EE602: ALGORITHMS

HOMEWORK #6 Due: 12/12

Hualiang Li

(1) 15.1-1 Show that equation (15.4) follows from equation (15.3) and the initial condition T(0) = 1.

$T(0) = 1$
$T(1) = 1+T(0)$
$T(2) = 1+T(1) + T(0)$
$T(n) = 1+T(n-1)+T(n-2)+T(n-3)+\ldots\ldots+T(2)+T(1)+T(0)$
$T(n+1) = 1+T(n) +T(n-1)+T(n-2)+T(n-3)+\ldots\ldots+T(2)+T(1)+T(0)$
$= 1+(1+T(n-1)+T(n-2)+T(n-3)+\ldots\ldots+T(2)+T(1)+T(0))$
$+ T(n-1)+T(n-2)+T(n-3)+\ldots\ldots+T(2)+T(1)+T(0)$
$= 2*(1+T(n-1)+T(n-2)+T(n-3)+\ldots\ldots+T(2)+T(1)+T(0))$
$=2*T(n)$

so T(n) = T(n-1) * 2, with initial condition T(0) = 1, it is easy too see that T(n) = 2^n.

(2) 15.1-3 Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

```
Let r[0…n] and s[0…n] be new arrays
R[0] = 0
For j = 1 to n
        q = -infinity
        for i=1 to j
                if q<p[i]+r[j-i]+c
                        q = p[i] + r[j-i]+c
                        s[j] = i
        r[j] = q
return r and s
```

(3) 15.2-4 Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

P(n) = 1, if n= 1,
P(n) = sum(p(k)*p(n-k)) from k=1 to k = n-1, otherwise
There are (1+n+1)*n/2 vertices, (n+2)*n edges.

(4) 16.1-1 Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes c[i, j] as defined above and also produce the maximum-size subset of mutually compatible activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

```
initialize c[i,j] = 0
  for i <- 1 to n
     do for j <- 2 to n
        do if i >= j
           then c[i,j] <- 0
        else
           for k <- i+1 to j-1
              do if c[i,j] < c[i,k] + c[k,j] + 1
                 then c[i,j] <- c[i,k] + c[k,j] + 1
                             s[i,j] <- k
```

Time complexity for dynamic programming: O(n^3)

For greedy method, it is O(n)

(5) 16.1-2 Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields and optimal solution.

```
n <- length[s]
A <- {an}
i <- n
for m <- n-1 to 1
   do if fi >= si
      then A <- A U {am}
         i <- m
         return A
```

We are given a set S = {a1,a2,...,an}
of activities, where ai = [si, fi), and we propose to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Instead, let us create a set S′ = {a1′ ,a2′ ,...,an′ }, where ai′ = [fi,si]. That is, ai′ is ai in reverse. Clearly, a subset of {ai1 , ai2 , . . . , aik } ⊆ S is mutually compatible if and only if the corresponding subset {a′ , a′ , . . . , a′ } ⊆ S′ is also i1i2 ik mutually compatible. Thus, an optimal solution for S maps directly to an optimal solution for S′ and vice versa. The proposed approach of selecting the last activity to start that is compatible with all previously selected activities, when run on S, gives the same answer as the greedy algorithm from the textóselecting the first activity to finish that is com- patible with all previously selected activitiesówhen run on S′ .

(6) 16.2-4 Professor Gecko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border from Montana. The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections, than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of

water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distance between these locations.
The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution and give its running time.

At each gas station, Professor Midas should check whether he can make it to the next gas station without stopping at this one. If he can, skip this one. If he cannot, then fill up. Professor Midas doesn't need to know how much gas he has or how far the next station is to implement this approach, since at each fill-up, he can determine which is the next station at which he'll need to stop.

The optimal structure is as following: suppose there are m possible gas stations. Consider an optimal solution with s stations and whose first stop is at the kth gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m - k$ stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $s - 1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

The greedy solution is as following: suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the kth station as its first stop. No station beyond the kth works as a first stop, since Professor Midas runs out of gas first. If a solution chooses a station $j < k$ as its first stop, then Professor Midas could choose the kth station instead, having at least as much gas when he leaves the kth station as if he'd chosen the jth station. Therefore, he would get at least as far without filling up again if he had chosen the kth station.

If there are m gas stations on the map, Midas needs to inspect each one just once. The running time is O(m).

(7) 17.2-1 Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Charge \$2 for each PUSH and POP operation and \$0 for each COPY. When we call PUSH, we use \$1 to pay for the operation, and we store the other \$1 on the item pushed. When we call POP, we again use \$1 to pay for the operation, and we store the other \$1 in the stack itself. Because the stack size never exceeds k, the actual cost of a COPY operation is at most \$k, which is paid by the \$k found in the items in the stack and the stack itself. Since there are k PUSH and POP operations between two consecutive COPY operations, there are \$k of credit stored, either on individual items (from PUSH operations) or in the stack itself (from POP operations) by the time a COPY occurs. Since the amortized cost of each operation is O(1) and the amount of credit never goes negative, the total cost of n operations is O(n).

(8) 17.2-3 Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as Θ(1), show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time O(n) on an initially zero counter. (Hint: Keep a pointer to the high-order 1.)

We introduce a new field max[A] to hold the index of the high-order 1 in A. Initially, max[A] is set to −1, since the low-order bit of A is at index 0, and there are initially no 1ís in A. The value of max[A] is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTs.

```
INCREMENT(A)
i ← 0
while i < length[A] and A[i] = 1
    do A[i] ← 0
       i ← i + 1
if i < length[A]
  then A[i] ← 1
       ▷ Additions to book's INCREMENT start here
       if i > max[A]
         then max[A] ← i
  else max[A] ← −1

RESET(A)
for i ← 0 to max[A]
    do A[i] ← 0
max[A] ← −1
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update max[A]. Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing. In addition, we'll use \$1 to pay to update max, and if max increases, we'll place an additional \$1 of credit on the new high-order 1. (If max doesn't increase, we can just waste that \$1óit won't be needed.) Since RESET manipulates bits at positions only up to max[A], and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to max[A], every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of A by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes O(n) time.

(9) 17.3-3 Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in O(lg n) worst-case time. Give a potential function Φ such that the amortized cost of INSERT is O(lg n) and the amortized cost of EXTRACT-MIN is O(1), and show how it works.

Let the potential function be $\Phi(D_i) = \sum_{k=1}^{i} \lg k$. Then the amortized cost for INSERT is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le \lg i + \sum_{k=1}^{i} \lg k - \sum_{k=1}^{i-1} \lg k = 2\lg i \in O(\lg n).$$

The amortized cost for EXTRACT-MIN is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le \lg i + 1 + \sum_{k=1}^{i-1} \lg k - \sum_{k=1}^{i} \lg k = 1 \in O(1).$$

(10) 22.1-6 Most graph algorithms that take an adjacency-matrix representation as input require time Ω(V2), but there are some exceptions. Show how to determine whether a directed graph G contains a universal sink – a vertex with in-degree |V|-1 and out degree 0 – in time O(V), given an adjacency matrix for G.

If vertex i is a universal sink according to the definition, the i-th row of the adjacency-matrix will be all "0", and the i-th column will be all "1" except the aii entry, and clearly there is only one such vertex. We then describe an algorithm to find out if a universal sink really exist.
Starts from a11. If current entry aij = 0 then j = j + 1 (take one step right); if aij = 1 then i = i +1 (take one step down). In this way, it will stop at an entry akn of the last row or ank of the last column (n = |V|, 1 □ k □|V|). Check if vertex k satisfies the definition of universal sink (check for kth row to contain V zeros and kth column to contain k-1 1s, because the column in adjacency matrix defines in degree of a vertex), if yes then we found it, if no then there is no universal sink. Since we always make a step right or down, and checking if a vertex is a universal sink can be done in O(V), the total running time is O(V).
If there is no universal sink, this algorithm won't return any vertex. If there is a universal sink u, the path starts from a11 will definitely meet u-th column or u-th row at some entry. Once it's on track, it can't get out of the track and will finally stop at the right entry.

(11) 22.2-3 Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if lines 5 and 14 were removed.

We do not need to use "grey" color, because every time we visit a white color, we know that it is not visited and we will eventually pain it to black if all the adjacency are visited, therefore, the gray color can be removed.

(12) 22.3-5 Argue that in a breadth-first search, the value u.d assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using

Figure 22.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

The correctness proof for the BFS algorithm shows that u.d = δ(s, u), and the algorithm doesn't assume that the adjacency lists are in any particular order.

In Figure 22.3, if t precedes x in Adj[w], we can get the breadth-first tree shown in the figure. But if x precedes t in Adj[w] and u precedes y in Adj[x], we can get edge (x, u) in the breadth-first tree.

(13) 23.1-1 Let (u,v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

In the first step of GENERIC-MST, we could choose such a cut, node u is on one side, node v is on another side. Then(u, v) is a light-edge through this cut. So, it is safe to add (u, v)

(14) 23.1-6 Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Assuming there are two MSTs called T and T'. For any edge e in T, if we refove e from T, then T becomes unconnected and we have a cut(S, V - S). According to exercise 23.1-3, e is the light edge through cut(S, V - S). If edge x is in T' and through cut(S, V - S), then x is also a light weight. Because the light edge is unique. So e and x is the same edge, e is also in T'. Because we choose e at random, of all edges in T, also in T'. As a result, the MST is unique.
Counter example: if there are only three nodes, with 2 edges, both of them are weighted 1. Then, we still have a unique MST

(15) 23.2-4 Suppose that all edge weights in a graph are integers in the range from 1 to |V|. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range 1 to W for constant W?

If w is a constant we can use counting sort
- Sorting the edges: O(E lg E) time.
- O(E) operations on a disjoint-set forest taking O(Eα(V)).

The sort dominates and hence the total time is O(E lg E). Sorting using counting sort when the edges fall in the range 1, . . . , |V | yields O(V + E) = O(E) time sorting. The total time is then O(Eα(V)). If the edges fall in the range 1, . . . , W for any constant W we still need to use Ω(E) time for sorting and the total running time cannot be improved further.