CSE 101 Homework 7

Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

December 4, 2025

1. **Bus routes**

I am going by bus across town, along a path that includes bus stops $1...m$. There are n different bus routes BR_1 $...BR_n$, where BR_i goes along my route between stop s_i to stop f_i, and costs c_i to ride. I want to go from stop 1 to stop m paying the smallest total amount as I can. You can assume there will always be a later bus for each route passing each stop.

$BTBusCosts(BR_1..BR_n; current; m)$

(a) IF $current \geq m$ return 0.

(b) $Best =$ infinity;

(c) FOR $I = 1$ to n do:

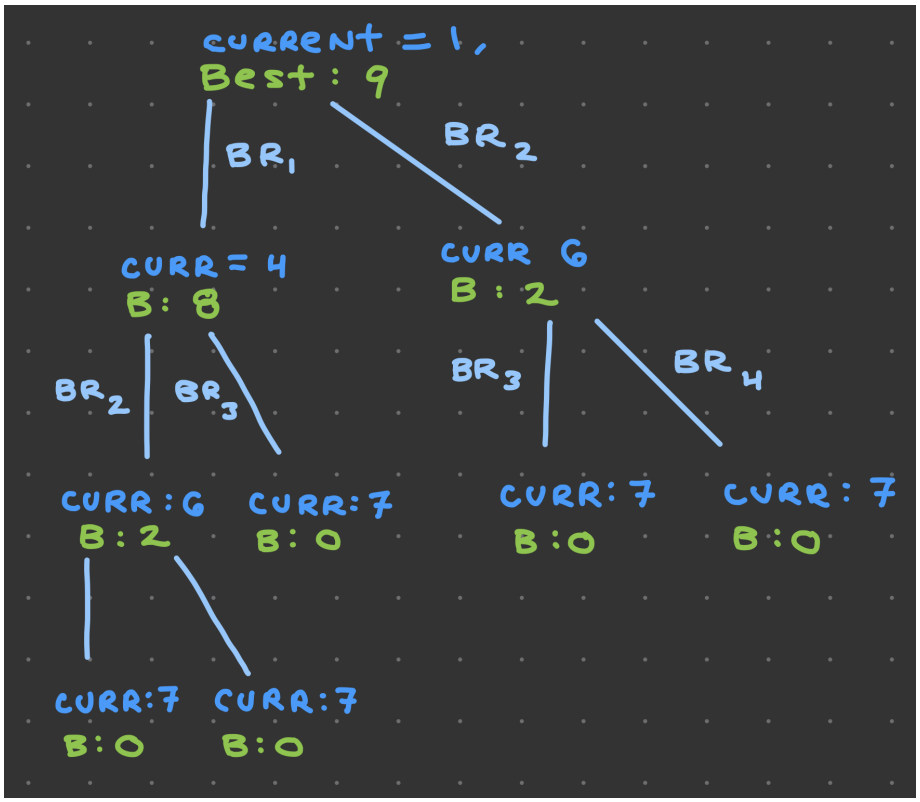
(d) IF $s_i \leq current < f_i$

(e) THEN $Best = min(Best, c_i + BTBusCosts(BR_1...BR_n; f_i; m)$.

(f) Return $Best$

Questions

(a) Give the tree of recursive calls on the instance $current = 1, m = 7$, $BR_1 = (1, 4, 3), BR_2 = (1, 6, 7), BR_3 = (2, 7, 8), BR_4 = (5, 7, 2)$.



(b) Give an upper bound for the total number of recursive calls for this algorithm.

Note that the worst case has bus routes of the following form, since it maximizes the number of recursive calls at each level.

$$\begin{aligned} BR_1 &= (1,2) \\ BR_2 &= (1,3) \\ &\vdots \\ BR_n &= (1,n+1) \end{aligned}$$

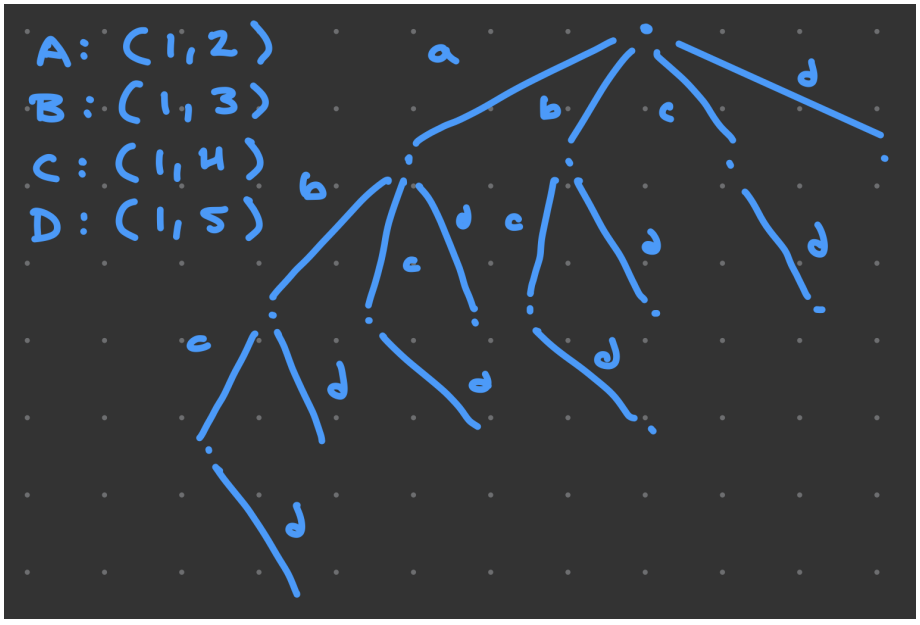
Let x be the distance from current position to position m. Then $T(x)$ is the number of recursive calls at a distance x to the finish:

$$\begin{aligned} T(x) &= T(x-1) + T(x-2) + ...T(1) \\ &\leq x \cdot T(x-1) \text{ Since, } T(x-i-1) \leq T(x-i) \forall i \geq 1 \end{aligned}$$

Solving the recurrence relation with base case $T(1) = 1$

$$\begin{aligned} T(x) &\leq x \cdot T(x-1) \\ &\vdots \\ &\leq (x)(x-1)\cdots T(1) \\ &\leq x! \end{aligned}$$

Supplemental Diagram:



(c) Which variables change during this recursion?
Only the value of *current* changes each iteration.

(d) Define sub-problems: Use the above to define a set of sub-problems that can be used to convert the BT algorithm to a DP algorithm, and define an array or matrix to hold the answers to these subproblems.

- Each subproblem starts at some $i \in 1, ...m$ and returns the lowest cost path from i to m. Each operates with the same bus stops and works towards the same final point m. The total number of distinct subproblems is m.
- The solutions to subproblems can be stored in a length m array, where each position is the cost to get from position i to m

(e) What are the base cases?

The singular base case is $i = m$, the subproblem where you start at the end. The cost is 0.

(f) Translate the BT algorithm into a definition of array/matrix values from other values

On iteration k, $k \in \{1, ...m\}$:

for BR_i such that $s_i \leq k < f_i$:

$costs[k] = min(costs[k], c_i + costs[f_i])$

(g) In what order should we fill this array or matrix?

This matrix will be filled from the last position $k = m$ to the first position $k = 1$

(h) Assemble these pieces into a DP algorithm.

```
def bus_stops(BR1...BRn, curr, m):

    // there may not be a valid path from every point i to m,
    // so fill all spots except the ending
    cost = [ infinity ]
    cost[m] = 0

    // go from the end of the route to the start
    for k from m - 1 to 1:

        // si, fi, ci represent the start, finish, and cost of a bus route
        // they are defined and accessible from outside the function
        for all BRi in BR1...BRn such that si <= k < fi:
            cost[k] = min(cost[k], ci + cost[fi])

    return cost[1]
```

(i) Give a time analysis for your DP algorithm.

- Assuming accessing costs array is $O(1)$ then the main loop $\in O(m) \cdot O(1) \in O(m)$

- Assume finding all relevant BR_i can be done in worst case $\in O(n)$

Therefore, the total runtime of the algorithm is $\in O(n \cdot m)$

(j) Give the array or matrix for your DP algorithm on the given example.

2. **Bounded difference sequence**

We say a sequence of numbers $a_1, ..a_n$ is K-bounded difference if $|a_{i+1} - a_i| \leq K$ for all $1 \leq i \leq n-1$. The maximum bounded difference subsequence problem is, given sequence $a_1, ..a_n$, and a real number $K > 0$, decide the length ℓ of the longest subsequences $a_{i_1}, ..a_{i_\ell}$, $1 = i_1 < i_2 < ...i_\ell$ including position one, which is K-bounded difference.

For example, if $a[1..5] = 2, 7, 1, 4, 3$ and $K = 2$, $2, 1, 3$ is a maximum length K bounded subsequence, as is $2, 4, 3$.

(a) **Description of sub-problems**

Let $L[i]$ be the length of the longest K-bounded difference subsequence that ends at index i and specifically includes position 1 (i.e., the subsequence must start at a_1).

Then, there are n distinct subproblems, each running on a prefix $A_1, ...A_i$

(b) **Base Case(s)**

$$L[1] = 1$$

The subsequence ending at index 1 is simply the single-element sequence $\{a_1\}$, which has a length of 1.

(c) **Recursive definition of answers (with justification)**

For $i > 1$:

$$L[i] = 1 + \max\left(\{L[j] \mid 1 \leq j < i \text{ AND } |a_i - a_j| \leq K \text{ AND } L[j] > 0\} \cup \{0\}\right)$$

Justification: To find the longest valid subsequence ending at a_i, we iterate through all previous indices $j < i$. We check two conditions:

i. $L[j] > 0$ (this ensures the sequence ending at j traces back to index 1).

ii. $|a_i - a_j| \leq K$.

If these conditions are met, we can extend the sequence from j by appending a_i. If no such j exists, $L[i]$ remains 0, indicating the chain is broken. We take the maximum to ensure optimality.

(d) **Order in which sub-problems are solved**

We solve for $L[i]$ in increasing order of i, from $i = 2$ to n. This ensures that when computing $L[i]$, the values for all potential predecessors $L[1], \ldots, L[i-1]$ are already available.

(e) **Form of output**

The final answer is the maximum value in the array L:

$$\text{Result} = \max_{1 \leq i \leq n}(L[i])$$

(f) **Pseudocode**

```
1: function LongestBoundedSubsequence(A, n, K)
2:     Let L be an array of size n + 1 initialized to 0
3:     L[1] ← 1
4:     max_length ← 1
5:     for i ← 2 to n do
6:         for j ← 1 to i − 1 do
7:             diff ← |A[i] − A[j]|
8:             if L[j] > 0 and diff ≤ K then
9:                 L[i] ← max(L[i], L[j] + 1)
10:            end if
11:        end for
12:        if L[i] > max_length then
13:            max_length ← L[i]
14:        end if
15:    end for
16:    return max_length
17: end function
```

(g) **Runtime analysis**

- Time complexity: $O(n^2)$. The algorithm uses two nested loops. The outer loop runs n times, and the inner loop runs i times. The total operations are proportional to $\sum_{i=1}^{n} i \approx \frac{n^2}{2}$.
- Space complexity: $O(n)$. We utilize a single array L of size n to store the sub-problem solutions.

(h) **A small example explained**

Input: $A = [2, 7, 1, 4, 3]$, $K = 2$. Initialize $L = [0, 0, 0, 0, 0]$.

- $i = 1$: Base case $L[1] = 1$. $L = [1, 0, 0, 0, 0]$.
- $i = 2$ ($val = 7$): Compare with $j = 1$ ($val = 2$). $|7 - 2| = 5 > K$. No update. $L[2] = 0$.
- $i = 3$ ($val = 1$): Compare with $j = 1$. $|1 - 2| = 1 \leq K$. Update $L[3] = 1 + 1 = 2$. $L = [1, 0, 2, 0, 0]$.
- $i = 4$ ($val = 4$):
  - Compare $j = 1$: $|4 - 2| = 2 \leq K \rightarrow L[4] = 2$.
  - Compare $j = 3$: $|4 - 1| = 3 > K \rightarrow$ No update.

  $L = [1, 0, 2, 2, 0]$.
- $i = 5$ ($val = 3$):
  - Compare $j = 1$: $|3 - 2| = 1 \leq K \rightarrow L[5] = 2$.
  - Compare $j = 3$: $|3 - 1| = 2 \leq K \rightarrow L[5] = \max(2, L[3] + 1) = 3$.
  - Compare $j = 4$: $|3 - 4| = 1 \leq K \rightarrow L[5] = \max(3, L[4] + 1) = 3$.

  Final $L = [1, 0, 2, 2, 3]$.

Result: The maximum value in L is **3**.

3. **Minimum weight connected subtree of given size**

You are given a binary tree of size n, where every vertex x has pointers $lc.x$, and $rc.x$ (which could be NIL if those children don't exist) and each vertex has a value, $value.x > 0$. You are also given an integer $1 \leq k \leq n$. You want to find the connected sub-tree with exactly k vertices, which minimizes the total weight of vertices in the sub-tree.

(a) Characterize Sub Problems

Find the size and weight of every subtree within the initial tree. If the tree has k nodes, compare the weight of that tree to the running minimum.

The set of all distinct subproblems, S, = the subtrees starting at every distinct node $v \in V$. This implies there are $|S| = |V| = n$ distinct subproblems

(b) Base Case

The root of the subtree has no children (the subtree is 1 node).

- $size = 1$
- $weight = value.v$

(c) Recursive Definition of Answers

If $|v.children| = 0$:

$$\begin{aligned} weight[v] &= value.v \\ size[v] &= 1 \end{aligned}$$

If $|v.children| = 1$:

$$\begin{aligned} weight[v] &= value.v + weight[v.child] \\ size[v] &= 1 + size[v.child] \end{aligned}$$

else:

$$\begin{aligned} weight[v] &= value.v + weight[lc.v] + weight[rc.v] \\ size[v] &= 1 + size[lc.v] + size[rc.v] \end{aligned}$$

Brief Justification: For each subtree subproblem, there are 3 possibilities:

- The root of the subtree has no children. In this case the size of the subtree is just 1, and its weight is just the weight of its root. There is no recursive relationship, and thus the return values do not depend on the memoized array.

- The root of the subtree has 1 child. In this case the size of the subtree is 1 + the size of the root's only child, and the weight it the weight of root + the weight of its child. In this case, the current solution depends on the return values of its only child, thus the call to $weight[v.child]$ and $size[v.child]$

- The root of the subtree has 2 children. In this case the size of the subtree is 1 + the size of each of the subtrees rooted at its children, and its weight is the weight of the root + the weight of those subtreets. The solution depends on the return values of both the left and right child.

Therefore, the above recurrence covers the only 3 possibly cases for each subtree in the graph.

(d) Subproblem Order

The subproblem originating at node v must come after the subproblems originating at its children. Therefore, subproblems must be solved in a post-order traversal (both children before their parents.)

(e) Form of Output

While calculating the weight of each subtree, keep track of the smallest weight. For those with node count $= k$, check if its weight it lower than the current smallest. After scanning through all subtrees, return the smallest.

Note: Alternatively, keep track of node v where the smallest subtree occurs to return not just the smallest weight, but the root of the subtree too.

(f) Pseudocode

```
def MWCS(root, k) -> smallest:

// if a subtree does not contain k nodes (ie. does not satisfy constraints of
    potential solution)
// return infinity
smallest = infinity

for v in (post-order traverse root):
    if |v.children| == 0:
        weight[v] = v.value
        size[v] = 1

    if |v.children| == 1:
        weight[v] = v.value + weight[v.child]
        size[v] = 1 + size[v.child]

    else:
        weight[v] = v.value + weight[lc.v] + weight[rc.v]
        size[v] = 1 + size[lc.v] + size[rc.v]

    // attempt to update the variable for subtrees satisfying problem
        constraints
    if size[v] == k:
        smallest = min(smallest, weight[v])
```

```
    return smallest
```

(g) Runtime Analysis

- Post-traversal $\in O(n)$
- Runtime for each node: $\in O(1)$

Thus, total runtime $\in O(n)$

(h) Small Example

4. **Road trip**

You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

(a) Sub-problems:

- Each subproblem represents the next hotels we can stop at after our current hotel.
- Each subproblem has some $a_i, ...a_n$ and returns the lowest cost path from i to n. Each subproblem contains the same set of hotels.
- The solutions to each subproblem can be stored in an array, $TravelPenalty[]$, of length n, where $TravelPenalty[i]$ = the minimum penalty to reach this hotel; this penalty is the minimum sum for all travel days taken after hotel a_i.

(b) Base Case:

- The base case is when $i == n$, i.e. the subproblem where the start position is the final hotel. $TravelPenalty[n] = 0$

(c) Recursive definition:

- If we are not at our final destination (the base case), our next hotel stop after a_i is some a_j. Because we can only travel forwards, we know $i < j \leq n$. If we stop at a_j, the penalty for this travel day is stored in $TravelPenalty[j] = (200 - (a_i - a_j))^2$. To find the penalty for i: $TravelPenalty[i] = min_{i+1 \leq j < n}(200 - (a_i - a_j))^2 + TravelPenalty[j]$.

(d) Order in which sub-problems are solved:

- Because we can only travel forwards, we need to fill the array in reverse order i.e. starting at $i = n - 1$ to $i = 1$.

(e) Form of output (how do we get the final answer?):

- The final answer is given by $TravelPenalty[0]$ as we fill the array in reverse order, so this will be the minimum penalty to travel n miles to the final destination a_n.

(f) Pseudocode:

- $a_0 = 0$.
- Initialize $TravelPenalty[0..n]$.
- $TravelPenalty[n] = 0$
- FOR $I = n - 1$ down to 1 do:
- $\quad MinPenalty = \infty$
- $\quad$ FOR $J = I + 1$ to n do:
- $\quad\quad MinPenalty = min(MinPenalty, (200 - a_I + a_J)^2 + TravelPenalty[J])$.

- $TravelPenalty[I] = MinPenalty$
- Return $TravelPenalty[0]$.

(g) Runtime analysis:

- There are two nested for loops, so the runtime is quadratic $O(n^2)$.

(h) Small example:

Let there be a 500 mile road trip. There are hotels at miles [25,150,400,500]. The final stop is mile 500. To find the minimum travel penalty:

- $TravelPen[4] = 0$
- $TravelPen[3] = (200 - (500 - 400))^2 + TravelPen[4] = 10000 + 0 = 10000$
- $TravelPen[2] = min((200-25)^2+TravelPen[3], (200-(500-150))^2+TravelPen[4]) = min(30625 + 10000, 22500) = 22500$
- $TravelPen[1] = min((200-(150-25))^2+TravelPen[2]+TravelPen[3]), (200-(500-25))^2) + TravelPen[4] = min(5625 + 22500 + 10000, 75625) = 38125$
- $TravelPen[0] = min((200 - 25)^2 + TravelPen[1], (200 - 150)^2 + TravelPen[2], (200 - 400)^2+TravelPen[3], (200-500)^2+TravelPen[4] = min(68750, 25000, 50000, 90000)) = 25000$. The route with the lowest penalty is to stop at hotel 2 (mile 150) and hotel 3 (mile 400). The penalty for the first travel day is 1500, and the penalty for the second travel day is 22500.

5. **Knapsack variant**

Consider the variant of knapsack where you are allowed to choose items more than once. You still have n items each with values v_i and costs c_i, and a budget U. If you pick item i p_i times, you must have $\sum_{1 \leq i \leq n} p_i c_i \leq U$, and your objective is to maximize $\sum_{1 \leq i \leq n} p_i v_i$. Each p_i must be a non-negative integer.

(a) Characterize Subproblem

Let $I_1, ..I_n$ be the set of objects ordered by non-increasing value of cost ratio: v_i/c_i

Each subproblem is given some suffix of this array with the budget leftover after attempting to maximize the selection of all higher valued $I_1, I_2, ...$ For example, subproblem j is given the set of object $(I_j, ..I_n)$ and budget $U - \sum_{1 \leq i < j} p_i \cdot c_i$

Then, the set of subproblems is all subsets $u_j, ...u_n | 1 \leq j \leq n$ and the empty set. Therefore the set of all unique subproblems, S, $\implies |S| = n + 1$

(b) Base Case

$U = 0 \implies p_j, ...p_n = 0$

(c) Recursive Definition

Let the remaining budget after selecting p_j of item I_j be stored in a list U':

$$\begin{aligned} P[i] &= \lfloor U'[i-1]/c_i \rfloor \\ V[i] &= V[i-1] + P[i] \cdot v_i \\ U'[i] &= U'[i-1] - P[i] \cdot c_i \end{aligned}$$

Brief Explanation of Correctness:

The above definition uses the fact that the best solution always starts by trying to maximize the items with the best value / cost ratio. Whatever budget is leftover after maximizing item I_j should try to be spent on maximizing the next best item, item I_{j+1}. Specifically:

- p_j is just the most of item I_j we can currently afford with the leftover budget $(U'[j-i])$
- the total value after picking all items I_j is the previous value $(V[j-1])$ + the new value from item j $(p_j * v_j)$
- The new leftover budget is the old leftover budget $(U'[j-1])$ - the cost of item j $(p_j * c_j)$

(d) Solution Order:

Start at the most valuable and iterate to the least valuable until the budget is completely used or there are no more items.

(e) Form of Output Either return $V[n]$ for the maximized value, or return P for the total amounts of each item.

(f) Pseudocode

```
def KV(I1,...In, U):

sort (I1,...In) by the value ratio of each item

// create memoized arrays
U' = [infinity]; U'[0] = U
V = [0]
P = [0]

// loop through each item
for Ii in I1,...In:

    // if the previous item fully depleted the budget, all remaining item
        counts are 0
    // and the function should return
    if U'[i - 1] == 0:
        break

    P[i] = floor( U'[i-1] / ci )
    V[i] = V[i - 1] + P[i] * vi
    U'[i] = U'[i - 1] - P[i] * ci

return P
```

(g) Time Analysis

- Sorting $\in O(n \cdot logn)$
- Loop: $\in O(n) \cdot O(1) \in O(n)$

Therefore, the total runtime is $O(n \cdot logn)$