Homework 1 - Robert Duque

Thursday, September 8, 2022 9:58 AM

PSID: 1582182

P0 (20pt): Solve the following recurrences. Give the answer in terms of Big-Theta notation. Assume base case has cost $T(1) = T(0) = 1$. Ignore ceilings and floors.

1. $T(n) = 3T(n/2) + n$
2. $T(n) = T(n/2) + T(n/3) + n$
3. $T(n) = 2T(n-1) + 2$
4. $T(n) = 3T(n/3) + n^2$
5. $T(n) = T(3n/4) + T(n/3) + n$

P0.1) I will use Master Theorem to solve this recurrence, the following equation has a = 3 and b = 2. With the value of $\log_2 3 > 1$, we take the form of $\theta(n^{\log_b a})$ which translates to

T(n) = $\boldsymbol{\theta(n^{\log_2 3})}$

P0.2) Using a recurrence tree,

```
              N                      n
          /        \
      n/2           n/3             5n/6
    /   \          /     \
  n/4  n/6      n/6   n/9          25n/36
```

The height of the tree will be influenced by n/2 since it is the greater value of the two subproblems. Height of the tree will come out to be $\log_2 n$.
We will then form a series for the amount of work done at each level,

level 0: $\left(\frac{5}{6}\right)^0 n$

Level 1: $\left(\frac{5}{6}\right)^1 n$

Level 2: $\left(\frac{5}{6}\right)^2 n$

n stays constant, therefore we can count total number of levels with n

Count total number of levels:

Choose longest path (n/2)

$$\left(\frac{1}{2}\right)^0 n \to \left(\frac{1}{2}\right)^1 n \to \left(\frac{1}{2}\right)^2 n \to \left(\frac{1}{2}\right)^k n$$

At last level, size becomes 1 so, $\left(\frac{1}{2}\right)^k n = 1$

$$\left(\frac{1}{2}\right)^k = \frac{1}{n}$$

$\mathrm{K} = \log_2 n$

Count total number of nodes in the last level:

Nodes at last level: $2^{\log_2 n} = n^{\log_2 2}$

Sum up the cost of all levels:

$= n * \log_2 n + O(n^{\log_2 2})$

Therefore, T(n) = $\boldsymbol{\theta(n \log_2 n)}$

P0.3) I will use substitution to solve this recurrence problem, first we will take the first recursion and plug it into the original equation,

T(n-1) = 2T(n-2)+2 $\rightarrow$ T(n) = 2[2T(n-2)+2]+2 $\rightarrow$ 2^2T(n-2) + 4 + 2
T(n-2) = 2T(n-3)+2 $\rightarrow$ 2[2^2T(n-3) + 4 + 2]+2 $\rightarrow$ 2^3T(n-3) + 8 + 4 + 2

These series of equations will lead to 2^kT(n-k) + $2^k + 2^{k-1} \ldots 2^2 + 2^1$
Assume n-k = 0, n = k
T(n) = 2^nT(0) + 2 + 2^2 + . . . + 2^k
= 2^n+$2^n - 2$
= $2^{n+1} - 2$
Therefore, the runtime complexity is

T(n) = $\boldsymbol{\theta(2^n)}$

P0.4) Using the Master Theorem, the following equation has a = 3, b = 3, and c is going to be 2. We will check to see if $\log_b a$ is smaller, greater, or equal to c to find the correct case.

$\log_3 3 = 1$ < (c = 2)
Since $\log_b a$ is less than c, we will use case 3 which implies that T(n) = f(n) which translates to

T(n) = $\boldsymbol{\theta(n^2)}$

P0.5) Using a recurrence tree,

```
                         N                                          n
                 /                   \
              n/3                     3n/4                          13/12 n
            /   \                   /         \
        n/9  3n/12              3n/12       9n/16                   (13/12)^2 n
     /    \     /     \          /     \       /     \
  n/27  3n/36 3n/36 9n/48   3n/36 9n/48  9n/48  27n/64              (13/12)^3 n
```

Calculate work at each level:
Right side of recursion tree

Count total num of levels:
$\left(\frac{1}{3}\right)^0 n \rightarrow \left(\frac{1}{3}\right)^1 n \rightarrow \left(\frac{1}{3}\right)^2 n \rightarrow \left(\frac{1}{3}\right)^k n$
At last level, size of problem becomes 1 so, $(\frac{1}{3})^k n = 1$
So, $\left(\frac{1}{3}\right)^k = \frac{1}{n}$
Therefore, $\mathrm{k} = \log_3 n$

Count total number of nodes at last level:
$2^{\log_3 n} \rightarrow n^{\log_3 2}$

Sum up the cost of all levels in the tree:
$\mathrm{T(n)} = \mathrm{n} + \mathrm{n} + \mathrm{n} + \ldots + \log_3 n + \theta(n^{\log_3 2})$
$\mathrm{T(n)} = \mathrm{n}\log_3 n + \theta(2^{\log_3 n})$

Therefore, T(n) = $\boldsymbol{\theta(n \log_3 n)}$

P1 (20pt): You are given n stones (assume that n is a power of 2) each having a distinct weight. You are also given a two-pan balance scale (no weights are given). For example, given two stones, you can use the scale to compare which one is lighter by placing the two stones on the two different pans. The goal is the find the heaviest and the lightest stone by using as few weighings as possible. Give a divide and conquer strategy that uses only $\frac{3n}{2} - 2$ weighings, or less.

For this problem, we will start by comparing every stone in the original sample and separate each stone into a section having heavier stones and lighter stones, we will separate them into section H and L respectively. When we do this, we will make **n/2** comparisons to assign H and L sections. With that, we will start by finding the heaviest stone. We will do this by taking 2 stones in the H section and assign the heavier of the 2 as x. We will compare x with all other stones in H section, if the stone is bigger than x, x will be replaced with the heavier stone and if the stone is smaller than x, we will keep x and remove smaller stone from the H section. With this, there will be **(n/2)-1** comparisons. We do the same for the L section.

Pseudocode:

```
Index = 0
Split_array(stones)
    If(stone1 > stone2)
        Stone1 -> H
        Stone2 -> L
    Else
        Stone1 -> L
        Stone2 -> H
    Index = index + 2

Find_biggest(H)
    Max = H[0]
    For stones in H
        If(stone > H[0])
            Max = stone
    Return Max
```

So, the equation for total weighing's = **n/2 + (n/2) - 1 + (n/2) - 1**

Which will simplify to 3n/2 - 2

P2 (20pt): (a) Given two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(\log n)$ time.

(b) **[optional]** Given two sorted arrays $A[1 \ldots m]$ and $B[1 \ldots n]$ and an integer k. Describe an algorithm to find the k-th smallest element in the union of A and B (i.e. $A \cup B$) in $\Theta(\log(m+n))$ time. [Hint: use your solution in part (a)].

For this algorithm, we will calculate the median of both arrays, lets call these variables m1 and m2. If m1 > m2, then that means the median is present in either the range from the first element of A to m1 or from m2 to the last element of B. If m2 > m1, then that means that median is present in either the range from m1 to the last element of A or from the first element of B to m2. We repeat this process until both arrays are of size of 2 and then we will calculate the true median of both arrays with median = (max(A[0], B[0]) + min(A[1], B[1])) / 2.

Pseudocode:

```
Find_median(A, B)
    If size of A & B = 2
        Return (max(A[0], B[0]) + min(A[1], B[1]))/2
    Else if median(A) > median(B)
        Return Find_median(A[first element of A to median(A)], B[median(B) to last element of B])
    Else if median(B) > median(A)
        Return Find_median(A[median(A) to the last element of A], B[first element of B to median(B)])
```

This will result in an algorithm with a run-time complexity of $\theta(logn)$

P3 (20pt): You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an length n array of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the array (left and right). Describe an $O(\log n)$ algorithm that finds a number that is bigger than either of its neighbors.

For this problem, in order to get a runtime complexity of O(logn) we would need to ignore half of the array since there can possibly be multiple answers in the array. We will initialize indexes L and R (left and right) to 0 and to the end index of the array, respectively. We will define a function called find_box that takes parameters of the original array, the beginning index and the ending index. In the function find_box, we will initialize a variable called M (mid) that will take the middle element of L and R with M = (L + (R - L))/2. We find a box that beats our neighbors in 3 ways;

if M = 0 and array[M] > array[M+1]
if M = size(array) and array[M] > array[M-1]
or if array[M] > array[M-1] AND array[M] > array[M+1]

If none of these requirements are met, we will use an else if function that says
if array[M] < array[M+1], then L = array[M+1]
Or if array[M] < array[M-1], then R = array[M-1]
And we will recursively call this function until we find a box that is bigger than it's neighbors.

```
Pseudocode:
Array = array of boxes
L = 0
R = sizeof(array)
Find_box(array, L, R)
    M = (L + (R - L))/2
    If (M = 0 or array[M] > array[M-1]) and (M = sizeof(array) or array[M] > array[M+1])
        Return array[M]
    Else if (array[M] < array[M+1]
        L = array[M+1]
        Return Find_box(array, L, R)
    Else if (array[M] < array[M-1]
        R = array[M-1]
        Return Find_box(array, L, R)
```

This will result in an algorithm that is of time complexity of $\theta(logn)$

P4 (20pt): You are given an $n \times n$ matrix represented as a 2-dimensional array $A[1..n][1..n]$ (there are totally n^2 elements). Each row of A is sorted in increasing order and each column is sorted in increasing order. Your goal is to find whether some given element x is in A or not. Note that you can do only comparisons between elements (no hash table, or any other set data structures). Give an algorithm that takes $O(n)$ comparisons. [Hint: try to start on one some corner, and eliminate one row/column per move].

Since the rows/columns are sorted in increasing from left to right and top to bottom, we will evaluate each row and column to see if the value x is the range of that sorted row by taking the range of each row/column. We can start on the top right corner of the matrix and work backwards.

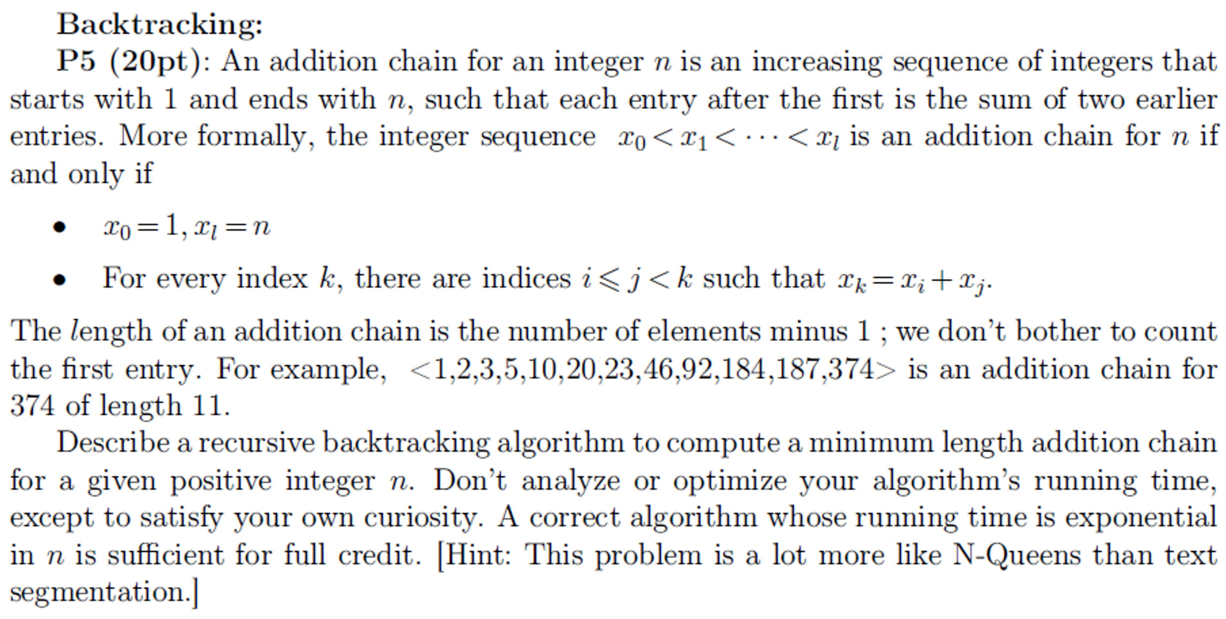
The function will check if the current index of the matrix is equal to the value we are trying to find.
If the value is smaller than the current index of the matrix, then we will get rid of that column since that would mean all values in that column are larger than the value we are looking for.
If the value is larger than the current index of the matrix, then we will get rid of that row since that would mean all values in that row are smaller than the value we are looking for.

We recursively call a function that performs these comparisons until we find the element x in the matrix. The function will take in parameters of the matrix, the index of the row, the index of the column, and the value we are looking for.

Pseudocode:

```
Initialize matrix[1…n][1…n]
Find_x(matrix, r=0, c=n, x)
    If (matrix[r][c] == x)
        Return matrix[r][c]
    Else
        If (matrix[r][c] < x and (r+1) < n)
            Return Find_x(matrix, r+1, c, x)
        If (matrix[r][c] > x and (c-1) > 0)
            Return Find_x(matrix, r, c-1, x)
    Return not in matrix
```

There will be one comparison in each recursive call and since n = all elements in the matrix, the runtime complexity of this algorithm is O(n) where O(n) is the worst case scenario



Backtracking:

P5 (20pt): An addition chain for an integer n is an increasing sequence of integers that starts with 1 and ends with n, such that each entry after the first is the sum of two earlier entries. More formally, the integer sequence $x_0 < x_1 < \cdots < x_l$ is an addition chain for n if and only if

- $x_0 = 1, x_l = n$
- For every index k, there are indices $i \leqslant j < k$ such that $x_k = x_i + x_j$.

The *length* of an addition chain is the number of elements minus 1 ; we don't bother to count the first entry. For example, $\langle$1,2,3,5,10,20,23,46,92,184,187,374$\rangle$ is an addition chain for 374 of length 11.

Describe a recursive backtracking algorithm to compute a minimum length addition chain for a given positive integer n. Don't analyze or optimize your algorithm's running time, except to satisfy your own curiosity. A correct algorithm whose running time is exponential in n is sufficient for full credit. [Hint: This problem is a lot more like N-Queens than text segmentation.]

For this algorithm, we will use backtracking as a technique to find the shortest path for a number n. We know that the first element will be 1 so we have a solid first element of the vector. We will use two vectors, one for testing and one for the answer. We will define a function backtracking that will take a single input of an integer. We will start by calling the function and initializing it to 1 and we will insert 1 into the test vector. We will also have a value that will store the vector with the smallest size called min. The main operation that is happening in the backtracking function is that we will use a for loop to iterate through each element before (and including) the current index and perform a couple of things. We add the iterated index of the vector to the current index, if the summation is less than or equal to n, then we will push that value into the test vector and recursively call the backtracking function.

There will be a statement in the same for loop to pop and element in case this recursive path is not the minimum or does not meet the criteria of an addition chain. There will be an if statement before the for loop in the function that will check if the size of the vector is less than min, if it is not then we return nothing. There will be one more if statement that will check and see if the current index is EQUAL to n. If it is, then we will check if this current vector size is smaller than the current min. If that is the case, we will clear the answer vector for any previous solution and update the ans vector to the current addition chain.

We will globally declare both vectors, n, and min so we can use these variables in both the main function and the backtracking function.

Pseudocode:

```
Vectors v = [], ans = []
Integers n, min

Backtracking(int k)
    If (v.size >= min)
        Return nothing
    If (k == n)
        If(v.size < min)
            Min = v.size()
            Clear ans
            Ans = v

    For(i = v.size; i >= 0; i--)
        If(k + v[i] <= n)
            V.push(k + v[i])
            Backtracking(k + v[i])
            V.pop()
```