ps2

Ji Qiankun

June 12, 2024

1

(a) Use the following algorithm:

1. Compute ac.
2. Compute bd.
3. Compute (a+b)(c+d).

Then, the real part of the result is ac - bd and the imaginary part is (a+b)(c+d) - ac - bd.

Here is a Python implementation of this algorithm:

```
def multiply_complex(a, b, c, d): ac = a * c
bd = b * d
ab_cd = (a + b) * (c + d)
real = ac - bd
imag = ab_cd - ac - bd
return real, imag
```

This function takes four arguments: a, b, c, and d, which represent the real and imaginary parts of two complex numbers a + bi and c + di. It returns two values: the real and imaginary parts of the product of the two complex numbers.

(b) Use the following approach:

1. Create two new matrices C and D, each of size n × n, such that:
- The elements of C are given by C[i][j] = A[i][j] + B[i][j] for all i, j.
- The elements of D are given by D[i][j] = A[i][j] - B[i][j] for all i, j.
2. Square the matrices C and D using the $\Theta(n^{\alpha})$time algorithm for squaring matrices to get matrices E
3. The product of A and B can then be computed as (E + F) / 2.

This approach works because:

$(A + B)^2 = A^2 + 2AB + B^2$

$(A - B)^2 = A^2 - 2AB + B^2$

By adding these two equations, we can get:

$(A + B)^2 + (A - B)^2 = 2A^2 + 2B^2$

So,$AB = ((A + B)^2 - (A - B)^2)/4$

2

(a) First try to prove the recurrence $T(n) = 2T(n-1)+1$with the assumption $T(n) \leq c \times 2^n$:

Assume $T(n) \leq c \times 2^n$ for all $n < k$. We want to show that $T(k) \leq c \times 2^k$.

Substituting T(k - 1) from the assumption into the recurrence, we get:

$T(k) = 2T(k-1) + 1 \leq 2(c \times 2^{k-1}) + 1 = c \times 2^k + 1$

This is not less than or equal to $c \times 2^k$, so the assumption does not hold.

Secondly, subtract a lower-order term to make the substitution proof work. We can subtract n from both sides of the inequality to get $T(n)-n \leq c \times 2^n - n$.

Assume $T(n) - n \leq c \times 2^n - n$ for all $n < k$. We want to show that $T(k) - k \leq c \times 2^k - k$.

Substituting T(k - 1) from the assumption into the recurrence, we get:

$T(k) - k = 2T(k-1) + 1 - k \leq 2(c \times 2^{k-1} - (k-1)) + 1 - k = c \times 2^k - 2k + 1$

This is less than or equal to $c \times 2^k - k$ for sufficiently large c and k, so the assumption holds with the subtraction of the lower-order term.

(b) To prove that $T(n) = \Omega(n)$ using the substitution method, we need to show that there exists a constant $c > 0$ and the value n0 such that $T(n) \geq cn$ for all $n \geq n0$.

For $n \geq n0$, the recurrence is $T(n) = T(n/4) + T(3n/4)$.

First we assume $n/4 > n0$,then $T(n) = T(n/4) + T(3n/4) \geq cn/4 + 3cn/4 = cn$.

Second we assume $n/4 < n0$ but $3n/4 > n0$,then$T(n) = T(n/4)+T(3n/4) = 1 + 3cn/4 \geq cn$,so $1 + 3cn0/4 \geq cn0$,we get $c \leq 4/n0$.

Last station,$3n/4 \leq n0$,then $T(n) = T(n/4) + T(3n/4) = 2 \geq cn$,so $2 \geq cn0$,we get $c \leq 2/n0$.

Substituting T(n/4) and T(3n/4) from the assumption into the recurrence, we get:

$T(n) = T(n/4) + T(3n/4) \geq c(n/4) + c(3n/4) = cn$ (among them $c \leq 2/n0$)

Therefore, by induction $T(n) = \Omega(n)$ for all $n \geq n0$.

(c) To solve the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + \Omega(n)$, the recursion tree is as fellow.

The root of the tree represents the original problem T(n). This node has two children, $T(\alpha n)$ and $T((1-\alpha)n)$, representing the two recursive calls. The cost at the root is $\Theta(n)$.

At the second level, $T(\alpha n)$ branches into $T(\alpha^{2n})$ and $T(\alpha(1-\alpha)n)$, and $T((1-\alpha)n)$ branches into $T(\alpha(1-\alpha)n)$ and $T((1-\alpha)^{2n})$. The cost at this level is also $\Theta(n)$.

This pattern continues for each level of the tree. At each level, the total cost is $\Theta(n)$, and the number of levels is proportional to $log(n)$, because at each level we are reducing the problem size by a constant factor.

Therefore, the total cost of all levels of the tree is $\Theta(n)log(n) = \Theta(nlogn)$, which justifies that $T(n) = O(nlogn)$.

To show that $T(n) = \Omega(nlogn)$, we observe that the cost at each level of the tree is at least a constant times n, and there are at least log(n) levels in the tree. Therefore, the total cost of all levels of the tree is at least a constant times n log n, which justifies that $T(n) = \Omega(nlogn)$.

So, the solution to the recurrence is $T(n) = \Theta(nlogn)$.

3

(a) Define $m = lgn$, which implies $n = 2^m$. Then define $S(m) = T(2^m)$. The original recurrence $T(n) = 2T(\sqrt{n}) + \Theta(lgn)$ can be rewritten in terms of m and S(m) as:

$S(m) = 2S(m/2) + \Theta(m)$

(b) Solve the Recurrence for S(m) and T(n)

The recurrence for S(m) is a standard form of divide-and-conquer recurrence, which can be solved using the Master Theorem. It falls into Case 2 of the Master Theorem, and the solution is $(m) = \Theta(mlogm)$.

Substituting back for $m = lgn$ and $S(m) = T(2^m)$, we get $T(n) = \Theta((lgn)log(lgn))$.

(c) Solve the Recurrences by Changing Variables:

1] $T(n) = 2T(\sqrt{n}) + \Theta(1)$

Define $m = lgn$ and $S(m) = T(2^m)$. The recurrence can be rewritten as $S(m) = 2S(m/2) + \Theta(1)$. This falls into Case 1 of the Master Theorem, and the solution is $S(m) = \Theta(m)$.

Substituting back and we get $T(n) = \Theta(lgn)$.

[2] $T(n) = 3T(n^{1/3}) + \Theta(n)$

Define $m = \log_3 n$ and $S(m) = T(3^m)$. The recurrence can be rewritten as $S(m) = 3S(m/3) + \Theta(3^m)$. This falls into Case 3 of the Master Theorem, and the solution is $S(m) = \Theta(3^m)$.

Substituting back and we get $T(n) = \Theta(n)$.

4

The algorithm to convert an infix expression to postfix uses a stack to hold operators and follows these steps:

1. Scan the infix expression from left to right.
2. If the scanned character is a digit, output it.
3. If the scanned character is an operator, pop operators from the stack and output them until you find an operator with less precedence or the stack is empty. Then push the scanned operator onto the stack.
4. If the scanned character is '!', which is a unary operator, output it immediately.
5. If the infix expression is completely scanned and there are still operators in the stack, pop and output them.

Here is the pseudocode for the algorithm:

```
function infixToPostfix(expression)
    stack = new Stack()
    postfix = ""
    for each character in expression do
        if character is an operand then
            postfix+ = character
        else if character is an operator then
            while not stack.isEmpty() and hasHigherPrecedence(stack.peek(),
character) do
                postfix+ = stack.pop()
            end while
            stack.push(character)
        end if
    end for
    while not stack.isEmpty() do
        postfix+ = stack.pop()
    end while
    return postfix
end function
function hasHigherPrecedence(op1, op2)
    if op1 is "!" and op2 is not "!" then
        return true
    else if op1 is "×" and op2 is "+" then
        return true
    else
        return false
    end if
end function
```

Correctness: We prove it by induction.

Induction hypothesis: the algorithm will turn infix expression to sufix expression correctly when the length of input is at most n.

When $n = 1$, the input is a single digit, the output is correct.

When $n > 1$, denote the last operator with the smallest priority as A[x]. (When there are three operator !x+, that means A[x] is the last + in the expression). Then A[x] should be in the last place in postfix expression. That is true in our algorithm, since when we add A[x] to the stack, it will pop all elements in stack out (because it has the smallest priority), so it will fall into the bottom of the stack. And nothing will pop it out until the fifth line of our algorithm, which will add A[x] to the last place in the postfix expression. Now consider A[0...x 1] and A[x + 1...n] (might be empty string). They both have length smaller than n. Before A[x] is push into the stack, the procedure can be seen as running our algorithm in A[0...x 1], since A[x] will pop all elements out. Thus, according to induction hypothesis, A[0...x1] will be turned to postfix

expression correctly. $A[x+1...n]$ can also be seen as running our algorithm independently, since all elements in A[0....x 1] is not in the stack, and A[x] is in the bottom of the stack while no one popping it out. Thus, the final string is postfix(A[0...x 1]) + postfix(A[x + 1...n]) + A[x], which is correct.

Complexity: Each operator will be popped and pushed at most twice, and each digit will be looped at most once. The complexity is O(n).

5

The original FINDMAXIMUMSUBARRAY algorithm in CLRS uses a divide-and-conquer approach, but it spends O(n) time in the "combine" step to find the maximum crossing subarray, leading to an overall time complexity of O(n log n).

To modify it to run in O(n) time, we need to compute some additional information during the "conquer" step so that the "combine" step can be done in O(1) time. Specifically, for each subarray A[low, ..., high], we compute four values:

1. The sum of the entire subarray.
2. The maximum subarray sum that includes the first element.
3. The maximum subarray sum that includes the last element.
4. The maximum subarray sum anywhere in the subarray.

Here is the pseudocode for the modified algorithm:

procedure FINDMAXIMUMSUBARRAY(A, low, high)
  if low == high **then**
    return (A[low], A[low], A[low], A[low])
  else
    mid ← (low + high) / 2
    (leftSum, leftPrefix, leftSuffix, leftMax) ← FINDMAXIMUMSUBARRAY(A, low, mid)
    (rightSum, rightPrefix, rightSuffix, rightMax) ← FINDMAXIMUMSUBARRAY(A, mid + 1, high)
    totalSum ← leftSum + rightSum
    maxPrefix ← max(leftPrefix, leftSum + rightPrefix)
    maxSuffix ← max(rightSuffix, rightSum + leftSuffix)
    maxSum ← max(leftMax, rightMax, leftSuffix + rightPrefix)
    return (totalSum, maxPrefix, maxSuffix, maxSum)
  end if
end procedure

correctness: In the pseudocode, findMaximumSubarray(A, low, high) returns a tuple of four values for the subarray A[low, ..., high]: the sum of the entire subarray, the maximum subarray sum that includes the first element, the maximum subarray sum that includes the last element, and the maximum subarray sum anywhere in the subarray.So the output values are correct during the process.

The time complexity of this algorithm is O(n), where n is the size of the input array. This is because each recursive call processes a subarray of half the size and the "combine" step takes constant time.

6

(a) Algorithm: Suppose n friends are numbered from 1 to n, and we have an operation query(a, b) for $1 \leq a = b \leq n$, which returns a pair (I1, I2) where $I1, I2 \in C, W$ (C means citizen and W means werewolf), I1 is the identity of a told by b and I2 is the identity of b told by a.

Define function Judge(a, A) where A is an array of citizens as follows:

$cnt \leftarrow 0$

For $i = 1$ to A.length where A[i] = a do:

$(I1, I2) \leftarrow query(a, A[i])$, if $I1 = C$ then $cnt \leftarrow cnt + 1$.

If $cnt \geq (A.length - 1)/2$then return C. Otherwise return W.

Correctness:

In each round, we ask everyone about the identity of the target player. Since citizens always tell the truth, if everyone says the target player is a citizen, then the target player is a citizen. If one says that the target player is a werewolf, then the target player is a werewolf, since citizens tell the truth and werewolves may lie. Therefore, no matter how the werewolf answers our questions, we are always able to determine the identity of the target player. This proves that the algorithm is correct.

(b) A Divide-and-Conquer algorithm:

1. Divide the group into two subgroups.

2. For each subgroup, pick any two friends and ask each of them about the other's identity.

3. Follow the same rules as in the linear time algorithm to find a citizen in each subgroup.

4. Repeat this process recursively until you find a citizen.

This is implemented as follows:

Define function FindCitizen(l, r) as following:

1 If $l == r$, return A[l].

2 Set $m = [(l + r)/2]$. Let a = FindCitizen(l, m) and b = FindCitizen(m + 1, r).

3 If Judge(a, A[l...r]) = C, return a. If Judge(b, A[l..r]) = C, return b.

If $l \leq r$ and there are more citizen than werewolf in A[l...r], then FindCitizen(l, r) will return a real citizen.

Proof of correctness(proved by induction):

Basic steps: If a group has only one person, then this person is a citizen, since there are always more citizens than werewolves.

Induction step: Suppose that our algorithm correctly finds a citizen for all groups smaller than n. Now consider a group of n individuals. I split this group into two sub-groups, each with less than n people. By my induction hypothesis,

I know that our algorithm can find a citizen in each subgroup. I can then determine that at least one person is a citizen by asking about the identity of the two citizens. Since citizens always tell the truth, if one person says another person is a werewolf, then that person must be a werewolf. Thus, I can be sure that at least one person is a citizen.

Therefore, no matter how the werewolf answers my questions, I can always find a citizen. This proves that my algorithm is correct.

Complexity: Suppose FindCitizen(l, r) use time T(n) where $n = r - l + 1$, then $T(n) \leq 2T([n/2]) + O(n)$. According to the master method, T(n) = O(n log n).

(c) One possible algorithm:

1. Start by randomly selecting two people and ask each of them if the other person is a citizen or a werewolf.

2. If both individuals claim that the other person is a citizen, then one of them must be telling the truth. We can select either of them as a potential citizen and move on to the next step.

3. If one person claims that the other person is a citizen and the other person claims that they are a werewolf, then the person claiming to be a citizen is definitely a citizen. We can select them as the citizen and stop the algorithm.

4. If both individuals claim that the other person is a werewolf, then both of them are either werewolves or one of them is a citizen. In either case, we can discard both of them and move on to the next pair of individuals.

5. Repeat steps 1-4 until a citizen is found or until all pairs of individuals have been exhausted.

The algorithm works because if there is at least one citizen among the group, then there will always be a pair of individuals where one claims the other is a citizen. By discarding pairs where both claim the other is a werewolf, we can eliminate werewolves and focus on potential citizens.

Correctness consider the possible scenarios:

1. If there is no citizen in the group, then all individuals will claim that the other person is a werewolf. In this case, the algorithm will exhaust all pairs and not find a citizen, which is the correct result.

2. If there is at least one citizen in the group, then there will always be a pair of individuals where one claims the other is a citizen. The algorithm will eventually encounter this pair and select the citizen as the result.

Therefore, the algorithm is correct regardless of the behavior of the werewolves and it runs in O(n) queries.