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) + n2

• T(n) = 8T(n/2) + n3

• T(n) = 7T(n/2) + Θ(n2)

• T(n) = T(n/2) + Θ(1)

• T(n) = 5T(n/4) + Θ(n2)

1. 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 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 Θ(n2) 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?

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

1. Matrix multiplication:

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

to create :

Show the 2 × 2 matrices A11, A12, A21, and A22.

∗Many of these questions came from outside sources.

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

For the matrices given above:

– Compute P1, P2, and C12

– Compute A11 · B12 and A12 · B22

– Check to see that C12 = A11 · B12 + A12 · B22.

C12 matches with the Strassen method

• Verify that C22 = P5+P1−P3−P7 by replacing each Pi with its value and reducing the expression.

1. 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 θ(n2.81) time and that it takes θ(n2) 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 θ(n2) to form the n x n resulting matrix which is still smaller than the nlog2(7) time needed to compute Strassen’s algorithm

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

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

This would be more efficient than doing standard matrix multiplication which would require θ(n3) 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]

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])

1. Use the substitution method to prove that T(n) = 2T(n/2) + cn log n is O(n log2n).
2. Use the substitution method to prove that if T(n) = 2T(n − 1) + 3 and T(1) = 1 then T(n) is O(2n).
3. 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 = {[a1, b1], [a2, b2], ..., [an, bn]} of subintervals in the interval [0, 1], with 0 ≤ ai < bi ≤ 1, for i = 1, 2, . . . , n, such that there is an associated height, hi, for each interval [ai, bi] in S. The skyline of S is defined to be a list of pairs [(x0, c0),(x1, c1),(x2, c2), . . . ,(xm, cm),(xm+1, 0)], with x0 = 0 and xm+1 = 1, and ordered by xi values, such that, each subinterval, [xi , xi+1], is the maximal subinterval that has a single highest interval, which is at height ci , in S, containing [xi , xi+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.

Diagram

Description automatically generated

* Given the set S from a, b1 to an,bn
* Given a set of sub-intervals from S
  + Each of those intervals represent 1 building
  + Each building has a height hi
  + 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 hi 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

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

1. (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(n2) time. How many subdivisions can we split this problem and retain the most efficient time complexity of