Virpobe Paireepinart

Algorithms Fall 2009 Assignment 1

1. My algorithm:

suppose all disks are on peg1.

If you have an odd number of disks, move the top disk to the destination peg. if you have an even number of disks move the top disk to the swap peg.

After this, move disk 2 to the peg you did not move disk 1 to. Then place disk 1 on to disk 2. Then move disk 3 to the place you moved disk 1 to originally. Then move disk 1 back to the source, move disk 2 onto disk 3, and then place disk 1 back onto it. after this, you just continuously move new disks into the empty space, and shuffle disks back and forth to stack onto this new disk.

Suppose you are presented with 6 disks.

The disks must then be colored alternately,

such as Red-Blue-Red-Blue-Red-Blue.

If we think of each of these colors as being the state of whether a number is even or odd. Thus the bottom ring, ring 6, is red because it's an even number, and ring 5 is blue because it's an odd number, and so on.

In order to move these items to post 3 from post 1, we merely say to move all items except the bottom disk from post 1 to post 2, then move the final disk to post 3, and then move all of the items from post 2 onto post 3. And in order to move the 5 disks, we just say to move the 4 disks and then move the fifth one and move the 4 disks back on top of it.

We already know this algorithm works because it is outlined in class. We must also prove that, at no point in the algorithm, disks of the same color are on top of each other.

Take the case where n = 1. Moving 1 disk from peg 1 to peg 3 is trivial. you just move it, so our algorithm holds.

if n = 2, peg 1 is 21, peg 2 is [] and peg3 is []. move the 1 to peg 2, the 2 to peg 3, and then the 1 to peg 3. This satisfies the requirement.

if n = 3, peg1 is 321, peg2 is [] and peg3 is []. the sequence of moves will be

32 [] 1

3 2 1

3 21 []

[] 21 3

And from here it's the same as moving 2 items from peg 2 to peg 3 with peg 1 as the swap space.

suppose we have n = 4. the sequence will be

4321 [] []

432 1 []

43 1 2

43 [] 21

4 3 21

41 3 2

41 32 []

4 321 []

[] 321 4

and then from there it's the same as n=3, moving 3 elements from peg 2 to peg 3 by peg 1.

Suppose n = 5. the sequence will be

54321 [] []

5432 [] 1

543 2 1

543 21 []

54 21 3

541 2 3

541 [] 32

54 [] 321

5 4 321

5 41 32

52 41 3

521 4 3

521 43 []

52 43 1

5 432 1

5 4321 []

[] 4321 5

and from here it's the same as moving 4 rings from peg 2 to peg 3 with peg1 as swap space.

If you will note above, for any number of rings n, the pattern of swaps that takes place will never place the same color disks on each other. To move n rings, it will move n-1 rings, and thus n-2 rings, etc… until it has only to move 1 ring. Suppose the case n = 5. In order to move the 5 rings to peg 3, it must move 4 rings first to peg 2. In order to do this operation, it will have to use peg 2, 3, and 1 as swap points, but it must end up with peg 3 as being empty. Thus the ring "1" must be either on peg 1 or peg 3, so that at the end of the sequence it may swap to peg "2". At this point, peg 2 will contain 432 and peg 1 will contain 5. Therefore we know that 1 must be on peg "3" because if it were on peg 1 it would break our condition. Because there are an odd number of rings being moved, whichever peg we put disk 1 on, it will end up there on the second-to-last step. So our algorithm must place disk 1 onto peg 1 as the first step. However, if there are an even number of rings to be moved, our algorithm must place disk 1 onto peg 2.

Suppose we are halfway through our algorithm (and our n is odd.). There are n-4 disks on peg 1, and there are 4 disks on peg 2 (4 3 2 1), and peg 3 is empty. We know that peg 1 will contain n …. 5, and peg 3 will be empty. Then disk 5 will be moved to peg 3, and we will need to move 4 disks from peg 2 to peg 3. as stated in the algorithm, since we're moving an even # of disks (4) we will place the top disk onto the swap peg. In this case, peg 3 is our destination peg and peg1 is our swap peg, so we will move 1 onto the swap peg. So peg 1 will contain n …. 6 1 which does not violate our condition. then peg 3 will get the value 2 and will then contain 5 2, which also does not violate the condition. Then 1 will be moved onto peg 3 which will then contain 5 2 1. As you can see, the swap peg will always contain an

opposite polarity value to the one we are placing on it, and the same for the destination. So we will never place a disk onto a peg that has the same value.

2) Binary 1 performs better for a successful search, on average. Binary 2 is the same as binary1 except it will perform an additional check to see if the middle value is our target.

Suppose we have n elements, and assume that the element we're searching for, "k", is contained in the list. The likelihood of the middle value being our target is 1. The likelihood of one of the other elements being "k" are n-1. Just intuitively we can see that this is not worth considering, because adding an extra comparison will double our runtime in every case except where the middle is our searched element. The case of the middle element being the element we are looking for is exceptionally rare, especially as n approaches infinity (as it should, for O analysis).

Therefore, binary 1 will perform better on average. Binary2 will run almost twice as slowly, unless the middle value is the target, and as we've said, that is very unlikely (less and less likely as n -> inf.).

Therefore, for the average case, binary2 will run 2lgn - c, where c is a very small constant.

The average case of binary1 will be lgn, which is just the regular runtime efficiency of a normal binary search.

3. under O consideration we don't need to count constants, or lesser degree terms, so we get

A = log(log(n)) + log(log(log(n^2))) =O(log^2 (n))

B = 2^log(n) – 3n^(1/2) + 10 = n – 3(sqrt(n)) = O(n)

C = 2^(1/2 * log(n)) + n^(1/2) = 2^1/2 + n + n^(1/2) = n + n^(1/2) = O(n)

D = 3 * 2^2^n + log(2log(n)) = O(2^2^n)

E = 2^n + 5n + 500 = O(2^n)

F = 10 + n/2 * n^(1/2) + 432 = O(n/2 * n^(1/2)) = O(n^1.5)

A < B <= C <= F < E < D

4.

Base case. For F(1), the runtime is 1. For F(2), F(1) needs to be calculated, so the runtime is 2.

Induction: For any n > 2, you will need to calculate F(n-1) and F(n-2) and one more calculation to find F(n). The time to perform F(n-1) will be bounded below by (2^(n-1)). For F(n-2) it will be (2^(n-2)) Therefore to calculate F(n) you must spend at least 2 * 2^(n-2) + 1 + c operations. (3/2)^n is the highest value less than this, so the sequence is bounded below by (3/2)^n.

5. In order to test the wine without killing a wine tester per bottle, the king must mix bottles. If half of the bottles are mixed, and the other half are mixed, one taster can taste both. Then a third of the bottles are mixed, another third, and another third. one taster tastes each. then a fifth of the bottles are mixed for each of 5 tasters. Then 7, then 11, etc. It follows the pattern of prime numbers until it gets to n/2. So suppose there are 10 bottles. You would divide into halves, then thirds, then fifths. This would require 11 tasters total, which is not log(n), but after a certain point the number of tasters decreases to log(n). With this pattern of tasters, any single bottle can be identified as the source of the poison by determining every person who died, and figuring out which one bottle they all tasted.