Solutions for HW6 - CS 6033 Fall 2021

Q1 → Master Theorem for recurrence formulae

1. Solve these recurrence formulas using Θ notation:

- $T(n) = 2T(n/3) + 1$
- $T(n) = 5T(n/4) + n$
- $T(n) = 7T(n/7) + n$
- $T(n) = 9T(n/3) + n^2$
- $T(n) = 8T(n/2) + n^3$
- $T(n) = 7T(n/2) + \Theta(n^2)$
- $T(n) = T(n/2) + \Theta(1)$
- $T(n) = 5T(n/4) + \Theta(n^2)$

Recurrence formula	a	b	$n^{\log_b a}$	f(n)	Case	T(n)
$2T(n/3) + 1$	2	3	$n^{\log_3 2}$	1	1 → cost dominated by leaves	$\Theta(n^{\log_3 2})$
$5T(n/4) + n$	5	4	$n^{\log_4 5}$	n	1 → cost dominated by leaves	$\Theta(n^{\log_4 5})$

$7T(n/7)\ +\ n$	7	7	$n^{log_7 7}$	n	2 → cost same at every level	$\Theta(nlogn)$
$9T(n/3)\ +\ n^2$	9	3	$n^{log_3 9}$	n^2	2 → cost same at every level	$\Theta(n^2 logn)$
$8T(n/2)\ +\ n^3$	8	2	$n^{log_2 8}$	n^3	2 → cost same at every level	$\Theta(n^3 logn)$
$7T(n/2)\ +\ \Theta(n^2)$	7	2	$n^{log_2 7}$	n^2	1 → cost dominated by leaves	$\Theta(n^{log_2 7})$
$T(n/2)\ +\ \Theta(1)$	1	2	$n^{log_2 1}$	1	2 → cost same at every level	$\Theta(logn)$
$5T(n/4)\ +\ \Theta(n^2)$	5	4	$n^{log_4 5}$	n^2	3 → cost dominated by root	$\Theta(n^2)$

Q2 → Master Theorem for divide-conquer problems

2. Suppose you came up with three solutions to a homework problem:

- The first solution, Algorithm A, divides the original problem into 5 subproblem of size $n/2$, recursively solves the subproblems, and then solves the original problem by combining the subproblems in linear time.
- The second solution, Algorithm B, divides the original problem into two subproblems of size $\frac{9}{10}n$, recursively solves the subproblems, and then solves the original problem by combining the subproblems in linear time.
- The third solution, Algorithm C, divides the original problem into problems of size $n/3$, recursively solves the subproblems and then solves the original problem by combining the subproblems in $\Theta(n^2)$ time

Provide the recurrence formula for each of the algorithms. What are the running times of each of these algorithms (in Θ notation), and which of your algorithms is fastest?

Recurrence formula	**a**	**b**	$n^{log_b a}$	**f(n)**	**Case**	**T(n)**
$5T(n/2)\ +\ \Theta(n)$	5	2	$n^{log_2 5}$	n	1 → cost dominated by leaves	$\Theta(n^{log_2 5})$
$2T(9n/10)\ +\ \Theta(n)$	2	10/9	$n^{log_{10/9} 2}$	n	1 → cost dominated by leaves	$\Theta(n^{log_{10/9} 2})$
$3T(n/3)\ +\ \Theta(n^2)$	3	3	$n^{log_3 3}$	n^2	3 → cost dominated by root	$\Theta(n^2)$

Comparing the powers of n for the running times, we find that $2\ <\ log_2 5\ <\ log_{10/9} 2$. This shows us that Algorithm C is the fastest.

(P.S: Since we didn't say it clearly for algorithm C - we will accept answers when a = 1)

Q3 → Matrix Multiplication 1

3. Matrix multiplication:

- Divide the 4×4 matrix A matrix into 4 smaller matrices of size 2×2:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \text{ to create: } \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

  Show the 2×2 matrices A_{11}, A_{12}, A_{21}, and A_{22}.

- Perform some of the calculations needed to compute $A \times B$ using Strassen's algorithm:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \cdot \begin{bmatrix} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{bmatrix} = C$$

  For the matrices given above:
  - Compute P_1, P_2, and C_{12}
  - Compute $A_{11} \cdot B_{12}$ and $A_{12} \cdot B_{22}$
  - Check to see that $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$.

- Verify that $C_{22} = P_5 + P_1 - P_3 - P_7$ by replacing each P_i with its value and reducing the expression.

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} A_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 17 & 18 \\ 21 & 22 \end{bmatrix} B_{12} = \begin{bmatrix} 19 & 20 \\ 23 & 24 \end{bmatrix} B_{21} = \begin{bmatrix} 25 & 26 \\ 29 & 30 \end{bmatrix} B_{22} = \begin{bmatrix} 27 & 28 \\ 31 & 32 \end{bmatrix}$$

$$P_1 = A_{11} * (B_{12} - B_{22}) = \begin{bmatrix} -24 & -24 \\ -88 & -88 \end{bmatrix}$$

$$P_2 = (A_{11} + A_{12}) * B_{22} = \begin{bmatrix} 294 & 304 \\ 758 & 784 \end{bmatrix}$$

$$C_{12} = P_1 + P_2 = \begin{bmatrix} 270 & 280 \\ 670 & 696 \end{bmatrix}$$

$$A_{11} * B_{12} = \begin{bmatrix} 65 & 68 \\ 233 & 244 \end{bmatrix}$$

$$A_{12} * B_{22} = \begin{bmatrix} 205 & 212 \\ 437 & 452 \end{bmatrix}$$

$$(A_{11} * B_{12}) + (A_{12} * B_{22}) = \begin{bmatrix} 270 & 280 \\ 670 & 696 \end{bmatrix} = C_{12}$$

There are two ways to prove $C_{22} = P_5 + P_1 - P_3 - P_7$. You can prove it using the values given for A and B or using a generic method. Both will be shown here.

$$P_1 = A_{11} * (B_{12} - B_{22}) = A_{11} * B_{12} - A_{11} * B_{22} = \begin{bmatrix} -24 & -24 \\ -88 & -88 \end{bmatrix}$$

$P_3 = (A_{21} + A_{22}) * B_{11} = A_{21} * B_{11} + A_{22} * B_{11} = \begin{bmatrix} 802 & 844 \\ 1106 & 1164 \end{bmatrix}$

$P_5 = (A_{11}+A_{22})*(B_{11}+B_{22}) = A_{11}*B_{11}+A_{22}*B_{11}+A_{11}*B_{22}+A_{22}*B_{22} = \begin{bmatrix} 1256 & 1308 \\ 2024 & 2108 \end{bmatrix}$

$P_7 = (A_{11}-A_{21})*(B_{11}+B_{12}) = A_{11}*B_{11}+A_{11}*B_{12}-A_{21}*B_{11}-A_{21}*B_{12} = \begin{bmatrix} -640 & -672 \\ -640 & -672 \end{bmatrix}$

$C_{22} = A_{21} * B_{12} + A_{22} * B_{22} = \begin{bmatrix} 1070 & 1112 \\ 1470 & 1528 \end{bmatrix}$

Using the values, we get

$P_5 + P_1 - P_3 - P_7 = \begin{bmatrix} 1070 & 1112 \\ 1470 & 1528 \end{bmatrix} = C_{22}$

Using the generic method, we get

$$\begin{aligned} & P_5 + P_1 - P_3 - P_7 \\ &= A_{11} * B_{11} + A_{11} * B_{12} + A_{11} * B_{22} - A_{11} * B_{22} + A_{22} * B_{11} + A_{22} * B_{22} \\ &- (A_{11} * B_{11} + A_{11} * B_{12} - A_{21} * B_{11} + A_{21} * B_{11} + A_{22} * B_{11} - A_{21} * B_{12}) \\ &= A_{22} * B_{22} - (-A_{21} * B_{12}) = A_{22} * B_{22} + A_{21} * B_{12} = C_{22} \end{aligned}$$

Q4 → Matrix Multiplication 2

4. Design an efficient algorithm to multiply a $n \times 3n$ matrix with a $3n \times n$ matrix where you use Strassen's algorithm as a subroutine. Justify your run time. No points will be given for an inefficient algorithm.

An n x 3n matrix will look like $\begin{bmatrix} A & B & C \end{bmatrix}$ while a 3n x n matrix will look like $\begin{bmatrix} D \\ E \\ F \end{bmatrix}$, where A, B, C, D, E, and F are n x n matrices.

The most efficient algorithm will look like the following:

$$\begin{bmatrix} A & B & C \end{bmatrix} * \begin{bmatrix} D \\ E \\ F \end{bmatrix} = \begin{bmatrix} A * D & B * E & C * F \end{bmatrix}$$

This will call Strassen's algorithm for each multiplication of the n x n matrix. Therefore, the run-time is equal to Strassen's algorithm times 3. So, the algorithm is effectively $O(n^{\log_2 7})$.

```
MULTIPLY_MATRICES(A, B):
    answer = [] // This is an empty n x n matrix
    for i=0:2
        answer = answer + STRASSEN(A[1:n,i*n + 1:(i+1)*n], B[i*n +
1:(i+1)*n,1:n])
    return answer
```

Q5 → Substitution Method 1

5. Use the substitution method to prove that $T(n) = 2T(n/2) + cn \log n$ is $O(n \log^2 n)$.

$$Prove: T(n) \leq dnlog^2 n$$
$$IH: T(k) \leq dklog^2 k \ for \ k < n \ and \ d > 0 \ and \ c > 0$$
$$T(n) \leq 2d(n/2)(log(n/2))^2 + cnlogn$$
$$T(n) \leq dn(logn - log2)^2 + cnlogn$$
$$T(n) \leq dn(logn - 1)^2 + cnlogn$$
$$T(n) \leq dn(log^2 n - 2logn + 1) + cnlogn$$
$$T(n) \leq dnlog^2 n - (2d - c)nlogn + dn$$
$$Since \ our \ base \ case \ will \ be \ n \geq 2 \rightarrow dn \leq dnlogn$$
$$T(n) \leq dnlog^2 n - (2d - c)nlogn + dnlogn$$
$$T(n) \leq dnlog^2 n - (d - c)nlogn$$
$$If d \geq c, then T(n) \leq dnlog^2 n$$
$$Therefore, T(n) \ is \ O(nlog^2 n)$$

Q6 → Substitution Method 2

6. Use the substitution method to prove that if $T(n) = 2T(n-1) + 3$ and $T(1) = 1$ then $T(n)$ is $O(2^n)$.

We can prove it using two methods.
Method 1 → Guess & Check

$$Prove: T(n) \leq d2^n$$
$$IH: T(k) \leq d2^k - c \ for \ k < n \ and \ d > 0 \ and \ c > 0$$
$$T(n) \leq 2 * (d2^{n-1} - c) + 3$$
$$T(n) \leq d2^n - (2c - 3)$$
$$If \ c \geq 3, \ then \ T(n) \leq d2^n - c$$
$$Therefore, T(n) \ is \ O(2^n)$$

Method 2 → Recursive Substitution

$$T(n) = 2T(n-1) + 3$$
$$T(n) = 2[2T(n-2) + 3] + 3 = 2^2 T(n-2) + 3(1+2)$$
$$T(n) = 2^2[2T(n-3) + 3] + 3(1+2) = 2^3 T(n-3) + 3(1+2+2^2)$$

$$T(n) = 2^k T(n-k) + 3(1 + 2 + 2^2 + ... + 2^{k-1})$$

$$T(n) = 2^{n-1} T(1) + 3(1 + 2 + 2^2 + ... + 2^{n-2})$$
$$We \ know \ that \ T(1) = 1$$
$$Therefore \ T(n) = 2^{n-1} + 3(1 + 2 + 2^2 + ... + 2^{n-2})$$
$$T(n) = 2^{n-1} + 3[(2^{n-1} - 1)/(2 - 1)] = 2^{n-1} + 3(2^{n-1} - 1)$$
$$T(n) = 2^{n-1} + 3 * 2^{n-1} - 3 = 2^2 * 2^{n-1} - 3 = 2 * 2^n - 3$$
$$Therefore, T(n) \ is \ O(2^n)$$

Q7 → Skyline

7. Suppose you have a geometric description of the buildings of Manhattan and you would like to build a representation of the New York skyline. That is, suppose you are given a description of a set of rectangles, all of which have one of their sides on the x-axis, and you would like to build a representation of the union of all these rectangles.

   Formally, since each rectangle has a side on the x-axis, you can assume that you are given a set, $S = \{[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]\}$ of subintervals in the interval $[0, 1]$, with $0 \leq a_i < b_i \leq 1$, for $i = 1, 2, \ldots, n$, such that there is an associated height, h_i , for each interval $[a_i, b_i]$ in S. The skyline of S is defined to be a list of pairs $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \ldots, (x_m, c_m), (x_{m+1}, 0)]$, with $x_0 = 0$ and $x_{m+1} = 1$, and ordered by x_i values, such that, each subinterval, $[x_i, x_{i+1}]$, is the maximal subinterval that has a single highest interval, which is at height c_i , in S, containing $[x_i, x_{i+1}]$, for $i = 0, 1, ..., m$.

   Design (using pseudo-code) an $O(n \log n)$- time algorithm for computing the skyline of S. Justify the running time of your algorithm.

   Question from Goodrich, Michael T.; Tamassia, Roberto. Algorithm Design and Applications

   This question uses techniques from previous lectures.

For this question, there may be a variety of solutions. One solution is to use divide and conquer.

Firstly divide the city arbitrarily into two sets (each has a size of n/2), then we recursively find the skyline for each set. Finally, merge two sets of skylines(A, B).

Steps:

1. Divide → Given the length of the array, we can divide the array in O(1).
2. Conquer → base case should be there only be one range. So the skyline is composed of two points (a1, h1), (b1, 0): C_x = [a1, b1], C_h = [h1, 0]
3. Merge → for this part, assuming A_x, B_x are sorted x-coordinates in two sets. And A_h, B_h is their corresponding heights.
   How can we get merged result C:
   scan A_x, B_x from left to right: Each time focus on the array with smaller x value(x = min(A_x[i], B_x[i]))
   case 1: A_x[i] = B_x[j], C_h[k] should be max(A_x[i] == B_x[j])
   case 2: A_x[i] < B_x[j]: the height of skyline A at x should be A_x[i], but the height of skyline B at x should still be B_x[j-1]. So C_h[k] should be max(A_x[i] = B_x[j-1])
   case 3: A_x[i] > B_x[j]: similar to case 1

Note:
1. C_h[k-1] contains a height that starts at x = C_x[k-1] and whose endpoints are not yet determined. And A_x[i], B_x[j] are both greater than C_x[k-1]. So we need to check if the current max_h is the same as the previous height in C_x, we don't need to add it again.
2. Since in the base case, A and B must be sorted. And we can see that, in SKYLINE-MERGE, given sorted A, B, we will also get a sorted C. So the assumption that A and B are sorted is valid.

Since the combining part will only go through A, B set one time, the time complexity should be O(n). The recurrence formula for the whole algorithm is $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = O(nlogn)$.

```
// let skyline.x be a list of x-coordinates and skyline.h be a list of
heights
GET-SKYLINE(S, H):
    n = len(S)
    // base case
    if n == 1:
        skyline.x = [S[1][1], S[1][2]] // if there is only one building we
have two skyline points
        skyline.h = [H[1], 0]
        return skyline
    // not base case - divide and conquer
    left_skyline = GET-SKYLINE(S[:n//2], H[:n//2])
    right_skyline = GET-SKYLINE(S[n//2:], H[n//2:])
    return SKYLINE-MERGE(left_skyline, rightskyline)

// Merge two sets of skylines
SKYLINE-MERGE(A, B):
    i = j = k = 1
    m = len(A.x)
    n = len(B.x)
    while i <= n or j <= m:
        x = min(A.x[i], B.x[j])
        if A.x[i] < B.x[j]:
            max_h = max(A.h[i], B.h[j-1]) // suppose if j-1= 0, then B.h is
-inf
            i = i + 1
        else if A.x[i] > B.x[j]:
            max_h = max(A.h[i-1], B.h[j]) // suppose if i-1= 0, then A.h is
-inf
            j = j + 1
        else // A.x[i] = B.x[j]:
            max_h = max(A.h[i], B.h[j])
            i = i + 1
            j = j + 1
        if k == 1 or max_h != C.h[k-1]:
            C.x[k] = x
            C.h[k] = max_h
            k += 1
    return C
```

```
DRAW-SKYLINE(S, H):
    res = GET-SKYLINE(S,H)
    n = len(S)
    // according to the question, we need to make sure two points at x=0
and x= 1
    if res.x[1] != 0:
        res.x = [0] + res.x
        res.h = [0] + res.h
    if res.x[n] != 1:
        res.x = res.x + [1]
        res.h = res.h + [0]
    return res
```

FYI: One good solution with heap:
https://leetcode.com/problems/the-skyline-problem/discuss/741467/Python3-priority-queue