Brandon Vo

1. Solve these recurrence formulas using Θ notation:

- T(n) = 2T(n/3) + 1

$$Master\ method:\ n^{\log_3 2}\,3\ \ vs. O(1)$$

$$Case\ 1: \theta(n^{0.63})$$

- T(n) = 5T(n/4) + n

$$Master\ Method: n^{\log_4 5} >\ O(n)$$

$$Case\ 1: \theta(n^{1.16})$$

- T(n) = 7T(n/7) + n

$$Master\ Method: n^{\log_7 7} == O(n)$$

$$Case\ 2:\ \theta(n \log n)$$

- T(n) = 9T(n/3) + n^2

$$Master\ Method: n^{\log_3 9} == \theta(n^2)$$

$$Case\ 2:\ \theta(n^2 \log n)$$

- T(n) = 8T(n/2) + n^3

$$Master\ Method: n^{\log_2 8} == n^3$$

$$Case\ 2:\ \theta(n^3 \log n)$$

- T(n) = 7T(n/2) + Θ(n^2)

$$Master\ Method: n^{\log_2 7} > n^2$$

$$Case\ 1:\ \theta(n^{2.807})$$

- T(n) = T(n/2) + Θ(1)

$$Master\ Method: n^{\log_2 1} == \theta(1)$$

$$Case\ 2:\ \theta(\log n)$$

- T(n) = 5T(n/4) + Θ(n^2)

$$Master\ Method: n^{\log_4 5} < n^2$$

$$Case\ 3:\ \theta(n^2)$$

Brandon Vo

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.

$$T(n) = 5 * T\left(\frac{n}{2}\right) + n$$

$$Master\ Method: n^{\log_2 5} > n$$

$$\theta(n^{2.32})$$

• The second solution, Algorithm B, divides the original problem into two subproblems of size 9/10*n, recursively solves the subproblems, and then solves the original problem by combining the subproblems in linear time.

$$T(n) = 2T\left(\frac{9n}{10}\right) + n$$

$$Master\ Method: n^{lo\ \ {}_{9/10}\ 2} < n$$

$$\theta(n)$$

• 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

$$T(n) = 3T\left(\frac{n}{3}\right) + \theta(n^2)$$

$$Master\ Method: n^{\log_3 3} < n^2$$

$$\theta(n^2)$$

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?

The second solution is the fastest because it runs in Θ (n) time.

Brandon Vo

3. Matrix multiplication:

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

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

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

$$A_{11}: \begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix}$$

$$A_{12}: \begin{matrix} 3 & 4 \\ 7 & 8 \end{matrix}$$

$$A_{21}: \begin{matrix} 9 & 10 \\ 13 & 14 \end{matrix}$$

$$A_{22}: \begin{matrix} 11 & 12 \\ 15 & 16 \end{matrix}$$

∗Many of these questions came from outside sources.

• Perform some of the calculations needed to compute A × B using Strassen's algorithm:

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

For the matrices given above:

– Compute P_1, P_2, and C_{12}

$$P_1 = A_{11} * (B_{12} - B_{22})$$

$$P_1 = \begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix} * \left(\begin{matrix} 19 & 20 \\ 23 & 24 \end{matrix} - \begin{matrix} 27 & 28 \\ 31 & 32 \end{matrix}\right)$$

$$P_1 = \begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix} * \begin{pmatrix} -8 & -8 \\ -8 & -8 \end{pmatrix}$$

$$P_1 = \begin{matrix} -24 & -24 \\ -88 & -88 \end{matrix}$$

Brandon Vo

$$P_2 = (A_{11} + A_{12})B_{22}$$

$$P_2 = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix. + \begin{matrix} 3 & 4 \\ 7 & 8 \end{matrix}\Big) * \begin{matrix} 27 & 28 \\ 31 & 32 \end{matrix}$$

$$P_2 = \begin{matrix} 4 & 6 \\ 12 & 14 \end{matrix} * \begin{matrix} 27 & 28 \\ 31 & 32 \end{matrix} = \begin{matrix} 294 & 304 \\ 758 & 784 \end{matrix}$$

$$C_{12} = P_1 + P_2$$

$$C_{12} = \begin{matrix} -24 & -24 \\ -88 & -88 \end{matrix} + \begin{matrix} 294 & 304 \\ 758 & 784 \end{matrix} = \begin{matrix} \mathbf{270} & \mathbf{280} \\ \mathbf{670} & \mathbf{696} \end{matrix}$$

– Compute $A_{11} \cdot B_{12}$ and $A_{12} \cdot B_{22}$

$$A_{11} * B_{12} = \begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix} * \begin{matrix} 19 & 20 \\ 23 & 24 \end{matrix} = \begin{matrix} 65 & 68 \\ 233 & 244 \end{matrix}$$

$$A_{12} * B_{22} = \begin{matrix} 3 & 4 \\ 7 & 8 \end{matrix} * \begin{matrix} 27 & 28 \\ 31 & 32 \end{matrix} = \begin{matrix} 205 & 212 \\ 437 & 452 \end{matrix}$$

– Check to see that $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$.

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{12} = \begin{matrix} 65 & 68 \\ 233 & 244 \end{matrix} + \begin{matrix} 205 & 212 \\ 437 & 452 \end{matrix} = \begin{matrix} \mathbf{270} & \mathbf{280} \\ \mathbf{670} & \mathbf{696} \end{matrix}$$

C_{12} matches with the Strassen method

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

$$C_{22} = (A_{11} + A_{22})(B_{11} + B_{22}) + A_{11} * (B_{12} - B_{22}) - (A_{21} + A_{22})B_{11} - (A_{11} - A_{21})(B_{11} + B_{12})$$

$$C_{22} = \cancel{A_{11}B_{11}} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + A_{22}B_{22} + \cancel{A_{11}B_{12}} - \cancel{A_{11}B_{22}} - \cancel{A_{21}B_{11}} - \cancel{A_{22}B_{11}} - \cancel{A_{11}B_{11}} - \cancel{A_{11}B_{12}} + \cancel{A_{21}B_{11}} + A_{21}B_{12}$$

$$C_{22} = +A_{22}B_{22} + A_{21}B_{12}$$

$$C_{22} = \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}\begin{pmatrix} 27 & 28 \\ 31 & 32 \end{pmatrix} + \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix}\begin{pmatrix} 19 & 20 \\ 23 & 24 \end{pmatrix}$$

$$C_{22} = \begin{pmatrix} 669 & 692 \\ 901 & 932 \end{pmatrix} + \begin{pmatrix} 401 & 420 \\ 569 & 596 \end{pmatrix} = \begin{matrix} 1070 & 1112 \\ 1470 & 1528 \end{matrix}$$

Brandon Vo

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

- With nx3n * 3nxn, multiplying leads to a matrix of size nxn
- To optimize, we can split the 3n x n and n x 3n matrices into 3 different matrices of size n x n
- Then, we can run Strassen's algorithm for each of the three matrices against each other
  - We already know that Strassen's algorithm is $\theta(n^{2.81})$ time and that it takes $\theta(n^2)$ time to merge the set
- There will be an extra set of 3 multiplications and 2 sets of block additions needed to compute for Strassen's algorithm
- Merging would still take $\theta(n^2)$ to form the n x n resulting matrix which is still smaller than the $n^{\log2(7)}$ time needed to compute Strassen's algorithm

Aiming to use Strassen's formula when we can simplify it as

$$\begin{matrix} nk_1 & nk_2 & nk_3 \end{matrix} * \begin{matrix} nk_1 \\ nk_2 \\ nk_3 \end{matrix}$$

Which would require only an additional set of 3 multiplications and 2 sets of additions

$$T(n) = 3 * n^{\log_2 7} + n^2 + 2 \; where \; 3n^{\log_2 7} \; is \; asymptotically \; the \; largest$$

$$\theta(3n^{2.81}) \; which \; is \; still \; \theta(n^{2.81})$$

This would be more efficient than doing standard matrix multiplication which would require $\theta(n^3)$ time necessary to perform matrix multiplication.

```
MULTIPLY(MATRIX_1, MATRIX_2)

//MATRIX_1 holds n x 3n

// MATRIX_2 holds 3n x n


//Split into 3 smaller matrices of size n x n

MATRIX_A.append(MATRIX_1[n, 1:n])

MATRIX_A.append(MATRIX_1[n, n:2n])

MATRIX_A.append(MATRIX_1[n, 2n:3n])


MATRIX_B.append MATRIX_2[1:n, n]
```

Brandon Vo

MATRIX_B.append MATRIX_2[n:2n, n]

MATRIX_B.append MATRIX_2[2n:3n, n]

For i = 1 to MATRIX_A.length

Run STRASSEN'S ALGORITHM and Merge on (MATRIX_A[i] * MATRIX_B[i])

Brandon Vo

5. Use the substitution method to prove that T(n) = 2T(n/2) + cn log n is O(n log²n).

$$T(n) \leq dn \log^2 n - d'n * \log n$$

$$\boldsymbol{IH}\text{: } T(k) \leq dk \log^2 k$$

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \log n$$

$$\leq 2d\left(\frac{n}{2} * \log^2 \frac{n}{2}\right) + cn \log n$$

$$\leq d(n\, log\, n - n \log 2)^2 + cn \log n$$

$$\leq dn\ (\log n - 1)^2 + cn \log n$$

$$\leq dn(\log^2 n - 2 \log n + 1) + cn \log n$$

$$\leq dn \log^2 n - 2dn \log n + dn + cn \log n$$

$$\leq dn \log^2 n + (cn \log n - 2dn \log n) + dn$$

$$= \boldsymbol{dn \log^2 n - n \log n\, (c - 2d) + dn}\ where\ (c - 2d) \geq 0$$

Brandon Vo

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

$$\boldsymbol{IH}: T(n) \leq d2^n - 2c$$

$$T(n) = 2T(n-1) + 3$$

$$T(n) \leq 2(d2^{n-1} - 2c) + 3$$

$$T(n) \leq d2^n - 4c + 3$$

$$T(n) \leq 2^n \; where \; c > 0$$

$$If \; T(1) = 1, then \; c > d$$

Brandon Vo

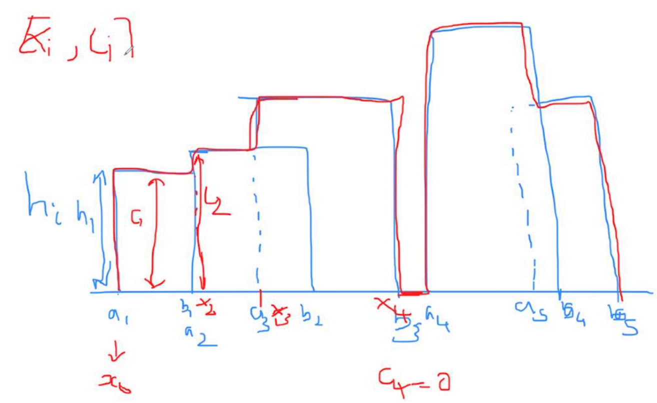
7. Suppose you have a geometric description of the buildings of Manhattan and you would like to build a representation of the New York skyline. That is, suppose you are given a description of a set of rectangles, all of which have one of their sides on the x-axis, and you would like to build a representation 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 $0 \le a_i < b_i \le 1$, 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.



- Given the set S from a, b_1 to a_n,b_n
- Given a set of sub-intervals from S
    - Each of those intervals represent 1 building
    - Each building has a height h_i
    - We're searching for intervals where buildings might intersect and to trace their heights
    - We're only picking the x value where the height changes
    - Overlapping lines do not count
- Each block has a height h_i associated with it as well
- The output should have no two pairs that share the same height
- Must include any gap between two non-overlapping buildings
- Will operate similarly to merge sort where we're building the data structure as we compute each skyline

- We keep subdividing the set of all rectangles in halves until we get only one pair of rectangles
- We compare the rectangles and merge them based on where the change in height occurs
- If they do not overlap, we have to add the extra gap between them
- Then, we can recursively merge the pairs and adding the heights
  - It takes 2T(n/2) to subdivide a problem into two smaller problems
  - It takes θ(n) time to merge two skylines

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

$$Master\ Method\ Case\ 2: n^{\log_2 2} == \ n$$

$$\theta(n \log n)$$

Create array HEIGHTS

Skyline(S, S.length, HEIGHTS)

//Divide the skylines into halves like merge-sort and solve them through merging

Skyline(S, n, HEIGHTS)

if n == 0

    Return Nil

If n == 1

    Return S

Else

    HALF_1 = S[1:n/2]

    SKYLINE(HALF_1, HALF_1.length, HEIGHTS)   //First half of the skylines

    HALF_2 = S[n/2:n]

    SKYLINE(HALF_2, HALF_2.length, HEIGHTS)   //Other half of the skylines

    MERGE(HALF_1, HALF_2, HEIGHTS)

//Merge the two skylines

MERGE(HALF_1, HALF_2, HEIGHTS)

If (Only 1 rectangle is present in both of HALF_1 and HALF_2)

```
        APPEND HALF_1 or HALF_2 to HEIGHTS
        Return HEIGHTS
M, N = 1        //Index for HALF 1 and HALF 2
Y = 0           //Keep track of our skyline

//Merge the two sets of skylines together
P = HALF_1[M]
Q = HALF_2[N]
While(M < HALF_1.length and N < HALF_2.length)
        //HALF_1 is empty
        If P == NIL
                //Just add HALF_2
                Insert Q into HEIGHTS
                Q = HALF_2[N++]
        //HALF_2 is empty
        Else if Q == NIL
                //Just insert HALF_1
                Insert P into HEIGHTS
                P = HALF_1[M++]
//The two buildings occupy the same x-coordinate width and must differ only by height
        Else If (P.A == Q.A and P.B == Q.B)
//Take the start and end x-coordinates and use the largest height
                Y = MAX(P.H, Q.H)
                Insert( [P.A, Q.B, Y] ) into HEIGHTS
                //Increment both
                Q = HALF_2[N++]
                P = HALF_1[M++]
        Else If (P and Q overlap)
                //Start with the furthest left rectangle
```

```
        If P.A < Q.A
            LEFT = P.A
        //Find the intersection between P and Q.  This is where the height changes
        // RIGHT = Intersection between (P and Q) X-coordinates
            RIGHT = MIN(P.B, Q.A)
            Insert (LEFT, RIGHT, P.h) into HEIGHTS
            P = HALF_1[M++]
    //We need to update Q to remove the gap and adjust for the intersection coordinates
            Q = (MAX[P.B, Q.A], Q.B, Q.h)
        Else    //Q.A < P.A
            LEFT = Q.A
            RIGHT = MIN(Q.B, P.A)
            Insert (LEFT, RIGHT, Q.h) into HEIGHTS
            Q = HALF_2[N++]
    //Readjust the rightmost building's coordinates and remove the gap overlap
    //This will be the building we compare with for the next iteration of the merge
            P = (MAX[Q.B, P.A], P.B, P.h)
    Else //P and Q do not overlap
//There are 3 elements:  P, Q, and the gap between them
//Insert the leftmost one
        If (P.A < Q.A)
            Insert P into HEIGHTS
            LEFT = P.B
            RIGHT = Q.A
            P = (LEFT, RIGHT, 0)        //Move into the gap
//Copy the gap into our leftmost spot and give it a height of 0
//Must track this gap spot with the next rectangle in case there's an extra rectangle between them
which wasn't found yet during our merge
        Else
```

Insert Q into HEIGHTS

LEFT = Q.B

RIGHT = P.A

Q = (LEFT, RIGHT, 0) //Move into the gap

EndWhile

Return HEIGHTS

Brandon Vo

8. (3 bonus points) Think of a good exam/homework question for the material covered in Lecture .

We are presented with a problem that can be solved in O(n^2) time. How many subdivisions can we split this problem and retain the most efficient time complexity of $T(n) = K\,T\left(\frac{n}{K}\right) + O(n^2)$