Question 1)

Part A)

Assume that we have a graph like figure 6.28. The maximum total weight for that graph should be 14. However, if we run the algorithm using the following scenario, we find out that the algorithm doesn’t find the maximum total weight. Select 1, remove node 1 and 8. 6,3,6 are left. Select left-sided 6, remove 6 and 3. 6 is left. Returning S will give you the sum of 1,6 and 6 will be 13. This is a valid way to move around the graph using the algorithm and it shows that the algorithm doesn’t always produce the maximum total weight.

Part B)

Assume we have a graph like figure 6.28, 1,8,6,3,6 with a maximum total weight of 14. S1 and S2 are independent sets, so that means all the nodes inside each set can’t be connected to each other. Also, S1 has a total of odd numbers inside and S2 has an total of even numbers inside. That means, S1 can only be 1,6 and 6 and S2 can only be 8,3 since none of those numbers touch each other. When you take the sum of both S1 and S2, neither of them produces 14. Since none of them produces 14, it doesn’t show the maximum total weight.

Part C)

Algorithm Idea: Assuming that we have a graph similar to 6.28. We also assume that the definition of an independent set holds true. We first create an empty array S of size n+1 and having it start at index 1 instead of index 0. Now, we create an recursion function called Maxval that has a parameter of i. Within that function, we check if n is equals to either 0 or 1. If n equals to 0, then our max total weights is 0. If n equals to 1, then we just return the weight of that single node. Set S[i] = max value between Maxval(vi-1) and the sum of x and Maxval(vi-2). The recursion returns S[i]. Once it goes through the whole n values, we print out the last value of S.

Algorithm Detail:

Create array S of size n+1

Maxval(i):

If n = 0

Return 0

S[0] = 0

For i = 1, … , n

S[i] = max(Maxval(vi-1),vi + Maxval(vi-2))

Return S[i]

Return S[n+1]

Proof Idea: My algorithm does everything that the problem is asking. It is an algorithm that produces the maximum total weights of an independent set. I will prove that my algorithm runs in polynomial time of n.

Proof Detail: My algorithm looks for the maximum total weights of an independent set. We know that an independent set is when two nodes from a graph are not connected by an edge. For my algorithm, I am assuming that we have a graph that is similar to figure 6.28. If n happens to be either a 0 or 1, we don’t have to do much since the weight can either be 0 for no nodes or the weight of that single node. We also don’t want to have our algorithm start at index 0 of the array since there is no such thing v0. We also use an array for easy access of old data weights. Now, for the recursion. Since we know that each node right after another is connected, they are can’t be put as an independent set. So after we obtain the value of vi, we want to find the maximum weight value between the nodes/sets. So we use the max function, which compares our previous value found using the recursion, Maxval(vi-1) with our current vi plus Maxval(vi-2). The reason we add vi-2 is because there is no edge connecting vi and vi-2. This allows them to be in the same independent set, and since we want to find the maximum total weight of an independent set, we need to add them together. Also, vi-2 contains the total weight values of the previous independent set, while vi-1 contains the current maximum total weights. After comparing the values together, the highest set between the two becomes the owner of the current S[i]. Once we hit the very end, S[n+1], it should contain the maximum total weights.

Runtime Analysis: My algorithm runs in polynomial time of n. First, creating an empty array of n+1 size runs in O(n+1) or O(n). The recursion includes two if statements that runs in O(1), the main part of the recursion, the max function runs in O(n) since we are doing this for all values of n. So my recursion function runtime is O(n). Now, the sum of the whole algorithm becomes O(n) + O(n) = O(2n) = O(n), which is polynomial time of n.

Source: Lecture 34, textbook

Question 2)

Part A)

For this question, we will use the example that was given in problem 2 but we will switch values between week 1 and week 2. So now, Week 1 has the values of the old Week 2 and Week 2 has the values of the old Week 1. Here is a chart to show what I mean:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Week 1 | Week 2 | Week 3 | Week 4 |
| l | 1 | 10 | 10 | 10 |
| h | 50 | 5 | 5 | 1 |

With this, the maximum value becomes 50 + 10 + 10 + 10 = 80. Since we are allowed to choose h during week 1, this works. However, the algorithm doesn’t return this value. If we were to use the given algorithm, it would produce 31 because it doesn’t check the value of h in week 1. It just ignores it. So this is a counterexample of the given algorithm in part a.

Part B)

Algorithm Idea:

Given a chart similar to what I wrote for part a, we will place the values of l and h into two arrays, L and H of size n+1. We will also create a third array S of size n+1. All three arrays will start at index 1 instead of index 0. We now check if n = 0. If so, then return 0. Else, we make an int x = 1 and int y = 0. While x is less than or equal to n, we set x as the max value between L[i] and H[i]. if y == L[i], we set y as the max value between L[i] + L[i+1] and H[i+1]. If y == H[i+1], then we set S[i] = 0 and S[i+1] = y and i increase by 2. If y == sum of L[i] and L[i+1], then S[i] = L[i] and i increase by 1. However, if the first y comparison == H[i], then we set y = max value between H[i] + L[i+1] and H[i+1]. If y == H[i+1], set S[i] = 0, S[i+1] = y and increase i by 2. If y == H[i] + L[i+1] then S[i] = H[i] and i increase by 1. After all n iterations, we return S

Algorithm Detail:

Assume we have a graph similar to part A

Assume everything stated in the problem is true

Place values of l into array L of size n + 1 starting at index 1

Place values of h into array H of size n + 1 starting at index 1

Create array S of size n + 1 starting at index 1

If n == 0

Return 0

Else

int x = 1

int y = 0

while x <= n

y = max(L[i],H[i])

if y == L[i]

y = max((L[i] + L[i+1]),H[i+1])

if y == H[i+1]

set S[i] = 0

set S[i+1] = y

x = x + 2

if y == (L[i] + L[i+1])

set S[i] = L[i]

x = x + 1

if y == H[i]

y = max((H[i] + L[i+1], H[i+1])

if y == H[i+1]

set S[i] = 0

set S[i+1] = y

x = x + 2

if y == (H[i] + L[i+1])

set S[i] = H[i]

if S[i-1] != 0

set S[i-1] = 0

if S[i-2] == 0

S[i-2] = max(H[i-2],L[i-2])

Return sum of S

Proof Idea: My algorithm does everything that the problem is asking. It is an algorithm that takes values for l1-n and h1-n and produces an optimal value plan. I will prove that my algorithm runs in polynomial time of n.

Proof Detail: For my algorithm, I am using a table similar to what I have in part a. My algorithm takes all values of l and h and inputs them into two arrays L and H. This makes accessing data at certain weeks of l and h easier to find. My algorithm also finds the optimal value of the table by comparing l and h for week i. If it finds l to be the largest, then we need to check if the sum of week i and week i+1 for l is greater than week i+1 for h. If h is greater, we have to set the choice for week i to be none since week i+1 will produce a greater value. If l is greater, then we can set week i to a choice. My algorithm also checks each time we select h as our choice. Since h has to have a week for prep, week i-1 must be a 0. We deal with that and then make sure that week i-2 is changed as well because that week could be set to zero in case we choose week i-1 to be h. Once everything has ran through, we return the sum of S as our optimal value.

Runtime Analysis: My algorithm creates 3 arrays of size n+1, so the runtime of each creation is O(n+1) = O(n). If statements are O(1). Since I have a lot if statements in my algorithm, I will just say that the sum of all if statements are O(1). Max functions also run in O(1) since we dealing with only two values and I have a while statement that runs for n times, which is O(n). So in the end, it becomes O(n) + O(n) + O(n) + O(n) = O(4n) = O(n).

Source: Textbook, Lecture 35,36