathrm{O}(\log n)$ time.

For rough work. If used for continuation of answers from Pages 1–10, supply appropriate pointers earlier.