Solution for Problem Set 3

201300035 方盛俊

Problem 1

(a)

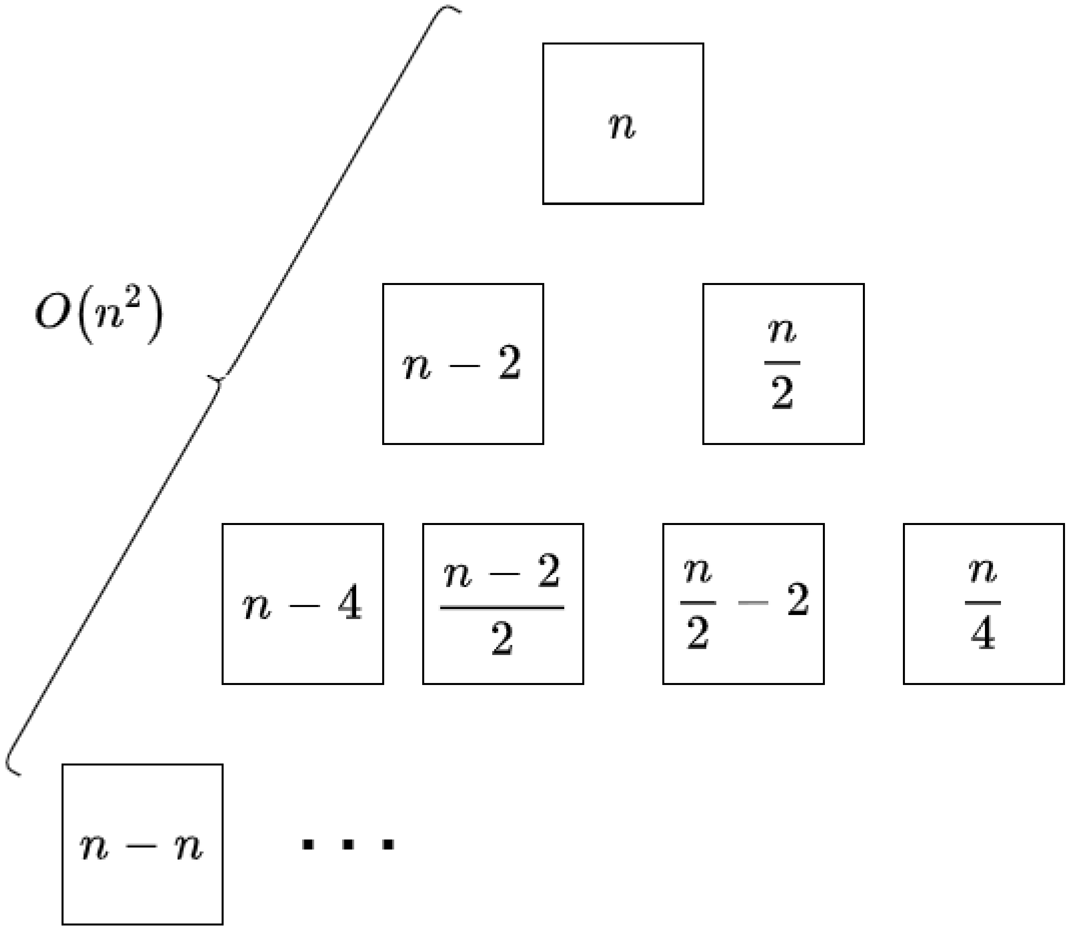
Use the substitution method.

We guess that $T(n) \leqslant dn \ln n - d'n, n \geqslant 2$ and use substitution-method.

- Induction Basis: $T(2) = c_1 \leqslant d \cdot 2 \cdot \lg 2 - d' \cdot 2$, so long as $2d - 2d' \geqslant c_1$
- Inductive Step: $T(n) = 2 \cdot T(\frac{n}{2}) + n \leqslant 2(d(\frac{n}{2}) \lg \frac{n}{2} - d'(\frac{n}{2})) + n = dn(\lg n - \lg 2) - d'n + n = dn \lg n - d'n + (1-d)n \leqslant dn \lg n - d'n$, so long as $d \leqslant 1$

So $T(n) = O(n \lg n)$

(b)



Sorry, I don't know how to solve the problem. I had tried my best but got nothing out.

Problem 2

(a)

$$\because T(n) = T(\alpha) + T(n-\alpha) + cn = \cdots = \frac{n}{\alpha} \cdot T(\alpha) + c \cdot \frac{(n+0) \cdot \frac{n}{\alpha}}{2} = \frac{c}{2\alpha} \cdot n^2 + \frac{T(\alpha)}{\alpha} \cdot n$$

$$\therefore T(n) \in \Theta(n^2)$$

(b)

$$\therefore T(n) \in \Theta(n \log n)$$

Problem 3

Overview:

Because $xy = (2^{\frac{n}{2}} \cdot x_L + x_R)(2^{\frac{n}{2}} \cdot y_L + y_R) = 2^n \cdot x_L y_L + 2^{\frac{n}{2}} \cdot ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R$, so the time is $T(n) = 3T(\frac{n}{2}) + c_1 n$. Let the $y = x$ and the time still is $T(n) = 3T(\frac{n}{2}) + c_1 n$.

Algorithm:

Algorithm 1 Square

---

  function $\textsc{FastMulti}(x, y)$
    if x **and** y are both of 1 bit **then**
      return $x \times y$
    end if
    $x_L, x_R =$ most, least significant $|x|/2$ bits of x
    $y_L, y_R =$ most, least significant $|y|/2$ bits of y
    $z_1 = \text{FastMulti}(x_L, y_L)$
    $z_2 = \text{FastMulti}(x_R, y_R)$
    $z_3 = \text{FastMulti}(x_L + x_R, y_L y_R)$
    return $2^n \times z_1 + 2^{\frac{n}{2}} \times (z_3 - z_1 - z_2) + z_2$
  end function
  function $\textsc{Square}(x)$
    return $\text{FastMulti}(x, x)$
  end function

Time Complexity:

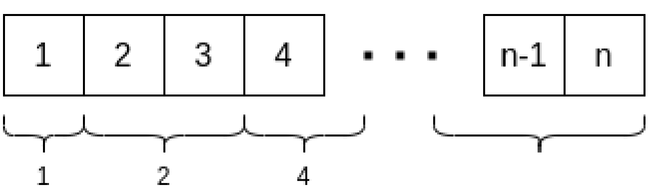
FastMulti / Square: $T(n) = 3T(\frac{n}{2}) + c_1 n$

We guess that $T(n) \leqslant dn^{\lg 3} - d'n$ and use substitution-method.

- Induction Basis: $T(1) = c_2 \leqslant d \cdot 1^{\lg 3} - d' \cdot 1$, so long as $d - d' \geqslant c_2$
- Inductive Step: $T(n) = 3 \cdot T(\frac{n}{2}) + c_1 \cdot n \leqslant 3(d(\frac{n}{2})^{\lg 3} - d'(\frac{n}{2})) + c_1 n = dn^{\lg 3} - (\frac{3}{2}d' - c_2)n \leqslant dn^{\lg 3} - d'n$, so long as $\frac{1}{2}d' \geqslant c_1$

So $T(n) = O(n^{\lg 3})$

Problem 4



Overview:

Compare x with the i-th element and the i increase exponentially by step $i = 2i$. If x is less than i-th element, then search it by binary search algorithm, which the max time is $O(\lg n)$, else continue.

Algorithm:

Let the array be A.

Algorithm 2 Search

```
function SEARCH(x)
    i = 1
    while true do
        if x == A[i] then
            return i
        end if
        if A[i/2] == ∞ then
            return 0
        end if
        if x < A[i] then
            return BinarySearch(i/2, i)
        end if
        i = 2i
    end while
end function
```

Time Complexity:

In the worst case, the x is the last element of array. So the time is $T(n) = c_1 \lg n + T_{BinarySearch}(n) = c_1 \lg n + O(\lg \frac{n}{2}) = O(\lg n)$.

Problem 5

Overview:

We need to find delegate belong to the majority party so that we can introduce him with other delegates to find if they are belong to the same party. In order to find the majority delegate, who smiles to over half persons, we can divide the delegates into two group, and get the majority delegates in the two group, and find the final delegate.

Algorithm:

Let G be all delegates.

Algorithm 3 Search

```
function CountSmileDelegates(x, group)
    count = 0
    for delegate in group do
        if PairwiseMeeting(x, delegate) == "smile" then
            count = count + 1
        end if
    end for
    return count
end function
function GetMajorityDelegate(group)
    n = length of group
    if n == 1 then
        return (group[1], 1)
    end if
    (left, leftCount) = GetMajorityDelegate(group[1...n/2])
    (right, rightCount) = GetMajorityDelegate(group[n/2+1...n])
    leftTotalCount = leftCount + CountSmileDelegates(left, group[n/2+1...n])
    rightTotalCount = rightCount + CountSmileDelegates(right, group[1...n/2])
    if leftTotalCount >= rightTotalCount then
        return (left, leftTotalCount)
    else
        return (right, rightTotalCount)
    end if
end function
function GetMajorityPartyDelegates()
    x = GetMajorityDelegate(G)
    l = a new list of delegate
    for delegate in G do
        if PairwiseMeeting(x, delegate) == "smile" then
            l.add(delegate)
        end if
    end for
    return l
end function
```

Correctness:

Firstly, we prove that it is impossible that the final delegate is not belong to the majority party. Because more than half of the delegates belong to the same political party, there is one group at least, where more than half of the delegates belong to the same political party, if we divide big group into two small groups. For example, there are $n+1$ delegates belong to the majority party in $2n$ group, we divide them into two groups equally, $\frac{n}{2}+1$ majority delegates in n delegates group, and more than half of n are majority delegates.

So we can get the majority delegate finally.

Time Complexity:

Based on the number of "pairwise meetings".

- **CountSmileDelegates:** $\Theta(n)$
- **GetMajorityDelegate:** $T_1(n) = 2T_1(\frac{n}{2}) + n = n \log n = \Theta(n \log n)$
- **GetMajorityPartyDelegates:** $T_2(n) = T_1(n) + n = n \log n + n = \Theta(n \log n)$

Problem 6

Overview:

Using the Find-Maximum-Subarray algorithm in 4.1, but the `conquer` return two values more, which are the position of the first negative number in two sides. So the `combine` can be run in time $\Theta(1)$.

Algorithm:

Let the array be A.

Algorithm 4 FindMaximumSubarray

```
function FindMaximumSubarray(A, low, high)
    if low == high then
        if A[low] < 0 then
            return (low, high, low, high, 0, 0, A[low])
        else
            return (low, high, 0, 0, A[low], A[low], A[low])
        end if
    else
        mid = (low + high) / 2
        (leftLow, leftHigh, leftLeftNegative, leftRightNegative, leftLeftSum, leftRightSum,
        leftSum) =
                FindMaximumSubarray(A, low, mid)
        (rightLow, rightHigh, rightLeftNegative, rightRightNegative, rightLeftSum,
        rightRightSum, rightSum) =
                FindMaximumSubarray(A, mid + 1, high)
        crossLow = (leftLeftNegative != 0 ? leftRightNegative + 1 : leftLow)
        crossHigh = (rightRightNegative != 0 ? rightLeftNegative - 1 : rightHigh)
        crossLeftNegative = the first no-zero number in [leftLeftNegative,
        leftRightNegative, rightLeftNegative, rightRightNegative]
        crossRightNegative = the first no-zero number in [rightRightNegative,
        rightLeftNegative, leftRightNegative, leftLeftNegative]
        returnedLeftSum = (leftLeftNegative != 0 ? leftLeftSum : leftLeftSum +
        rightLeftSum)
        returnedRightSum = (rightRightNegative != 0 ? rightRightSum : rightRightSum +
        leftRightSum)
        crossSum = leftRightSum + rightLeftSum
        if leftSum >= rightSum and leftSum >= crossSum then
```

```
            return (leftLow, leftHigh, crossLeftNegative, crossRightNegative,
            returnedLeftSum, returnedRightSum, leftSum)
        else if rightSum > leftSum and rightSum > crossSum then
            return (rightLow, rightHigh, crossLeftNegative, crossRightNegative,
            returnedLeftSum, returnedRightSum, rightSum)
        else
            return (crossLow, crossHigh, crossLeftNegative, crossRightNegative,
            returnedLeftSum, returnedRightSum, crossSum)
        end if
    end if
end function
```

Problem 7

(a)

Let H be the max-head.

Algorithm 5 SecondLargestElement

```
function SecondLargestElement(x)
    h = H.max
    return h.leftChild > h.rightChild ? h.leftChild : h.rightChild
end function
```

(b)

Overview:

We create a new max-heap H and insert the maximum of the max-heap M into H. Then repeat $k-1$ times the operation: extract maximum from H (which is the i^{th} largest element) and then insert the left and right child elements of the popped maximum. Finally, the maximum of H is the k^{th} largest element of M.

Algorithm:

Algorithm 6 SecondLargestElement

```
function kThLargestElement(x)
    H = a new max-heap
    H.insert(M.max)
    for i = 1 to k - 1 do
        node = H.extractMax()
        H.insert(node.leftChild)
        H.insert(node.rightChild)
    end for
    return H.max
end function
```

Time Complexity:

$$T(n) = c_1 + c_2 \lg n + (k-1)(c_3 \lg(k-t) + 2c_2 \lg(k-t)) + c_4 = O(k \lg k)$$