CS 577: HW 2

Daniel Ko

Summer 2020

§1 Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called optimal.

§1.1 Write the iterative version of the algorithm to find the maximal size.

We will assume that |coordinate of n| $\leq n$ and are using zero based numbering for lists.

```
Data: crd, a list of coordinates that corresponds to events.
Result: Maximal amount of events we can view, given that we must view the last event.
def optimal(crd):
    #dp holds the maximum amount of events you can view given you visit crd[i]
    dp ← a list of 0's with the same length of crd
    dp[n-1] = 1
    for int i = length of event - 2; i ≥ 0; i–:
        possibleEvents ← a empty list
        for int j = length of event - 1; i < j; j–:
            if abs(crd[j] - crd[i]) ≤ j - i and i + 1 - abs(crd[i]) ≥ 0:
                add crd[j] to possibleEvents
        if possibleEvents is not empty:
            dp[i] = 1 + max element of possibleEvents
    return max element of dp
```

§1.2 Show the algorithm for tracing the events selected.

```
Data: dp, the list used for memoization
      crd, a list of coordinates that corresponds to events
      max, the maximal amount of events we can view
Result: List coordiantes of events to view.
def events(dp, crd, max):
    view ← a empty list
    for each num in dp:
        find the right most max and add its index to view
        decrement max
    return view
```

§1.3 Give a brief argument of correctness.

$$\text{MaxEvent}(i) = \begin{cases} 1 & \textbf{when } i = n-1 \\ 1 + \max(\text{MaxEvent}[j_1], \text{MaxEvent}[j_2], \cdots) & \textbf{if } \exists j > i \text{ s.t. } \mid \text{MaxEvent}[i] - \text{MaxEvent}[j] \mid \leq i - j \\ & \text{and } i + 1 - \mid \text{MaxEvent}[i] \mid \geq 0 \\ 0 & \textbf{otherwise} \end{cases}$$

Note that MaxEvent$[n-1]$ corresponds to the nth event because of our zero based numbering for lists. We compute the recurrence relation descending, i.e. from $i = n-1$ to $i = 0$.

Theorem

The recurrence relation, MaxEvent(i) is most amount of events you can visit given that the first event you visit is at i and you end on nth event.

Proof. We show by strong backwards induction that our algorithm is correct. Let $P(i)$ be the predicate, "MaxEvent correctly computes the most amount of events you can visit given that the first event you visit is at i and you end on nth event". We define $i \in \mathbb{N}$.
Base Case: When $i = n-1$, MaxEvent returns 1 which is the correct output because if you start at the nth event, the last event, you can only visit one event. Hence, $P(n-1)$ holds.
Inductive step: Suppose $P(\alpha)$ holds for all $\alpha \leq n-1$. We show that $P(\alpha - 1)$ holds. Our recurrence relation looks for all $j > i = \alpha - 1$ such that if you start at event i, you can visit event j. This part is given by,

$$\exists j > i \text{ s.t. } \mid \text{MaxEvent}[i] - \text{MaxEvent}[j] \mid \leq i - j$$

The second condition of the recurrence checks if it is possible to view event i, given you start at coordinate 0 and can move 1 distance before the first event, i.e the second assumption given in the homework. This part is given by,

$$i + 1 - \mid \text{MaxEvent}[i] \mid \geq 0$$

If both conditions hold, the most amount of events you can visit is the event at i, which is 1 event, plus the biggest MaxEvent from the j's. By our strong backwards inductive hypothesis, MaxEvent(j) returns the correct amount of events because we defined $j > i = \alpha - 1 \Leftrightarrow j \geq \alpha$. This will give us the correct amount of events you can visit in the case where we can visit i and j.

In all other cases, we cannot visit i or j which means the max amount of events we can visit is 0.

By strong backwards induction, we have proven that $P(i)$ is true for all $i \in \mathbb{N}$. Therefore, MaxEvent is correct. □

Theorem

The algorithm, optimal, returns the maximal amount of events we can view, given that we must view the last event.

Proof. Our algorithm evaluates the recurrence MaxEvent(i) bottom-up by storing MaxEvent(i) in dp[i]. All values for subproblems referenced by the recurrence for MaxEvent(i) will have already been computed because we evaluate dp[i] as i decreases from n-1 to 0. After all the values in dp are computed, the algorithm returns the max element, which corresponds to the most amount of events you can visit somewhere in the list crd, which in turn is the most amount of events you can visit in the entire list crd. □

§1.4 Analyze the running time.

We claim that the running time is $O(n^2)$.

Proof. We have a nested for loop, which run at most n times each. In the outer for loop, we search for the max element of an array which at most is size n. The rest of the computations in the outer for loop take constant time. Additionaly, after all the loops have been run, we search for the max element of dp which is size n. This leads to a total time complexity of $n(2n) + n = 2n^2 + n$. Thus, our total time complexity of our algorithm is $O(n^2)$ □

§2 Design an $O(n^3)$ algorithm using dynamic programming methodology to find an optimal (least total testing cost) assembly order.

§2.1 Use iterative implementation for the algorithm to find the optimal cost

```
Data: arr, the $nxn$ matrix representing the grid of boxes.
Result: A peak.
def findLarge(arr):
    max ← max element of boundary elements, middle row, and middle column
    # Comparison is vacuously true for the neighbors that do not exist
    if max > max's neighbors:
        return max
    # New submatrix does not contain the rows/columns used to find max
    recursive_arr ← the submatrix of the quadrant containing the largest neighbor
    return findLarge(recursive_arr)
```

§2.2 Show the algorithm for finding the optimal order.

§2.3 Give a brief argument of correctness.

Proof. We show by strong induction that our algorithm is correct. Let $P(n)$ be the predicate, "findLarge halts and returns a peak in a nxn matrix". We define $n \in \mathbb{N}^+$.

Base Case: When $n = 1$, findLarge halts the only element in the matrix, which by definition is larger than its neighbors, a peak. Hence, $P(1)$ holds.

Theorem

A peak exists in any given matrix where its elements are unique integers.

Proof. Let the set containing all the unique integers in the matrix be $S = \{a_1, \cdots, a_\alpha\}$. Take set $S' = \{b_1, b_2, \cdots\}$ to be the upper bound of S, where the upper bound is defined as $\forall a_i \in S \quad \forall b_j \in S', a_i \leq b_j$. Using the well ordering principle, there must exist as smallest element $b \in S'$. By definition, this must be the greatest element $a \in S$. Hence, there exist a largest integer in a finite set of unique integers. By definition, this largest integer is a peak. □

Corollary

There exists a peak in the submatrix of the quadrant containing the largest neighbor.

Proof. By previous theorem, there must exist a peak in the submatrix of the quadrant containing the largest neighbor because any subset of the original matrix containing unique elements, will have unique elements. Consider the special case where the peak is on the edge of the new submatrix. Let α be the largest neighbor in the original matrix and γ be the max and peak on the edge. We know that $\gamma > \alpha$ because when we fix a max, by definition it is the greatest integer in the boundary elements, middle row, and middle column, which includes α. This directly implies that γ is greater than all its neighbors within the original matrix because of the fact that α was the greatest integer of the values that directly border the submatrix. □

Inductive step: Suppose $P(n)$ holds for all $n \leq k \in \mathbb{N}^+$. We show that $P(k+1)$ holds.

We observe that findLarge recursively calls with an matrix of size $\lfloor (k-1)/2 \rfloor x \lfloor (k-1)/2 \rfloor$. Since $\lfloor (k-1)/2 \rfloor \leq k$, we know by our strong inductive hypothesis that this recursive call must halt. Hence, findLarge halts with a matrix of size $(k+1)x(k+1)$.

If a peak does not exist in the boundary elements, middle row, and middle column, findLarge recursively calls the submatrix with the biggest neighbor. By the corollary above, there exist a peak in this submatrix. Since the size of the submatrix is $\lfloor (k-1)/2 \rfloor x \lfloor (k-1)/2 \rfloor$, and $\lfloor (k-1)/2 \rfloor < k$, by our strong inductive hypothesis this recursive call returns a valid peak.

By strong induction, we have proven that $P(n)$ is true for all $n \in \mathbb{N}^+$. Therefore, findLarge is correct. □

§2.4 Analyze the running time.

We claim that $T(n^2) = T((n/2)^2) + \Theta(6(n/2))$ which is $\Theta(n)$.

Proof. Suppose we have a size nxn matrix. Each recursive call will have a input size being one quarter of original matrix, which accounts for $T((n/2)^2)$. Additionally, when searching for the max element, we must search the boundary elements, middle row, and middle column, which would require, 6 times half the new input size i.e. $\Theta(6(n/2))$. The rest of the computations take constant time. Let $m = n^2$

$$T(n^2) = T((n/2)^2) + \Theta(6(n/2))$$
$$T(m) = T(m/4) + \Theta(3\sqrt{m})$$

Using the master theorem, it is clear that we get values of $a = 1, b = 4, d = \frac{1}{2}$ for this particular recurrence which results in a complexity of $\Theta(\sqrt{m}) = \Theta(n)$. □