Algorithm Design and Analysis

Assignment 1

Deadline: March 24, 2024

1. (25 points) Asymptotic notations.

   (a) In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). Justify your answer.

      1. $f(n) = n^{1/2}$ and $g(n) = 5^{\log_2 n}$
      2. $f(n) = 100n + \log n$ and $g(n) = n + (\log n)^2$
      3. $f(n) = (\log n)^{\log n}$ and $g(n) = n/\log n$
      4. $f(n) = (\log n)^{\log n}$ and $g(n) = 2^{(\log_2 n)^2}$
      5. $f(n) = \sum_{i=1}^{n} i^k$ and $g(n) = n^{k+1}$

   (b) Let $f(n) = \big(2 \cdot \lceil \frac{n}{2} \rceil\big)!$ and $g(n) = \big(2 \cdot \lfloor \frac{n}{2} \rfloor + 1\big)!$. Prove that neither $f = O(g)$ nor $f = \Omega(g)$ is true.

   Solutions:

   (a)
      1. $f = O(g)$, $g(n) = 5^{\log_2 n} = n^{\log_2 5} > n^2 >> n^{1/2} = f(n)$.
      2. $f = \Theta(g)$. Let $c_0 = 1, c_1 = 101, c_0 \cdot g(n) = n + (\log n)^2 \leq f(n) \leq 101n + 101(\log n)^2 = c_1 \cdot g(n)$, when n is large enough.
      3. $f = \Omega(g)$, $f(n) = (\log n)^{\log n} = n^{\log \log n} > n >> n/\log n = g(n)$. When $n > 10^{10}$.
      4. $f = O(g)$, $g(n) = 2^{(\log_2 n)^2} = n^{\log_2 n} >> n^{\log \log n} = f(n)$.
      5. $f = \Omega(g)$, We have:
         $n^{k+1} - (n-1)^{k+1} = \sum_{i=0}^{k} (-1)^{k-i} C_i^{k+1} n^i$
         $(n-1)^{k+1} - (n-2)^{k+1} = \sum_{i=0}^{k} (-1)^{k-i} C_i^{k+1} (n-1)^i$
         ...
         $1^{k+1} - 0^{k+1} = \sum_{i=0}^{k} (-1)^{k-i} C_i^{k+1} 1^i$
         plus them, we have:
         $n^{k+1} = \sum_{i=1}^{k} (-1)^{k-i} C_i^{k+1} S_i(n)$
         $S_k(n) = \sum_{i=1}^{n} i^k$
         then $f(n) = S_k(n) = (n^{k+1})/(k+1) - 1/(k+1) \sum_{i=0}^{k} (-1)^{k-i} C_i^{k+1} S_i(n)$, which has the same order of magnitude as $g(n)$.

   (b) $f \neq O(g)$: If there exists c that makes $0 \leq f(n) \leq cg(n)$, let n be an odd number, then $f(n) = (n+1)!, g(n) = n!$, let $n > c, cg(n) = cn! < (n+1)n! = f(n)$, which is not consistent with the assumption.
   $f \neq \Omega(g)$: If there exists c that makes $0 \leq cg(n) \leq f(n)$, let n be an even number, then $f(n) = n!, g(n) = (n+1)!$, let $n > c, cg(n) = c(n+1)! > n! = f(n)$, which is not consistent with the assumption.

2. (25 points) Prove the following generalization of the master theorem. Given constants $a \geq 1, b > 1, d \geq 0$, and $w \geq 0$, if $T(n) = 1$ for $n < b$ and $T(n) = aT(n/b) + n^d \log^w n$, we have

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log^{w+1} n) & \text{if } a = b^d \end{cases}.$$

Solutions: Expand the recursion tree based on the recurrence $T(n) = aT(n/b)+n^d \log^w n$ and observe its structure, each node has a cost of $n^d \log^w n$, and each level has a subproblems, each with a size of $1/b$ of the original problem. Thus,

- At the first level, there is 1 node with a cost of $n^d \log^w n$.
- At the second level, there are a nodes, each with a cost of $(n/b)^d \log^w(n/b)$.
- At the third level, there are a^2 nodes, each with a cost of $((n/b^2)^d \log^w(n/b^2))$.
- Continuing this pattern, the kth level has a^k nodes, each with a cost of $((n/b^k)^d \log^w(n/b^k))$.

We see that at the kth level, each node has a cost of $((n/b^k)^d \log^w(n/b^k))$. To find the depth of the recursion tree, we need to find k such that $n/b^k < b$. Solving this inequality gives $k = \log_b n$.

Consider the growth of the recursion tree under different scenarios:

1. $a < b^d$

In this case, each node's cost $((n/b^k)^d \log^w(n/b^k))$ is smaller than the cost of the nodes at the previous level $(n^d \log^w n)$. Therefore, the growth of the tree is controlled by the leaf nodes. Since the depth of the recursion tree is $\log_b n$, the number of leaf nodes is $a^{\log_b n}$. Thus, the total cost is $O(n^d \log^w n)$.

2. $a > b^d$

In this case, each node's cost $((n/b^k)^d \log^w(n/b^k))$ is greater than the cost of the nodes at the previous level $(n^d \log^w n)$. Therefore, the growth of the tree is controlled by the root node. According to the master theorem, the total cost of the tree is $O(n^{\log_b a})$.

3. $a = b^d$

In this case, each node's cost $((n/b^k)^d \log^w(n/b^k))$ is equal to the cost of the nodes at the previous level $(n^d \log^w n)$. Therefore, the cost remains the same at each level. Since the depth of the recursion tree is $\log_b n$, the total cost is $O(n^d \log^{w+1} n)$.

3. (25 points) Given an array $A[1 \cdots n]$ of integers, a pair of indices (i, j) is an *inversion* if $i < j$ and $A[i] > A[j]$. Design an algorithm that counts the number of inversions in $O(n \log n)$ time. **Hint:** Suppose the first half of the array $A[1 \cdots (n/2)]$ and the second half of the array $A[(n/2 + 1) \cdots n]$ (say, n is an even number) are sorted by ascending order, can you count the number of inversions in $O(n)$ time?

   Solution: For two arrays sorted by ascending order, the time complexity of merging them is $O(n)$. And during the merging process, if we found that the right element is smaller the left one, it is an *inversion*. Therefore, when sorting, we define a variable to store the number of *inversion*. If an *inversion* is found, count is increased to the number of remaining elements in the left array. Because the array on the left is arranged in ascending order, if an *inversion* is found, the remaining numbers in the left array will form an *inversion* for this element in the right array.
   Therefore, we can solve this problem using a method based on merge sorting. In merge sorting, we only need to count the number of *inversion* during each sorting steps, and the final value of the count variable is the answer. The time complexity of this method is the same as that of merge sorting, which is $O(n \log n)$.

4. (25 points) Let A be a square matrix. This question discusses the computation of A^2.

   (a) Show that five multiplications are sufficient to compute the square of a 2×2 matrix.

   (b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

   - Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$.

   (c) In fact, squaring matrices is no easier than matrix multiplication. In this part, you will show that if $n \times n$ matrices can be squared in time $S(n) = O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

   1. Given two $n \times n$ matrices A and B, show that the matrix $AB + BA$ can be computed in time $3S(n) + O(n^2)$.
   2. Given two $n \times n$ matrices X and Y, define the $2n \times 2n$ matrices A and B as follows:
   $$A = \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix} \qquad \text{and} \qquad B = \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix}.$$
   What is $AB + BA$, in terms of X and Y?
   3. Using 1 and 2, argue that the product XY can be computed in time $3S(2n) + O(n^2)$. Conclude that matrix multiplication takes time $O(n^c)$.

Solutions:

(a) To compute the square of a 2×2 matrix, we need to calculate $a^2 + bc, ab + bd, ac + cd, bc + d^2$, which needs seven multiplications.
By Strassen's magical idea, we can calculate:
$P_1 = a(a + b)$
$P_2 = b(-a + c)$
$P_3 = b(c + d)$
$P_4 = d(-b + d)$
$P_5 = c(a + d)$
And $A^2 = \begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix} = \begin{bmatrix} P_1 + P_2 & -P_2 + P_3 \\ P_5 & P_3 + P_4 \end{bmatrix}$,which needs only five multiplications.

(b) We have: $\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2 + BC & AB + BD \\ CA + DC & CB + D^2 \end{bmatrix} \neq \begin{bmatrix} A^2 + BC & B(A + D) \\ C(A + D) & CB + D^2 \end{bmatrix}$
This indicates that the subproblem of squared non second-order matrices cannot be solved using the method in (a), which needs only 5 multiplications. There are

still 7subproblems, beacuse matrix multiplication has no commutative law, only associative law.

(c) 1. $(A+B)^2 - A^2 - B^2 = A^2 + AB + BA + B^2 - A^2 - B^2 = AB + BA$, which needs 3 times of squaring matrices and several times of matrix addition, which can be computed in time $3S(n) + O(n^2)$.

2. $AB + BA = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$.

3. According to 1, calculate $AB + BA$ needs $3S(n) + O(n^2)$, According to 2, $AB + BA = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$, when $A = \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix}$, then the product XY can be calculated using the above method, which needs time $3S(2n) + O(n^2) = O(n^c)$.

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

I spent about three hours completing this assignment, and then another hour checking and revising the answers with my classmates.

Although I have learned some algorithmic knowledge and have done some exercises, this assignment is still not easy for me. More than half of the questions need to be completed by consulting textbooks and materials, or by thinking together with friends. I think the difficulty score is at least 4.

My collaborator is Zou Ruoqin. I referred to the content of Chapters 3 and 4 of "Introduction to Algorithms".