HW5 - S15
Warmuth

Justin Wong (jujwong)
5/12/15

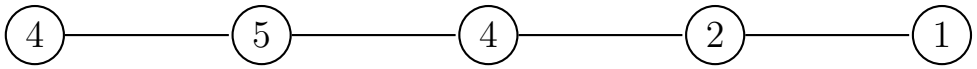
(0) Rabbit Jumping

The recurrence to find the number of ways is $T(n) = T(n-1) + T(n-3)$ when $T(0) = 0$, $T(1) = 1$ and $T(3) = 2$ similar to the Fibonacci Recurrence. The sub-problems are the sums of the hops equaling the desired n and thus the different combinations of hops. Each sub-problem is broken down as a problem that we have solved already. When we solve a recurrence we can store its results in an array and thus reduce the time complexity from exponential to linear by simply calling the already solved answers and not having to traverse the recurrence as far each time. The time complexity with the memoization reduces it from 2^n to $O(n)$ because of the reduced amount of recursion each time.

In the case of $n = 5$ it would be the result of $T(4) + T(3)$ which breaks down further to $T(3)+T(1)+T(2)+T(0)$ which continues down to $2+1+T(1)$ until it becomes $2+1+1 = 4$.

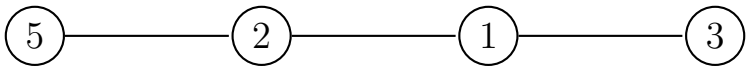
(1) #1 Page 312

(a) Heaviest-First Counter-Example



In this example, the heaviest-first would pick 5 then delete both 4's, and then take 2 next with a total of 7, however the largest independent set is found from taking $4+4+1 = 9$.

(b) Odd-Even Algorithm



The odd-even algorithm will choose odds, taking 1st and 3rd totaling 6. However the largest independent set is found from 1st and 4th being 8.

(c) Dynamic Programming Algorithm

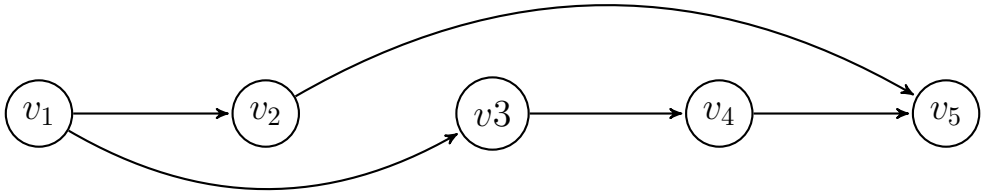
We will define S to be an independent set from v_1, v_2, ... v_i and W_i will be its weight. For any $i > 1$ either it will be chosen to and be a part of S or it won't. If it is chosen then the weight is taken and the next one taken can not be its neighbor.

$$\mathrm{W}_i = \begin{cases} 0, & \text{if } \mathrm{W}_0 = 0 \\ max(\mathrm{W}_{i-1},\, w_i + W_{i-2}), & \text{otherwise} \end{cases} \quad (1)$$

W_i is found from $i = 1$ to n. We are looking to find W_n and the set S can be found through using the what we decide to take from the recurrence. There are n elements and each takes constant time so the time complexity is $O(n)$.

(2) #3, p314

(a) Greedy Algorithm Counter Example



The shortest-first algorithm would have taken the path v_1, v_2, v_5 for a length of 2, however the longest is actually v_1, v_3, v_4, v_5 for a length of 3.

(b) Dynamic Approach

The subproblems will be to find the the longest path from v_1 to v_i. OPT(i) = the longest path from v_1 to v_i. If it is not possible to reach the value at v_i, we will define it -∞. The base case will have OPT(1) = 0 as the path from the first node to itself has 0 edges. Otherwise we will check longest paths with each choice taken, with the number of edges coming into it, and backwards through that.

Create an array of size 1 to n, with the 1st slot being 0. Then iterate through the ordered graph edges with one pointer at the beginning, j, and other at the node i that we want to reach. If the path from j to i is -∞ and if we can move foward in the graph then M[j] + 1. When we have explored all paths then we have M[n] as the longest. Th running time will be $O(n^2)$ because there are n nodes and all edges entering the ith node is found in $O(n)$ time.

(3) #6, p317

We know that the last line will end with the w_nth word, and will have to start with some word w_j, so if we can take out the last line then the smaller sub-problems are choosing from the remaining words w_1,...,w_{j-1}. OPT(i) will be the value of the optimal solution from the set $W_i = \{w_1, ..., w_i\}$ The slack defined as $S_{i,j}$ will have the words w_i to w_j and if the total length of the worlds is longer than L then $S_{i,j} = \infty$. The line w_j to w_n will only be in the optimal solution if the minimum is found from the index at word j. So our recurrence is:

$$\text{OPT(n)} = \min_{1 \le j \le n} (S^2_{j,k} + \text{OPT(j-1)})$$

Then we build up values in a loop. Using the length to build up the $S_{i,j}$ values. Initialize OPT(0) = 0. For $k = 1...n$ OPT(k) = $\min_{1 \le k \le n} (S^2_{j,k} + \text{OPT(j-1)})$. OPT(n) will be our solution.

Computing the $S_{i,j}$ values will take O(n) if we look at them in increasing order, and thus will take a total of O(n^2) time with n elements. Each iteration of the loop will take O(n) time and there are O(n) iterations, so the running time is O(n^2).

(4) #24, p331 Gerrymandering

We will start by taking the difference of a_i - b_i to be equal to f_i so it is the difference between voters for A and B in precinct i. Our goal is to find a way to divide the precincts into equal-sized districts where the sum of the f_i will be positive, meaning a majority throughout the districts.

So we will split the precincts into districts, one half in to one, and the other half in the other. We will be looking for the positive number when we have already decided to add f_n to district 1. Using an array to store our possible sub-problems of what happens when we add precincts to districts.

(5) #28, p334

(a) If we let J be the optimal solution, so by its definition all jobs in J can be scheduled to meet thier deadline. We want to minimize the maximum lateness only considering the jobs in J. So the minimum lateness will have to be 0, since all jobs can be scheduled. The greedy algorithm for minimizing lateness is already optimal, so the ordering jobs in J by their deadline does create an optimal schedule.