COSC 3320 Practice Problems Set 2

October 16, 2023

Problem 1

Given an array of *distinct* integers A, describe an algorithm to print out all permutations of the array.

Solution. Because there are are $n!$ permutations of a list of n distinct objects and we must consider all of them, a $\Omega(n!)$ algorithm is sufficient, and thus backtracking is ideal.

A backtracking solution would try, in turn, every possible object as the first, then recursively generate all permutations of the remaining objects. In pseudocode:

```
function PERMUTATIONS($A$)
    initialize result to $\emptyset$
    for i $\in A$:
        set $B$ to $A$ with i removed
        for P $\in$ PERMUTATIONS($B$):
            add [i] + P to result
    return result
```

The runtime complexity of this algorithm is described by the recurrence $T(n) = nT(n-1) + \mathcal{O}(1)$, which can be shown to run $\mathcal{O}(n!)$ time. □

Problem 2

Given an array of *possibly non-distinct* integers A, describe an algorithm to print out all *unique* permutations of the array.

Solution. One method would be to adapt the previous algorithm to this problem. Instead of choosing at each step from n distinct objects, we choose from the (at most n) unique objects, and recursively find the permutations of the remaining objects.

```
function UNIQUE-PERMUTATIONS($A$)
    initialize result to $\emptyset$
    generate U a set of all unique elements of $A$ in $O(|A|)$ time
    for i $\in U$:
        set $B$ to $A$ with the first instance of i removed
        for P $\in$ UNIQUE-PERMUTATIONS($B$):
            add [i] + P to result
    return result
```

This is guaranteed not to have any duplicates due to non-distinct values. Its runtime is determined from the recurrence $T(n) = nT(n-1) + \mathcal{O}(n)$.

□

Problem 3

You are given a sorted array consisting of numbers, each of which appears twice in the array, except for one number which appears only once. Describe an $\mathcal{O}(\log n)$ algorithm to find the value of the number which appears only once.

Solution. We utilize a modified form of binary search. We notice that if there is only one number that appears once, that the sorted array must have an odd number of elements: if there are m numbers that appear twice, then the array has $n = 2m + 1$ total.

We divide the array in three: a left subarray, the middle element by itself (define i to be the index of the middle element), and the right subarray.

We compare $A[i-1], A[i]$, and $A[i+1]$. If $A[i-1], A[i]$ and $A[i+1]$ all differ, then $A[i]$ must be the element that appears only once, and we return it. Otherwise, two of these are equal and the third differs.

In the remaining cases, we must consider that if the input is split into two parts, one even and one odd, such that the two border elements differ, the element that occurs only once must occur in the side with an odd number of elements, as the even element side cannot have an element that appears only once and all others appear twice.

We now consider whether $n \equiv 1 \mod 4$ or $n \equiv 3 \mod 4$:

In the case of $n \equiv 1 \mod 4$, the left and right subarrays have an even number of elements, as if the array has $4m + 1$ elements, the left and right subarrays each have $2m$ elements. That means that if $A[i-1] = A[i]$, that the left+middle subarray must contain the value that appears only once, and if $A[i] = A[i+1]$, that the middle+right subarray must contain the value that appears only once, as these subarrays have odd size and fit the description above. We can then recursively search the appropriate subarray to find the element.

In the case of $n \equiv 3 \mod 4$ it is the opposite: the left and right subarrays contain odd numbers of elements, and if $A[i-1] = A[i]$ the right subarray must have the value that only appears once, and if $A[i] = A[i+1]$ that the left subarray contains the value that appears only once. We recursively search the appropriate subarray.

In all cases, the runtime is governed by $T(n) = T(n/2) + \mathcal{O}(1)$, and this recurrence yields a $\mathcal{O}(\log n)$ runtime.

Example: $A = [1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 6]$. $n = 11$ so we are in the $3 \mod 4$ case.

We divide the input into subarrays $[1, 1, 2, 3, 3], [4], [4, 5, 5, 6, 6]$. We have that $A[i] = A[i+1]$, therefore by our algorithm the left subarray $[1, 1, 2, 3, 3]$ is where we recurse.

Recursing, we divide the input into $[1, 1], [2], [3, 3]$. We locate 2 as the element as it differs from its neighbors, and return it. $\square$

Problem 4

Let's play a game. You have a $n \times n$ square grid. Initially you must place a token on a square in the first row. In each turn, you move your token to one square to the right, or one square down. The game ends when you move your token out of the board. Each square on the grid has a numerical value, which could be positive, zero, or negative. You start with 0 score. Whenever the token lands on a square, you add its value to your score. You try to score as many points as possible.

For example, given the grid below, and placing initial token on 1st row 2nd column, and moving down, down, right, down, down, the game ends and you have score -9 (which is not the maximum possible):

−1	7	-8	10	−5
−4	−9	8	−6	0
5	−2	−6	−6	7
−7	4	7	−3	−3
7	1	−6	4	−9

Describe an algorithm to compute the maximum score, given a board A[1 . . .][1 . . . n], and analyze its time complexity.

This problem can be solved using backtracking or dynamic programming. For additional challenge, try both!

Solution. For backtracking, a simple solution is to notice that the best score possible is the highest value path from any of $A[1][j], 1 \leq j \leq n$ to any of $A[n][j], 1 \leq j \leq n$, or $A[i][n], 1 \leq i \leq n$ moving only to the right or down.

Thus, one can pick at turns each of moving downward or to the right, starting from any of those points, recursively find the best score for each path starting at the square downward or to the right, and choose the highest-score one (the maximum). The paths end when they fall off the board. Pseudocode for the backtracking algorithm is provided in the function $SOLVE$ which uses a backtracking search function $SCORE$.

This algorithm will typically call itself twice every time it is called, and thus it has an exponential runtime. The longest path possible is $\mathcal{O}(n)$, therefore this runtime is $\mathcal{O}(2^n)$.

```
function SCORE($A, i, j, s$)
    if i $> n$ or j $> n$:
        return $s$
    else:
        $s \leftarrow s + A[i][j]$
        return max(SCORE($A, i+1, j, s$), SCORE($A, i, j+1, s$))
function SOLVE($A$)
    return $\max_{1 \leq j \leq n}$(SCORE($A, 1, j, 0$))
```

The first step for a dynamic programming solution is to determine a set of recursive subproblems. Given that the token moves only down or to the right, the state of the game when the token is at $A[i][j]$ in general is determined entirely by the numerical value at $A[i][j]$, and its state at either $A[i-1][j]$ or $A[i][j-1]$. This gives us a clue for how we might structure subproblems that are recursively solvable.

We define a set of subproblems $SCORE(i, j)$ as describing the maximum score of any game where the token is located at $A[i][j]$. The game can end only when the token is in the last row or last column (it has the option to fall off the board), therefore the best possible game is given by the maximum of $SCORE(n, j)$ or $SCORE(i, n)$ over all i, j. By the earlier observation, since the state of the game when at $A[i][j]$ is determined by the state at either $A[i-1][j]$ or $A[i][j-1]$ in general (if $j = 1$ or $i = 1$ one of these might not exist), $SCORE(i, j)$ must be determined by $SCORE(i-1, j)$ and $SCORE(i, j-1)$. The best score available at that point is the best score at its preceding state, plus the numerical value at the token.

This leads to a recursive formulation:

$$SCORE(i, j) = \max(SCORE(i-1, j), SCORE(i, j-1)) + A[i][j]$$

which holds if $i, j \neq 1$. Now to find the base cases.

When $i = 1, j = 1$, we must have started at this point, as there is no point to the left or above it which we could have reached it from. Therefore, $SCORE(1, 1) = A[1][1]$. For $i = 1$ otherwise, it is possible to either start there, or reach there from the left. Therefore, $SCORE(1, j) = \max(SCORE(1, j-1) + A[1][j], A[1][j])$.

When $j = 1, i \neq 1$, we must have that $SCORE(i, 1) = SCORE(i-1, 1) + A[i][1]$ as it is impossible to get there from anywhere but $A[i-1][1]$.

The final recurrence is thus:

$$SCORE(i, j) = \begin{cases} A[1][1] & i = 1, j = 1 \\ \max(SCORE(1, j-1) + A[1][j], A[1][j]) & i = 1, j \neq 1 \\ SCORE(i-1, 1) + A[i][1] & i \neq 1, j = 1 \\ \max(SCORE(i-1, j), SCORE(i, j-1)) + A[i][j] & i \neq 1, j \neq 1 \end{cases}$$

And the solution to the problem overall is

$$\max(\max_{1 \leq j \leq n}(SCORE(n, j)), \max_{1 \leq i \leq n}(SCORE(i, n)))$$

There are n^2 different subproblems and they each require a constant number of comparisons. This implies that solving all subproblems given the correct recursion order (so that we never need to calculate any subproblem more than once) is $\mathcal{O}(n^2)$. The final step takes the maximum of $\mathcal{O}(n)$ different values on the bottom and right edges. Overall, the runtime remains $\mathcal{O}(n^2)$.

□