CPSC 320 2023S2: Assignment 3 Solutions

3 Exhibition guarding

In the Exhibition Guarding Problem (described also in Tutorial 6), we are given a line L that represents a long hallway in a gallery; assume that the line has a left and right end. We are also given an unordered array X of real numbers that represent the positions of precious objects in this hallway, where a position marks the distance from the left end of the line. A guard at a position p on the line can protect objects within distance d of p, that is, objects in the range $[p-d, p+d]$.

In this problem you'll analyze the correctness of this algorithm.

1. (3 marks) Explain why the algorithm provided in the solution to Tutorial 6, and also included below, produces a *valid* solution, i.e., a set of guard positions that ensures that all objects are guarded. Here, $X[1..n]$ describes the positions of n objects along a line, and a guard at a position p on the line can protect objects in the range $[p-d, p+d]$. (Do not concern yourself yet with whether the set of positions is minimized.)

```
procedure PLACE-GUARDS(($X[1..n], d$))
    ▷ $X$ contains positions of $n \geq 0$ objects on a line
    ▷ $d$ is the distance (both to the left and right) that a guard can protect
    Sort $X$ in increasing order
    return Place-Guards-Helper($X[1..n], d$)

procedure PLACE-GUARDS-HELPER(($X[1..n], d$))
    if $n == 0$ then
        return the empty set ▷ There are no objects, so no guard is needed
    else
        $p \leftarrow X[1] + d$ ▷ Place a guard as far right as possible while protecting $X[1]$
        ▷ skip the other objects protected by this guard:
        $i \leftarrow 2$
        while ($i \leq n$) and ($X[i] \leq p + d$) do
            $i \leftarrow i + 1$
        return $\{p\}$ +PLACE-GUARDS-HELPER($X[i..n], d$)
```

SOLUTION: By induction on n. Consider instance $(X[1..n], d)$. When $n = 0$, the PLACE-GUARDS-HELPER procedure returns no guards, which is a valid solution. When $n > 0$, the value of i at the end of the **while** loop is such that all objects at positions $X[1..i-1]$ are in the range $[p-d, p+d]$, where p is the guard position chosen by the algorithm. Also, by induction, the remaining objects are protected by the solution to the recursive call. So the overall solution $\{p\}$ + PLACE-GUARDS-HELPER$(X[i..n], d)$ is valid.

2. (4 marks) Show also that the algorithm produces an optimal solution, that minimizes the number of guards.

SOLUTION: We'll use an exchange argument (described in Chapter 4 of K&T) for this purpose. Let p_1^* and p_1 be the positions of the first (leftmost) guard in G^* and G, respectively.

We first show that $G^* - \{p_1^*\} + \{p_1\}$ is an optimal solution for instance $(X[1..n], d)$. Clearly $p_1^* \leq p_1$, since otherwise G^* would not protect the leftmost object at position $X[1]$. Consider an arbitrary object O protected by the guard placed at p_1^*. O's position must be in the range $[X[1], \ldots, p_1^* + d]$, which is a subset of the range $[X[1], \ldots, p_1 + d]$. Hence object O is also protected by the guard placed at p_1. Therefore every object protected by a guard at position p_1^* is also protected by a guard at p_1. This means that the solution $G^* - \{p_1^*\} + \{p_1\}$ protects all objects and has the same number of guards as G^*, and so is an optimal solution.

Next we prove correctness of the algorithm by induction on n, the number of objects. Consider any instance $(X[1..n], d)$. The base case is when $n = 0$, in which case the algorithm returns no guards, which is optimal. Let $n \geq 1$ and suppose that the algorihm is correct on instances with $n - 1$ objects. We know from the previous paragraph that there is an optimal solution G^* containing the first guard position p_1 chosen by our algorithm on instance $(X[1..n], d)$. The remaining guards at positions other than those in $G^* - \{p_1\}$ must protect all objects not protected by the guard at p_1. We know by induction that our algorithm finds the minimum number of guards to protect these remaining objects. Therefore overall, our algorithm finds a solution G with a number of guards that is no greater than $|G^*|$, and so G is optimal.

4 Pell numbers

The infinite sequence of rational approximations to the square root of 2 starts with the numbers 1/1, 3/2, 7/5, 17/12, and 41/29. The infinite sequence of denominators of this sequence is called the Pell sequence, so the Pell number sequence begins with 1, 2, 5, 12, and 29. The Pell numbers are also defined by the following recurrence relation:

$$P_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ 2P_{n-1} + P_{n-2}, & \text{otherwise.} \end{cases}$$

1. (3 marks) Use induction to show that

$$P_n = \frac{(1+\sqrt{2})^n - (1-\sqrt{2})^n}{2\sqrt{2}}.$$

SOLUTION: For our base cases when $n = 0$ and $n = 1$ we have that

$$P_0 = 0 = \frac{1-1}{2\sqrt{2}} = \frac{(1+\sqrt{2})^0 - (1-\sqrt{2})^0}{2\sqrt{2}},$$

and

$$P_1 = 1 = \frac{2\sqrt{2}}{2\sqrt{2}} = \frac{(1+\sqrt{2})^1 - (1-\sqrt{2})^1}{2\sqrt{2}}.$$

For the inductive step, let $n > 1$ and assume that

$$P_{n-1} = \frac{(1+\sqrt{2})^{n-1} - (1-\sqrt{2})^{n-1}}{2\sqrt{2}}$$

and

$$P_{n-2} = \frac{(1+\sqrt{2})^{n-2} - (1-\sqrt{2})^{n-2}}{2\sqrt{2}}.$$

Then

$$\begin{aligned}
P_n &= 2P_{n-1} + P_{n-2} \\
&= 2\frac{(1+\sqrt{2})^{n-1} - (1-\sqrt{2})^{n-1}}{2\sqrt{2}} + \frac{(1+\sqrt{2})^{n-2} - (1-\sqrt{2})^{n-2}}{2\sqrt{2}} \\
&= \frac{(1+\sqrt{2})^{n-2}(2+2\sqrt{2}+1) - (1-\sqrt{2})^{n-2}(2-2\sqrt{2}+1)}{2\sqrt{2}} \\
&= \frac{(1+\sqrt{2})^{n-2}(1+\sqrt{2})^2 - (1-\sqrt{2})^{n-2}(1-\sqrt{2})^2}{2\sqrt{2}} \\
&= \frac{(1+\sqrt{2})^{n} - (1-\sqrt{2})^{n}}{2\sqrt{2}},
\end{aligned}$$

completing the proof.

2. (3 marks) Write a recursive algorithm Pell(n) that returns the n th Pell number.

   SOLUTION:

```
Algorithm Pell($n$)
// Returns the $n$th Pell number
    If $n = 0$ then
        Return 0
    ElseIf $n = 1$ then
        Return 1
    Else
        Return 2 Pell($n-1$) + Pell($n-2$)
```

3. (3 marks) Write a recurrence relation that describes the running time of your algorithm as a function of n.

   SOLUTION: Let $T(n)$ denote the running time of the algorithm. Then $T(0)$ and $T(1)$ equal some positive constant c, and for $n > 1$,

$$T(n) = T(n-1) + T(n-2) + c.$$

   This is because there are two recursive calls with parameters $n-1$ and $n-2$, and the remaining code takes constant time to execute.

4. (3 marks) Describe the solution to the recurrence from part (3) in terms of another sequence of numbers you are almost certainly already familiar with (a closed form formula for the n^{th} term of that sequence is easily found online).

   SOLUTION: The recurrence for $T(n)$ is similar to that for the Fibonacci numbers, which are defined by the recurrence: $F(0) = 0$, $F(1) = 1$, and for $n > 1$

$$F(n) = F(n-1) + F(n-2).$$

   The closed form solution for $F(n)$ (easily found using a Google search) is

$$F(n+1) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}.$$

If we assume that the constant c is at least 1, then $T(n) \geq F(n)$ for all n.

Although beyond the scope of what is expected in your solutions to this problem, we can also show using strong induction that $T(n) = O(F(n)$; specifically that $T(n) \leq 4cF(n) - c$ for all $n \geq 1$.

Our base cases will be $n = 1$ and $n = 2$. When $n = 1$,

$$T(1) = c \leq 3c = 4cF(1) - c.$$

When $n = 2$,

$$T(2) = T(1) + T(0) + c = 3c = 4cF(2) - c.$$

Let $n > 2$. The inductive hypothesis is that for any $i, 1 \leq i \leq n-1$, $T(i) \leq 4cF(i) - c$. Then

$$T(n) = T(n-1) + T(n-2) + c \leq (4cF(n-1) - c) + (4cF(n-1) - c) + c = 4cF(n) - c,$$

completing the inductive step.

5 Bumps in a Log

Imagine you have n numbers stored in an array $A[1], \ldots, A[n]$. Furthermore, assume that no two adjacent numbers are equal. We'll define a *bump* to be a location i that is higher than its neighbours on either side, i.e., $A[i-1] < A[i] > A[i+1]$. (When $i = 1$ or $i = n$, it is a bump if it's higher than its only neighbour.)

1. (3 marks) The obvious greedy algorithm is to start at any point in the array and then as long as a neighbour has a larger number, move to that neighbour. (This is described as "hill climbing".) Describe a class of problem instances where hill climbing will take $\Omega(n)$ time.

   SOLUTION: If the numbers are in increasing order, and you start at $A[1]$, then the hill climbing algorithm will traverse the entire array. Similarly, if the numbers were in decreasing order.

2. (3 marks) Give an algorithm that finds a bump in time $O(\log n)$. This is somewhat surprising, as we can't even look at the entire array in that time bound, and we know nothing about the order of the items in the array. You should justify why your algorithm works, but you do not need to supply a fully formal proof.

   SOLUTION: From the $\log n$ time bound, you can suspect that you'll be doing a divide-and-conquer solution. The key idea is that you can do an algorithm much like binary search. You pick the midpoint of your search interval and check its two neighbours. If both are smaller, then you've found a bump and can terminate. Else, if you go in the direction toward a bigger neighbour (if both are larger, pick either one), there's guaranteed to be a bump in that half of the search space. You can see this because if you were hill-climbing in that half, you couldn't cross the midpoint at the end of that half, because it's downhill to that point. Since you keep dividing the search space in half, the runtime is $O(\log n)$.

   Here's some pseudocode. For convenience, we can imagine that $A[0] = A[n+1] = -\infty$.

```
1: procedure BumpInLog(array $A$ of size $n$)
2:     $l = 1$                                        ▷ Lower bound of search range.
3:     $h = n$                                        ▷ Upper bound of search range.
4:     while $l \neq h$ do
   // Loop invariant is that there's a bump in $[l, h]$
   // and that $A[l-1] < A[l]$ and $A[h] > A[h+1]$.
5:         Let $m = \lfloor (l+h)/2 \rfloor$.
```

```
        **if** $A[m-1] > A[m]$ **then**
            Let $h = m - 1$.
        **else if** $A[m] < A[m+1]$ **then**
            Let $l = m + 1$.
        **else**
            **return** $m$                                        ▷ Bigger than both neighbours
    **return** $m$
```