Design and Analysis of Algorithms: Homework #1

Due in class on January 30, 2017

Professor Kasturi Varadarajan

Alic Szecsei

Problem 1

Prove that it is possible for the Boston Pool algorithm to execute $\Omega(n^2)$ rounds. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid proposals.)

Solution

Suppose we have four doctors, α through δ, and four hospitals, A-D, with their respective preferences given below.

Hospital	Pref 1	Pref 2	Pref 3	Pref 4
A	α	β	γ	δ
B	α	β	γ	δ
C	α	β	γ	δ
D	α	β	γ	δ
Doctor	**Pref 1**	**Pref 2**	**Pref 3**	**Pref 4**
α	D	C	B	A
β	C	B	A	D
γ	B	A	D	C
δ	A	D	C	B

Specifically, note that all hospitals have the same preferences for doctors, and each doctor's preferences are shifted over by one.

We allow hospitals to propose to doctors, and thus use the following proposals, starting with Hospital A:

1. $A \leftarrow \alpha$.

2. $B \leftarrow \alpha$, since α prefers B to A.

3. $A \leftarrow \beta$.

4. $C \leftarrow \alpha$, since α prefers C to B.

5. $B \leftarrow \beta$, since β prefers B to A.

6. $A \leftarrow \gamma$.

7. $D \leftarrow \alpha$, since α prefers D to C.

8. $C \leftarrow \beta$, since β prefers C to B.

9. $B \leftarrow \gamma$, since γ prefers B to A.

10. $A \leftarrow \delta$.

There are 4 *initial* proposals to doctors, and $1 + 2 + 3$ *subsequent* proposals, in which doctors received a better proposal and swapped hospitals. Were another hospital and doctor to be added, following the same pattern, there would be 5 initial proposals and $1 + 2 + 3 + 4$ subsequent proposals, and so on. In general, if there are n initial proposals, then there will be $\sum_{i=1}^{n-1} i$ subsequent proposals. This summation has a closed form of $n(n-1)/2$ subsequent proposals.

Since $n(n-1)/2 + n$ is $\Theta(n^2)$, it's also $\Omega(n^2)$.

Problem 2

Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

- A hospital prefers an unmatched doctor to its assigned match.

- A doctor prefers an unmatched hospital to her assigned match.

- An unmatched doctor and an unmatched hospital appear in each other's preference lists.

Describe and analyze an efficient algorithm that computes a stable matching in this setting.

Note that a stable matching may leave some doctors and hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Dr. House as their only acceptable intern, then only House and Harvard will be matched.

Solution

Algorithm 1 General stable matching algorithm

1: **function** GENERALSTABLEMATCHING($H[1..m], D[1..n]$)
2:     $U \leftarrow$ a list of all hospitals
3:     **while** $\exists$ a hospital in U **do**
4:         pick a hospital H in U
5:         **if** H has proposed to all doctors on its preference list **then**
6:             remove H from U
7:             **continue**
8:         **end if**
9:         H proposes to the "next" doctor on its list (in order of preference), δ
10:         **if** δ is not matched **then**
11:             δ is matched with H
12:             remove H from U
13:         **else**
14:             **if** δ prefers its current match **then**
15:                 δ rejects H and remains matched with its current match
16:             **else**
17:                 δ abandons its current match and matches with H
18:                 add δ's abandoned match to U
19:             **end if**
20:         **end if**
21:     **end while**
22: **end function**

Correctness

Let us assume that GENERALSTABLEMATCHING runs on a set of doctors and hospitals. For each unstable situation, we can show that it cannot be a result of our algorithm.

A hospital prefers an unmatched doctor to its assigned match.

If the algorithm concludes, all hospitals have been removed from the set of unmatched hospitals U. In order to remove a hospital from U, it must have proposed to, and been rejected by, all doctors; or, it must be matched to a doctor. Since hospitals choose doctors in descending order of preference, any doctor that H prefers to its current match must have rejected it in favor of another hospital; thus, no hospital can prefer an unmatched doctor to its assigned match.

A doctor prefers an unmatched hospital to her assigned match.

When the algorithm concludes, all hospitals are matched *unless* they have been rejected by all doctors on their list of preferences. Thus, if a hospital is unmatched, all its doctors have rejected it for another hospital; therefore no doctor can prefer an unmatched hospital to her assigned match.

An unmatched doctor and an unmatched hospital appear in each other's preference lists.

Similar to the previous situation, all hospitals are matched *unless* they have been rejected by all doctors on their list of preferences. If a doctor is unmatched, then she has not been proposed to by a hospital, thus cannot be on a hospital's preference list. Therefore, no combination of unmatched hospital and doctor can appear in each other's preference lists.

Runtime

As a worst case, we can assume that each hospital is iterated over and must propose to each doctor on its list of preferences. Thus, the worst-case runtime for this algorithm is $O(m \times n)$, where m is the number of hospitals and n is the number of doctors.

Problem 3

Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed.

1. Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces, each composed of 3 squares.

2. Describe and analyze an algorithm to compute such a tiling, given the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time.

Solution
Part 1

Base case $n = 1$: If $n = 1$ there is a single "hole" and an L-shaped piece can be trivially rotated around the hole.
So, the theorem holds when $n = 1$.
Inductive hypothesis: Suppose the theorem holds for all values of n up to some k, $k \geq 1$.
Inductive step: Let $n = k + 1$.
Then we can split our $2^n \times 2^n$ chessboard into 4 quadrants. Our missing piece must be in one of these 4 quadrants. By the inductive hypothesis, this quadrant can be filled, since it is a $2^{n-1} \times 2^{n-1}$ chessboard with a missing piece. We can then place missing pieces for the other three quadrants such that the missing piece is one of the central 4 squares. By the inductive hypothesis, these quadrants can be filled, since they are $2^{n-1} \times 2^{n-1}$ chessboard with one missing piece each.
These 3 newly-created missing pieces can then be tiled trivially by a single L-shaped piece.

Part 2

Runtime

Each call to Tile Chessboard with $n > 1$ creates 4 recursive calls and some number, δ, of constant calls. Each of these recursive calls have size $2^{n-1} \times 2^{n-1}$, and therefore, $T(n) = 4T(n-1) + \delta$. The annihilator for this is $(\mathbf{E} - 4)(\mathbf{E} - 1)$, so our closed-form solution is $T(n) = \alpha 4^n + \beta$, which is $O(4^n)$.

Algorithm 2 Missing-Square Chessboard Tiling

1: **function** TILECHESSBOARD($n, row, column$)
2:     $l \leftarrow$ new $List$
3:     **if** $n = 1$ **then**   ▷ base case
4:         create new shape s
5:         append s to l
6:         **return** l
7:     **else**   ▷ recursion
8:         **if** missing piece is in top-left quadrant **then**
9:             $tl \leftarrow$ TILECHESSBOARD($n/2, row, column$)
10:            $tr \leftarrow$ TILECHESSBOARD($n/2, n/2 - 1, n/2$)
11:            $bl \leftarrow$ TILECHESSBOARD($n/2, n/2, n/2 - 1$)
12:            $br \leftarrow$ TILECHESSBOARD($n/2, n/2, n/2$)
13:        **else if** missing piece is in top-right quadrant **then**
14:            ...
15:        **else if** missing piece is in bottom-left quadrant **then**
16:            ...
17:        **else if** missing piece is in bottom-right quadrant **then**
18:            $tl \leftarrow$ TILECHESSBOARD($n/2, n/2 - 1, n/2 - 1$)
19:            $tr \leftarrow$ TILECHESSBOARD($n/2, n/2 - 1, n/2$)
20:            $bl \leftarrow$ TILECHESSBOARD($n/2, n/2, n/2 - 1$)
21:            $br \leftarrow$ TILECHESSBOARD($n/2, row - n/2, column - n/2$)
22:        **end if**
23:        append tl, tr, bl, br to l
24:        create new shape s in center, so missing piece is in the same quadrant as $row, column$
25:        append s to l
26:        **return** l
27:    **end if**
28: **end function**

Problem 4

Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the pegs are numbered 0, 1, and 2, as in problem ??, and your task is to move a stack of n disks from peg 1 to peg 2.

1. Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

2. Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make?

Solution
Part 1

Algorithm 3 Tower of Hanoi, Variant 1

1: **function** HANOIVARIANT1(n, src, dst, tmp)
2:     **if** $n > 0$ **then**
3:         HANOIVARIANT1($n - 1, src, dst, tmp$)
4:         move disk n from src to tmp
5:         HANOIVARIANT1($n - 1, dst, src, tmp$)
6:         move disk n from tmp to dst
7:         HANOIVARIANT1($n - 1, src, dst, tmp$)
8:     **end if**
9: **end function**

This algorithm makes 3 recursive calls, and two moves of the nth disk; therefore $T(n) = 3T(n - 1) + 2$, where $T(0) = 0$.

$$T(n) = 3T(n - 1) + 2 \Rightarrow T(n + 1) - 3T(n) = 2 \Rightarrow (\mathbf{E} - 3)T(n) = 2$$

Since $(\mathbf{E} - 1)$ annihilates any constant α, we can annihilate the residue by applying the operator $(\mathbf{E} - 1)$. Thus, our annihilator is $(\mathbf{E} - 1)(\mathbf{E} - 3)$. This annihilator gives us the generic solution $T(n) = \alpha 3^n + \beta$ for some unknown constants α and β. The base cases give us $T(0) = 0 = \alpha 3^0 + \beta$ and $T(1) = 2 = \alpha 3^1 + \beta$. When we solve this, we get $\alpha = 1$ and $\beta = -1$.

Therefore, our algorithm takes $T(n) = 3^n - 1$ moves.

As a sanity check, we can prove this closed-form inductively:

Base case $n = 0$: If $n = 0$, $T(0) = 3^0 - 1 = 0$ moves. Our base case was that $T(0) = 0$, so this checks out.
Inductive hypothesis: Suppose the theorem holds for all values of n up to some k, $k \geq 0$.
Inductive step: Let $n = k + 1$.
Then $T(n) = 3T(n-1) + 2 = 3T(k) + 2$. By the inductive step, this is $T(n) = 3 \times 3^k - 1 + 2 = 3^{k+1} - 1 = 3^n - 1$, which matches our closed form.
Part 2

7

Algorithm 4 Tower of Hanoi, Variant 2

1: **function** HANOIVARIANT2(n, src, dst, tmp)
2:     **if** $n > 0$ **then**
3:        HANOIVARIANT2($n - 1, src, dst, tmp$)   ▷ Move $n - 1$ disks from src to dst
4:        move disk n from src to tmp ▷ Move disk n from src to tmp
5:        HANOIVARIANT2($n - 1, dst, tmp, src$)   ▷ Move $n - 1$ disks from dst to tmp
6:        HANOIVARIANT2($n - 1, tmp, src, dst$)   ▷ Move $n - 1$ disks from tmp to src
7:        move disk n from tmp to dst ▷ Move disk n from tmp to dst
8:        HANOIVARIANT2($n - 1, src, dst, tmp$)   ▷ Move $n - 1$ disks from src to dst
9:     **end if**
10: **end function**

This algorithm makes 4 recursive calls, and has 2 moves of the nth disk; therefore $T(n) = 4T(n - 1) + 2$, where $T(0) = 0$.

$$T(n) = 4T(n - 1) + 2 \Rightarrow T(n + 1) - 4T(n) = 2 \Rightarrow (\mathbf{E} - 4)T(n) = 2$$

Since $(\mathbf{E} - 1)$ annihilates any constant α, we can annihilate the residue by applying the operator $(\mathbf{E} - 1)$. Thus, our annihilator is $(\mathbf{E} - 1)(\mathbf{E} - 4)$. This annihilator gives us the generic solution $T(n) = \alpha 4^n + \beta$ for some unknown constants α and β. The base cases give us $T(0) = 0 = \alpha 4^0 + \beta$ and $T(1) = 2 = \alpha 4^1 + \beta$. When we solve this, we get $\alpha = \frac{2}{3}$ and $\beta = -\frac{2}{3}$.

Therefore, our algorithm takes $T(n) = \frac{2}{3}4^n - \frac{2}{3}$ moves.

As a sanity check, we can prove this closed-form inductively:

Base case $n = 0$: If $n = 0$, $T(0) = \frac{2}{3}4^0 - \frac{2}{3} = 0$ moves. Our base case was that $T(0) = 0$, so this checks out.
Inductive hypothesis: Suppose the theorem holds for all values of n up to some k, $k \geq 0$.
Inductive step: Let $n = k + 1$.
Then $T(n) = 4T(n - 1) + 2 = 4T(k) + 2$. By the inductive step, this is $T(n) = 4 \times \frac{2}{3}4^k - 4 \times \frac{2}{3} + 2 = \frac{2}{3}4^{k+1} - \frac{2}{3} = \frac{2}{3}4^n - \frac{2}{3}$, which matches our closed form.

Problem 5

Consider this cruel and unusual sorting algorithm.

Algorithm 5 Cruel and Unusual

1: **function** CRUEL($A[1..n]$)
2:      **if** $n > 1$ **then**
3:          CRUEL($A[1..n/2]$)
4:          CRUEL($A[n/2 + 1..n]$)
5:          UNUSUAL($A[1..n]$)
6:      **end if**
7: **end function**
8: **function** UNUSUAL($A[1..n]$)
9:      **if** $n = 2$ **then**
10:         **if** $A[1] > A[2]$ **then**   ▷ the only comparison!
11:             swap $A[1] \longleftrightarrow A[2]$
12:         **else**
13:             **for** $i \leftarrow 1$ to $n/4$ **do**   ▷ swap 2nd and 3rd quarters
14:                 swap $A[i + n/4] \longleftrightarrow A[i + n/2]$
15:             **end for**
16:             UNUSUAL($A[1..n/2]$)   ▷ recurse on left half
17:             UNUSUAL($A[n/2 + 1..n]$)   ▷ recurse on right half
18:             UNUSUAL($A[n/4 + 1..3n/4]$)   ▷ recurse on middle half
19:         **end if**
20:     **end if**
21: **end function**

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called *oblivious*. Assume for this problem that the input size n is always a power of 2.

1. Prove by induction that CRUEL correctly sorts any input array. [*Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?*]

2. Prove that CRUEL would not correctly sort if we removed the for-loop from UNUSUAL.

3. Prove that CRUEL would not correctly sort if we swapped the last two lines of UNUSUAL.

4. What is the running time of UNUSUAL? Justify your answer.

5. What is the running time of CRUEL? Justify your answer.

Solution

Part 1