E0-225: Design & Analysis of Algorithms

Manish Kumar (21044)
M.Tech., Quantum Tech.

Assignment- II

September 14, 2023

Problem 1
Balanced Partition in a binary tree with n vertices

Solution. See scanned pages of a notebook at the last.

Problem 2

FFT for setSum in $\mathcal{O}(n \cdot log(n))$ times

Solution. Let the given two sets be-
$S = \{s_1, s_2, ...\}$ and, $T = \{t_1, t_2, ...\}$

• [Observation-I] Important observation from multiplying two polynomials, say $\sum_i x^{a_i}$ and $\sum_j x^{b_j}$.
$\sum_i x^{a_i} \cdot \sum_j x^{b_j} = \sum_{i,j} x^{a_i + b_j} \implies$ Each cross combination of exponents from $\{a_i\}$ *and* $\{b_j\}$ is added.

In our case, we embed sets S and T to the exponent of a polynomial, as mentioned above.
• Hence, calculating set $S + T$ in $\mathbf{O}(n \cdot log(n))$ is equivalent to calculating polynomial multiplication in the $\mathbf{O}(n \cdot log(n))$.
• Also, estimating the coefficient of the term x^{a_u} amounts to finding k_u.

Fast polynomial multiplication using FFT: This algorithm calculates the multiplication of two polynomials of degree n in $\mathbf{O}(n \cdot log(n))$ using FFT.

Step-I: Let the given polynomial be $S(x) = \sum_i x^{s_i}$ and $T(x) = \sum_i x^{t_i}$. Evaluate both functions at $2n$ distinct points using Horner's method that scales as $\mathbf{O}(n)$.

Step-II: This gives us two tuples, $\{(x_i, S(x_i))\}$ *and* $\{(x_i, T(x_i))\}$, each containing $2n$ data points representing the polynomials in point representation.

Step-III: Now, we need to multiply both tuples corresponding element-wise. It yields $\{(x_i, U(x_i))\}$ where $U(x) = S(x) \cdot T(x)$. This point-wise multiplication scale as $\mathbf{O}(n)$.

Step-IV: The goal is to determine $U(x)$ from $\{(x_i, U(x_i))\}$ in $\mathbf{O}(n \cdot log(n))$. This can be implemented by using polynomial interpolation via FFT.

• In class, it was shown how to evaluate a polynomial at n-th roots of unity if the polynomial is given (as its coefficients). Here, we need to estimate the inverse operation. From the given function evaluated points, estimate the coefficients. In terms of DFT matrix relation: $[M_n(w_n)][a_n] = [\hat{a}_n]$.
Our final goal is $[a_n] = M_n(w_n)]^{-1}[[\hat{a}_n]$. We use the inversion result for the DFT matrix. It says- $[M_n(w_n)]^{-1} = \frac{1}{n} M_n(w_n^{-1})]$. Now, the same FFT strategy is implemented to complete the final goal.

Runtime analysis: The main bottleneck of the algorithm is polynomial multiplication. Hence, Asymptotic runtime is $\mathcal{O}(n \cdot log(n))$.
Horner method and pointwise multiplication are insignificant for asymptotic analysis as they scale linearly with input size.

Proof of correctness: Two main assumptions are observation-I and FFT for polynomial interpolation. The correctness of the observation-I is evident from the algebraic expression. The correctness of FFT for polynomial interpolation is due to the *inversion theorem* of FFT.

Problem 3

Finding the median of the disjoint union of two sorted arrays using Divide and Conquer.

Solution. Four key observations/ideas:

Let the sorted array be $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_m\}$

• Point-I: The median of the disjoint union of A and B is the middle-ranked element. Or, half of the elements should appear before it, and the rest after it. (If the list has an even number of terms, the mean of two middle-ranked numbers is taken. We can manage this case by using conditional statements.)

• Point-II: Since our array is sorted, it is guaranteed that if we pick any number (say, p) from any of them, the number before p will be smaller than it.

• Point-III: The important thing to note: We don't need to worry about the sorting of elements coming before and after p. What matters for the median estimation is the correctness of the fact that the elements coming before p are smaller than it. Our algorithm will take care of this fact only.

• Point-IV: Hence, the algorithm will be search-based (in a sorted array) rather than sorting-based. This is to ensure logarithmic scaling with input size.

Algorithmic step

- Step-1: Determine the total length of the disjoint union set $A \cup B$. Determine the index value i for the middle element of the union, accordingly, for even and odd cases. For the odd case, $i = \frac{m+n}{2}$. This is constant time computation (based on the data structure available).

- Step-2: Now, we partition arrays A and B such that if we merge the first half of both, we get total i elements. For example:
  $A_{partition} = \{a_1 \mid a_2, a_3, a_4\}$ and $B_{partition} = \{b_1, b_2, b_3, \mid b_4\}$.

- Step-3: We find the valid partitions using the below conditions. This ensures that after merging, we get the correct median.
  - (I) Take the base case for partition as equal half partitioning. Check if $a_{i-1} < b_{i+1}$ and $a_{i+1} > b_{i-1}$. If true, then the first half of both arrays shall come before the median. Hence, the median = $max\{a_{i-1}, b_{i-1}\}$
  -(II) Otherwise, check which of $a_{i-1} < b_{i+1}$ and $a_{i+1} > b_{i-1}$ is/are voilated. If $a_{i-1} < b_{i+1}$ is violated, then we must push the partition barrier in the left half of array A. At the same time, the barrier will be pushed to the right half in array B. This update of the partition barrier is similar to binary search.
  -This strategy works because the given arrays are sorted. This argument plays a role in the correctness of the algorithm. This is repeated until we get a suitable partition consistent with case-(I) mentioned above. (i.e., the condition for median)
  -(III) If $a_{i+1} > b_{i-1}$ is violated, then we push the partition barrier in the reverse direction to the one mentioned in Case (II). The stopping condition remains the same.

Proof of Correctness: It is based on the correctness of the point-I about the median of the disjoint union of two sets. Also, shifting partition barriers like the binary search algorithm should yield a correct median because it is similar to searching in a sorted array. Hence, its correctness is due to the binary search in a sorted array.

Runtime Analysis: The main computation task is binary partitioning iteratively. This is bound to be completed in the maximum $\mathcal{O}(log(m+n))$ steps.

Problem 4
Optimizing the space of a bookcase in the library.

Solution. Given, $\{b_1, ..., b_n\}$ books with width w_i and height h_i

Strategy: • For using dynamic programming, we need to figure out the overlapping subproblems. Then, recursively solve the larger problem using the smaller subproblems' solution.

• In this case, we start from the base case of adding the first book to the bookcase. This is the bottle to upward building up of the solution.

• There is a constraint on the number of books that can be kept in a row due to the width of the bookcase S. While optimizing the height of the book's arrangement, we use the constraint suitably.

• The subproblem for the case is optimizing the height of the collection when the i-th book is added. The solution at this point is used in subsequent steps when the next book is added. In this sense, solutions to the subproblems are overlapping.

Algorithmic steps:

- Step-I(a): Initialize an array, say H, to store the total height of the collection at each iteration (i.e. addition of a new book).
- Step-I(b): Also, initialize an array, say C, to keep the information of the current configuration of the books on the case.
- Step-II(a): Start with placing book b_1. Add the book, say b_i , either at the current row or the row above it. Two constraints need to be satisfied. First, adding b_i in the current row shall be consistent with bookcase width S. Also, continue with that book configuration, which yields the minimum height at this iteration. This requires running a loop over the current strength of the books.
- Update arrays H and C accordingly.
- Step-II(b): Keep running iterations with the above prescription till all books are added. Finally, array C contains the optimal configuration, while the last element of H contains the minimum possible height for this problem.

Runtime analysis: Run time scales as $\mathcal{O}(n^2)$. Since an outer loop over all books is required. It is due to the addition of each book one by one.
Within each iteration, an optimal configuration must be obtained that requires running over the current strength of the books at that iteration.

Argument for optimality: There could be more than one optimal solution to the problem. This algorithm guarantees to find one of them. This is based on the execution of step-II(a). At each iteration, an optimal configuration is considered.

Problem 5

Finding the largest monochromatic square in the greed

- Part-1: Design an algorithm that solves the problem and employs dynamic programming
- Part-2: Include its complete recurrence relation

Solution. Strategy: Employing Dynamic Programming in this context requires-

- Point-I: Identifying the overlapping Subproblem and recursively constructing the solution of the whole problem in terms of the solution of the subproblem.
- Point-II: Associate *white* $\rightarrow$ 1 and *Grey* $\rightarrow$ 0 to the given coloured box input. This converts it into $A_{n\times n}$ matrix.
- Point-III: A trivial one-sized square is each of the squares containing 1.
- Point-IV: The next monochromatic square is 2×2 sized square (four adjacent squares containing 1). Or, the presence of the matrix element a_{ij} such that $a_{i,j} = a_{i,j-1} = a_{i-1,j} = a_{i-1,j-1} = 1$ implies such monochromatic.
- Point-V: **Overlapping subproblems**: The presence of $i \times i$ sized monochromatic matrix depends on the existence of $(i-1) \times (i-1)$ sized monochrome matrix.
- Point VI: We use the above facts in the recursive building of the memoization table for the problem.

Algorithmic steps:

- Step-I(a): Initialize a matrix, say M, of the same size as A. This is to recursively build a memo table useful to get the largest monochromatic square.
- Step-I(b): We traverse in the matrix $A_{n\times n}$ to search for patterns and then populate the matrix M. By pattern means monochromatic squares of different sizes.
- Step-II(a): M will be built from left to right and top to bottom style. Hence, the first row and column of M are the same as that of A. This acts like a base case.
- For the rest of its entries, we adhere to Point-IV mentioned here. It implies-
  - If $a_{i,j} = 0$, then fill $m_{i,j} = 0$, otherwise
  - $m_{i,j} = 1 + min(a_{i,j-1}, a_{i-1,j-1}, a_{i-1,j})$
- Step-II(b): At the end of each iteration, M counts and stores any (possible) larger monochromatic square by building upon the result obtained from the last iteration.
- Step-III: The above step output matrix M. The largest entry in the matrix corresponds to the size of the largest monochromatic square possible.

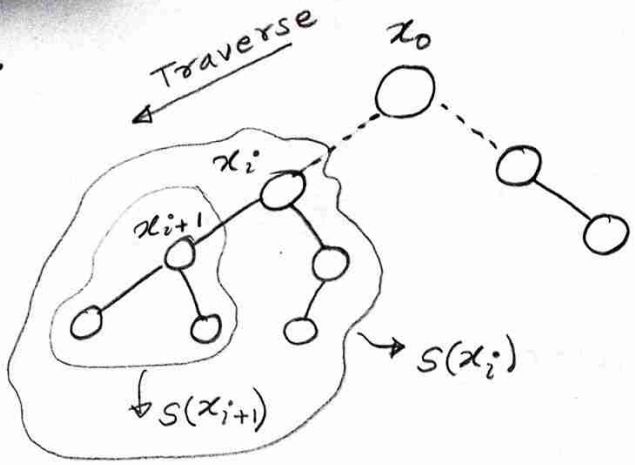
Runtime analysis: Run time scales as $\mathcal{O}(n^2)$. Since there is a single-time matrix ($M_{n\times n}$) traversal. Within each iteration, a constant number of comparisons amounts to a constant overhead of time.

Correctness Argument: For correctness, we argue how the algorithm's way of counting monochrome squares coincides with the definition of the largest monochrome square.

By definition, the largest monochrome square is the largest isolated bunch of 1s appearing together in the matrix.

We inspect the working of expression $m_{i,j} = 1 + min(a_{i,j-1}, a_{i-1,j-1}, a_{i-1,j})$ is exactly like this. We notice $m_{i,j}$ doesn't change its value if either of $a_{i,j-1}, a_{i-1,j-1}, a_{i-1,j}$ is zero. It means it has encountered a grey box in the region of concern. It keeps growing (sequentially) if it keeps encountering white-only boxes.

1.



$$S(x_i) \leq 2\,S(x_{i+1}) + 1$$

(a) we take above binary tree to illustrate the claim mentioned in the question.

→ The tree is unbalanched because the below arguments will also holds for balanced tree without any loss of generality.

→ We start from vertex x_0 and travel along the larger child. [We break any tie arbitarly.]

→ At any vertex, say x_i, we count the subtree associated to it. The size of subtrees at x_i and x_{i+1} is related by $S(x_i) \leq 2\,S(x_{i+1}) + 1$. Equality holds for balanced tree.

→ This observation is sufficient to derive the inequality mention in the question.

→ We take the case if $S(x_{i+1}) \geq \frac{n}{4}$. ⟶ Bound-I

$\Rightarrow$ $S(x_i) \leq \frac{n}{2} + 1$

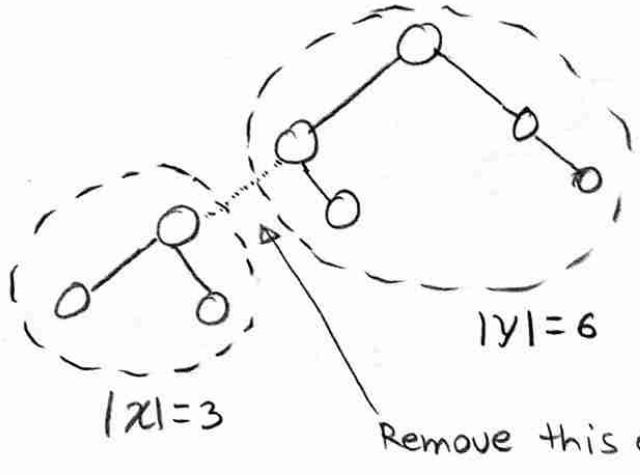
for $n \geq 4$, $\frac{n}{2} + 1 \leq \frac{3n}{4}$ ⟶ Bound-II

Hence, if we partition the tree by removing the edge between x_i and x_{i+1} we get two partition $\mathcal{X}$ and $\mathcal{Y}$, such that

$$\frac{n}{4} \leq |\mathcal{X}| \leq |\mathcal{Y}| \leq \frac{3n}{4}$$

(b)

we can use the construction of part-(a)



$n = 8$

$\Rightarrow \frac{3n}{4} = 6$

(c)

→ A tree with n-vertices has $(n-1)$ edges.

→ We also utilize the fact that for any vertex v_i, size of subtree of x_i is atmost three times the size of x_i.

$$S(x_{i+1}) \leq \frac{S(x_i)}{3}$$

→ It implies by partitioning similar to part (a), we get $\frac{S(x_i)}{3}$ sized tree.

→ We can iteratively apply above method to arrive at any desired value "k" $\in [0, n-1]$. This would be similar to search.

→