Elijah Chou

CS 526

Dr. Michelangelo Grigni

9/8/2023

**HW 1**

1. A) T(n) = 2 \* T(n/2) + O(n lg n)   
   In the tree analysis for this recursion, each level does O(n lg n) work. At the top level, it is a given that the work done is O(n lg n). To simplify the analysis, we can use k = lg n to substitute out the lg n. Each level down, the total work is equal to n \* (k – x), where x is equal to the level in the tree. In the sum of all work done on all levels, we can factor out the common n and get a sum of n \* (k – 1 + k – 2 + k – 3 + … + 1), which can simplify to n \* (k(k-1))/2. Substituting k = lg n again gives us **n \* (lg n \* lg (n-1)/2 = O(n lg^2 n).**

B) T(n) = 2 \* T(n/2) + O(n/lg n)  
In the tree analysis for this recursion, we can substitute k = lg n again and get the recursion T(n) = 2 \* T(n/2) + O(n/k). After drawing out the tree analysis, we get that each level of the tree will have a total work of n/(k-x), where x is equal to the level of the tree. Writing out the sum of all work done in all levels, we can factor out the common n factor and get the sum of n \* (1/(k-1) + 1/(k-2) + 1/(k-3) + … + 1), which we can then replace with the Harmonic number sum that equates to ln k + O(1). Substituting in the Harmonic number sum, we get that the total work done is n \* ln (k) + O(1), and after substituting k = lg n back in, we get that the big-Oh notation for this recurrence is **O(n ln (lg n)) + O(1) = O(n lg lg n).**

C) T(n) = + O(n)

Let n = 2^k, = 2^(k/2), k = log n. We can change the recurrence to be T(2^k) = 2^(k/2) \* T(2^(k/2)) + 2^k. Divide everything by 2^k and we get T(2^k)/2^k = (2^(k/2) \* T(2^(k/2)) / 2^k + 1, which simplifies to T(2^k) / 2^k = T(2^(k/2)) / 2^(k/2) + 1. If y(k) = T(2^k) / 2^k, then we can simplify it to y(k) = y(k/2) + 1. Applying Master Theorem here, we get that a = 1, b = 2, and d = 0. Since , we get the big Oh notation for y(k) as O(log k). Since T(2^k) = 2^k \* y(k), we get T(2^k) = 2^k \* log k 🡪 T(n) = n log log n. This also means that **T(n) = O(n log log n).**

D) T(n) = T(n/4) + T(n/2) + O(n)

To frame this recurrence for the charging argument, we get that T(n) = c \* n + T(n/4) + T(n/2). From this, we can see that ¼ of n items do not enter the subproblems of sizes n/4 and n/2. This means that we should charge 4 \* c units to each of those items to pay for the non-recursive term. Thus, we get that the total work is about T(n) <= 4 \* c \* n where c is the constant in the O(n) of the original recurrence (we are ignoring + O(1) for the base in this case). With this, we can deduce that the recurrence should be **T(n) = O(n).**

1. A) In the book, it is shown that Mn() is invertible because of the fact that Mn()’s columns are orthogonal to each other. However, if there exists a k such that 0 < k < n and that , then the kth column will have all values of 1. This is because the column values will be , , , … , , which will all evaluate to 1. This means that this column will be a column of ones, which is the same as the 0th column of all ones. Since there will be two columns that are essentially scalar multiples of each other, this will mean that Mn() will not have all columns be orthogonal to each other, and therefore Mn() cannot be invertible.

B) with n = 4 and = i would look like the following matrix:

To represent this matrix as a product of two 2-sparse matrices, we can use the following matrices A and B:

The rough work done to find the values of these two matrices is shown below. Multiplying A and B together should return M4(i). So AB = M4(i).

A whiteboard with blue writing

Description automatically generated  
  
  
  
  
  
  
  
Part C on next page

C)

A diagram of a mathematical problem

Description automatically generated with medium confidence

1. A great algorithm that can compute the coefficients would use a divide-and-conquer strategy and utilize the FFT algorithm to multiply coefficients in the combine steps. This should be able to finish in near linear time. The outline of the algorithm is as follows:  
     
   i) Base case: If there is only one complex number, then return the complex number and 1 as a polynomial. Ex. For the base case including complex number r1, then it would return the (x – r1) polynomial.

ii) Divide the complex numbers into two equal parts recursively until you reach a base case with only one complex number.

iii) Recursively compute the two smaller polynomials, A1(x) and A2(x) for example, that correspond to the two halves made from the complex number list done in the divide step.

iv) Use the FFT algorithm to multiply the polynomials A1(x) and A2(x) to obtain the coefficients of the original polynomial, A(x).

The FFT algorithm was shown in the textbook to be able to solve polynomial multiplication problems in O(n log n) time, which is near-linear. Specifically, it can multiply two degree d polynomials in O(d log d) time. In this algorithm, the FFT is used multiple times to calculate polynomials of greater degree. The FFT is called about O(log n) times, with each costing at most O(n/2 log n/2) since the final multiplication would be between two n/2 polynomials. The overall complexity should therefore **T(n) ≤ O(log n) \* O(n/2 log n/2) = O(n log^2 n).**

1. On a paper written by Kedlaya and Umans in 2011, they have the following quote regarding univariate multipoint evaluation: “Consider a degree n univariate polynomial f(X) ∈ Fq[X] (and think of q as being significantly larger than n). If we store f as a list of n coefficients, then to answer a single evaluation query α ∈ Fq (i.e. return the evaluation f(α)), we need to look at all n coefficients, requiring O(n log q) bit operations. On the other hand, a batch of n evaluation queries α1, . . . , αn ∈ Fq c an be answered all at once using **O(n log^2 n)** Fq-operations, using fast algorithms for univariate multipoint evaluation (cf. [vzGG99]).” They cite a 1999 textbook by Gathen and Gerhard for this statement regarding the O(n log^2 n) operations. The citations for both the paper and the textbook can be found below.  
     
   Kedlaya, Kiran S. et al. "Fast Polynomial Factorization and Modular Composition". SIAM Journal on Computing 40. 6(2011): 1767-1802.  
     
   J. von zur Gathen and J. Gerhard. Modern Computer Algebra. Cambridge University Press, 1999.
2. We can use a divide and conquer algorithm similar to the Planar-MIS algorithm discussed in class. The algorithm would be as follows:
3. Assuming that the graph has its vertices sorted from left to right, first pick 9 consecutive vertices from the middle so that there are at most n/2 vertices to the left or right of these vertices. Label these consecutive vertices as S.
4. Since there are only edges between vertices i and j when |i – j| < 10, the vertices on the left of S, which as a group we will call A, will not have any edges to any of the vertices on the right of S, which as a group we will call B. Since this effectively means that S is a separator, any inclusion choice we make from A will be independent from inclusion choices made in group B.
5. Recurse into A and B by again picking 9 consecutive vertices from the middle of A and B respectively similar to step 1.
6. Base case: when the subgraph has 9 or fewer vertices, return MIS, which would be one of the vertices in the subgraph.
7. Conquer each step by finding the largest independent set after doing a union with the independent sets of S, A, and B and return the largest possible combination of the independent sets.

For the time analysis of this algorithm, it would be similar to the Planar-MIS, but since it has a smaller subproblem, some values would be different. We would start off with a recursion of T(n) ≤ O(n) + \* 2T(n/2). There is O(n) work from separating the graph into S, A, and B like that in the Planar-MIS solution. In this case, we know that |S| = 9, so we can deduce the recursion to T(n) ≤ O(n) + \* 2T(n/2) = O(n) + \* T(n/2). From here, we can apply the Master Theorem to deduce the time complexity. We get that a = , b = 2, and d = 1. We find that = 10 > 1, so according to the DPV Master Theorem we would get that **T(n) = ( = ( = (.** With this algorithm, we are able to solve the MIS problem for question 5 in polynomial (O(n^10)) time!