Assignment2

Libo Liu 202095669

1. **Balancing act**

**a. Algorithm description**

First, we should sort sword and shield by increasing weight and renumber it.

Second, Calculating the absolute value of wight difference between sword and shield in the order above. If this value more than 2, we will compare the weight sword and shield. If the weigh sword more than shield, we will move to next shield. otherwise, we will move to next sword.

If this value is less than 2, this combination is balanced, let the maximum add 1.

Until there is no other sword and other shield to use. The maximum is our final output.

**b. Pseudocode**

Greedy\_balancing(n, w1, w2, …, wn, m, v1, v2, …, vm, k)

SORT sword by increasing weight and renumber so that w1 >= w2 >= … >= wn.

SORT shield by increasing weight and renumber so that v1 >= v2 >= … > =vn.

i 🡨 1

j 🡨 1

sum 🡨 0

WHILE ( i <= n AND j <= m){

IF |wi – vj| > k

IF (wi > vj) j++;

ELSE (vj > wi) i++;

ELSE sum++, i++, j++; }

RETURN sum

**c.** **Example**

Instance: SWORDS = [(a1, 1), (a2, 1), (a3, 6), (a4, 8), (a5, 5)]

SHIELDS = [(b1, 4), (b2, 2), (b3, 4), (b4, 5), (b5, 6)]

Constraints: the weight of the sword and the weight of the shield differ by more than 2 kilos.

Solution: Sorting sword wight and renumber: (a1, 1), (a2, 1), (a3, 5), (a4, 6), (a5, 8)

Sorting shield wight and renumber: (b1, 2), (b2, 4), (b3, 4), (b4, 5), (b5, 6)

After running above programming, we can get balance pair:

(a1, b1), (a3, b2), (a4, b3), (a5, b5)

So, the maximum number of competitors is 4.

**d. a proof of optimal solution**

Pf: By induction on the number of iterations of WHILE loop.

Base case: In S\_0, sum=0. Let S\_opt, sum’= max be an optimal solution. Since sum’ >= sum, S\_opt extends S\_1 using all pairs.

Induction hypothesis: Assume that S\_t = sum is promising. That is, there exists some S\_opt extending S\_t using items{i+1, …n, j+1… }

Induction step: show that S\_t+1 is promising.

If S\_opt extends S\_i+1, done(that is, S\_t+1 has the same value in S\_opt)

If S\_opt extends S\_j+1, done(that is S\_t+1 has the same value in S\_opt )

So S\_t+1. Is promising.

In conclusion, the algorithm is an optimal solution to balancing act.

**e. time complexity**

From pseudocode, “WHILE (i <=n ^ j <=m)” is the main step to have an effect on the running time. In the worst case, the running time is m+n, So the time complexity is O(n).

1. **UN-tiles**
2. **Description**

I used the bottom-up dynamic programming approach. we will reorganize the order in which we solve the subproblems. We will compute F(0), then F(1), and so on. This will allow us to compute the solution to each problem only once, and we’ll only need to save two intermediate results at a time. For example, when we’re trying to find F(2), we only need to have the solutions to F(1) and F(0) available. Similarly, for F(3), we only need to have the solutions to F(2) and F(1).

1. **Step1:** **Deciding the state**

We define our state by one parameters index i.e DP[i]. Here DP[i] tell us the possible to tile form range 0 to max\_tunnel. Obviously

Then , we can determine the boundary value. So. tunnel = 0, DP[0] is Ture. Therefore, here the parameters index can uniquely identify a subproblems for this UN\_tiles.

1. **Step2:** **Formulating a relation among the states**

When we observe state DP[n], here DP[n] means the tunnel is n. Now we need to compute state DP[n]. Let us assume tile size is t. DP[n]=DP[n-t].

In general, DP[n]=DP[n-t]

1. **Example:**

Tiles = {3, 5}

Tunnels= {4, 5, 6, 7, 8, 9}

**Pseudocode:**

Dynamic\_UN (tunnel, tile):

DP[0] = True

For i in range (1, max(tunnel)):

DP[i] = False

For j in range (0, len(tile)):

If i >= tiles(j):

DP[i] = DP[i- tile[j]]

After running, we will get this table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DP[0] | DP[1] | DP[2] | DP[3] | DP[4] | DP[5] | DP[6] | DP[7] | DP[8] | DP[9] |
| Ture | False | False | Ture | Ture | Ture | Ture | False | Ture | Ture |

Finally

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tunnel | 4 | 5 | 6 | 7 | 8 | 9 |
| Result | Ture | Ture | Ture | False | Ture | Ture |

1. **Running time**

We can observe this algorithm exist two for loop. Which is mainly take time. So the time complexity is O(mn).

***Implementation***

Text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

1. **UNnels soundscapes**

1. **Greedy algorithm**
2. Algorithm description

From question introduction, we can get this problem belongs to independent set on the trees. So, given a tree, find a maximum cardinality subset of nodes such that no two share an edge. If we use greedy algorithm, we can observe a tree on at least two nodes has at least two leaf nodes. Therefore, if v is a leaf, there exists a maximum size independent set containing v.

1. **Pseudocode**

Greedy\_max\_marimba(Tree\_map){

S 🡨 ø

While ( T\_map has at least one edge) {

Let e = (u, v) be an edge such that v is a leaf

S 🡨S U {v}

Delete vertices u and v from Tree\_map, and all edges related to them.

}

Return S U {isolated vertices in Tree\_map}

}

1. **Example**

Chart, shape

Description automatically generated

Step1 S= {V5, V6, V10, V11, V8, V9}

Step2 Delete V2, V3, V7, V4 and <2,5>,<2,6>,<7,10>,<7,11>,<3,8>,<4,9>

Step3 Return S U {V1 is isolated node}

The red nodes is a maximum number of intersection to place marimba.

Diagram

Description automatically generated

**d. a proof of optimal solution**

Pf: By induction on the number of iterations of WHILE loop.

Base case: In M\_0, max=0. Let M\_opt, s’= max be an optimal solution. Since sum’ >= sum, M\_opt extends M\_0 in independent set.

Induction hypothesis: Assume that M\_t = sum is promising. That is, there exists some S=M\_opt extending M\_t in tree degree {i+1, …n}

Induction step: show that M\_t+1 is promising.

If M\_opt extends M\_i+1, done(that is, M\_t+1 has the same value in M\_opt)

If M\_opt extends M\_j+1, done(that is M\_t+1 has the same value in M\_opt )

So S=M\_t+1. Is promising.

In conclusion, the algorithm is an optimal solution to this problem.

**e. Time complexity**

It can implement in O(n) time by considering nodes in postorder.

1. **Dynamic programming**
2. **Algorithm description**

If renovated intersections are of different sizes, and they want wo maximize total space hosting marimbas. This means that the problem become a weighted independent set on trees. Given a tree and node weights w>0, find an independent set S that maximizes Σv∈S Wv. So, if (u, v) is an edge such that v is a leaf node, then either DP includes u or DP includes all leaf nodes incident to u.

**b．step1: Deciding the state**

DP1[u] = max weight independent set of subtree rooted at u, containing u.

DP2[u] = max weight independent set of subtree rooted at u, not containing u.

**Step2: Determine the boundary value**

DP1[u] = Wu

DP2[u] = 0

**Step3: Formulating a relation among the states**

DP1[u] = Wu + DP2[v]

DP2[u] = max { DP1[v], DP2[v]}

Diagram

Description automatically generated children(u) = {v,w,x}

c **. pseudocode**

Dynamic\_weighted\_max\_marimba(Tree\_map){

Root the tree at a node r

Foreach (node u of Tree\_map in postorder) {

If (u is a leaf) {

DP1[u] = Wu

DP2[u]= 0

}

else{

DP1[u] = Wu + DP2[v]

DP2[u] = max { DP1[v], DP2[v] }

}

}

Return max(DP1[r], DP2[r])

}

1. **Example**

A picture containing text, clock, watch

Description automatically generated

Step1: postorder:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | r |
| I4 | I6 | I7 | I5 | I2 | I3 | I1 |

Step2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| DP1[0] =5 | DP1[1] =1 | DP1[2] =2 | DP1[3] =4 | DP1[4] =6 | DP1[5] =2 | DP1[6] =11 |
| DP2[0] =0 | DP1[1] =0 | DP1[2] =0 | DP1[3] =3 | DP1[4] =9 | DP1[5] =0 | DP1[6] =8 |

Step3: max{DP1[r], DP2[r]} = 11

Finally, 11 is the maximum total space to put marimbas.

1. **Time complexity**

The dynamic programming algorithm finds a maximum weighted independent set in a tree in O(n) time.