Luis Gascon, Ethan Webb, Femi Dosumu
COSC 336
April 11, 2023

Assignment 5

Similar to the longest increasing subsequence problem, we use dynamic programming to solve the problem more efficiently by minimizing the number of calculations we have to perform. We approach this problem bottom up as it this problem has a case of overlapping subproblems. We store the solutions to each subproblem to an array called `s[]`. We also have to find a way of returning a subsequence that sums up to the maximum sum, so to achieve that solution, we'll have to keep track of the elements, specifically, their indices as the we iterate through the sequence.

We iterate the sequence with an inner loop with the j index as inner and the i index as outer where $i > j$

To populate `s[]`:

```
if arr[i] >= arr[j]:
  s[i] = max(s[i], s[j]+arr[i])
```

In order to get the elements of the sequence that sum up to the maximum sum of increasing subsequence, we have to create another sequence called `p[]` that keeps track of indices where the addition happens.

To populate `p[]`:

```
if s[i] < s[j] + s[i]:
  p[i] = j
```

To have our algorithm return the maximum sum and its respective subsequence, we created a class called `Outputs`, which has the attributes `sum` and `sub_sequence`. To construct the subsequence, we create a function called `consruct_subsequence()` which has the parameters `sequence, N - 1` and `p` that returns the subsequence that sum up to the maximum sum.

```
static List<Integer> construct_subsequence(List<Integer> arr,
 int N, List<Integer> p) {
  // Return value
  List<Integer> subsequence = new ArrayList<>();
  // indices[] stores the target indices of the subsequence
  List<Integer> indices = new ArrayList<>();
  indices.add(N);
  get_predecessor_indices(N, p, indices);
  indices.forEach(x -> subsequence.add(arr.get(x)));
  // Reverse the list in place for an increasing subsequence
  Collections.reverse(subsequence);
  return subsequence;
}
```

To backtrack from the index of the maximum sum to the index of the subproblem solutions, we create a recursive function called `get_predecessor_indices(index, p, and sum_index)` where the base case is when the return value is the same as the value in `p`.

```
static int get_predecessor_indices(int index, List<Integer> p,
 List<Integer> sum_index) {

  // Base case
  if (p.get(index) == index)
    return index;

  sum_index.add(p.get(index));
  return get_predecessor_indices(p.get(index), p, sum_index);
}
```

The algorithm's complete code:

```
static Outputs max_sum(int N, List<Integer> sequence) {
    // s[] stores the sums at each index i. Initialized with
   values of sequence
    // p[] stores the indices of the elements that sum up to the
   maximum sum. Initialized with values from range of 0 to N - 1
    List<Integer> s = new ArrayList<>(sequence);
    List<Integer> p = new ArrayList<>();

    for (int index = 0; index < N; index++)
      p.add(index);

    for (int i = 1; i < N; i++)
      for (int j = 0; j < i; j++) {
        if (sequence.get(i) >= sequence.get(j)) {
          // Populate p[]
          if (s.get(i) < s.get(j) + sequence.get(i)) {
            p.set(i, j);
          }
          // Populate s[]
          s.set(i, Math.max(s.get(i), s.get(j) + sequence.get(i))
   );
        }
      }

    return new Outputs(s.stream().max((a, b) -> a - b).get(),
   construct_subsequence(sequence, N - 1, p));
  }
```

Solutions obtained

Input	Sum	Subsequence
input-5.3	130,021	[4601, 20255, 23073, 32092, 50000]
input-5.4	143,418	[25197, 26355, 29960, 30953, 30953]