ALG Problem Set 2

Problem 1

To design an algorithm which takes an infix expression and outputs the corresponding postfix expression. The first attempt may use divide-and-conquer method: finds the main operator (the operator that will be executed last) and recursively solves the left and right sub-expressions. If we brute force to find the main operator, the running time will be $O(n^2)$ in the worst case. We can optimize it to $O(n)$ by adding a operator specifier in each recursive call. The idea is: if the main operator is $+$, then in the right sub-expression, there is no more $+$ operator, and the scanner can stop when it meets a $\times$.

```
InfixToPostfix($A, l, r, op$)
// The initial call is InfixToPostfix($A, 1, n, +$).
1: $NEXT = \{+ : \times, \times : !,\ ! : \text{null}\}$
2: if $l > r$ then
3:     return
4: if $l = r$ then
5:     output $A[l]$
6:     return
7: for $i = r$ downto $l$ do
8:     if $A[i] = op$ then
9:         InfixToPostfix($A, l, i-1, op$)
10:        InfixToPostfix($A, i+1, r, NEXT[op]$)
11:        output $A[i]$
12:        return
13: InfixToPostfix($A, l, r, NEXT[op]$)
```

Analysis of running time:

Each character is scanned at most $f(op)$ times, where $f(+) = 3, f(\times) = 2, f(!) = 1$:

Base case: $l = r$, the character is scanned once.

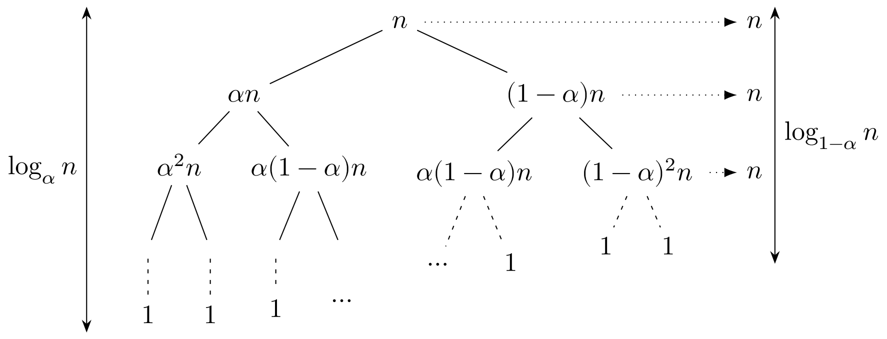
Induction: $l < r$, if the main operator is found at position k, the left sub-array $A[l, k-1]$ is only scanned in line 9 at most $f(op)$ times by induction hypothesis. And the right sub-expression $A[k+1, r]$ is scanned at most $f(NEXT[op]) = f(op)-1$ times in recursion, plus 1 scan in the current call. If the main operator is not found, the whole expression is scanned once, and then the sub-expression is scanned at most $f(NEXT[op])$ times by induction hypothesis. So the total times is at most $1 + f(NEXT[op]) = f(op)$.

So the total running time is $O(n)$.

Problem 2

(a) We guess that $\forall n \geq n_0, T(n) \geq cn$ for some constant $c > 0$. The base case is $T(n_0) = 2 \geq cn_0$, so long as $c \leq 2/n_0$. For the induction step, we have: $T(n) \geq c \cdot \frac{n}{4} + c \cdot \frac{3n}{4} = cn$. The inequality $T(n) \geq cn$ always holds. We can let $c = 2/n_0$ and the induction gives $\forall n \geq n_0, T(n) \geq cn$. So $T(n) = \Omega(n)$.

(b) Without loss of generality, assume $1/2 \le \alpha < 1$, we can draw the recursion tree for $T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$ (the asymptotic constant c is omitted) as follows:



Note that if $\alpha \ne 1/2$, the tree is unbalanced. Assume $T(n) = T(\alpha n) + T((1-\alpha)n) + f(n)$, and $c_1 n \le f(n) \le c_2 n$ for some constants $c_1, c_2 > 0$ and sufficiently large n. For the top $\log_{1-\alpha} n$ levels, it is a full binary tree, and the total cost of each level $\ge c_1 n$. Hence the total cost of the top $\log_{1-\alpha} n$ levels $\ge c_1 n \log_{1-\alpha} n$. $T(n) \ge c_1 n \log_{1-\alpha} n = \Omega(n \log n)$.

If we extend the tree to the bottom, the total number of levels is $\log_\alpha n$. The total cost of each level $\le c_2 n$. Hence the total cost of the whole tree $\le c_2 n \log_\alpha n$. So $T(n) \le c_2 n \log_\alpha n = O(n \log n)$.

As a result, $T(n) = \Theta(n \log n)$.

Problem 3

(a) $S(m) = T(2^m) = 2T(2^{m/2}) + \Theta(m) = 2S(m/2) + \Theta(m)$

(b) By Master Theorem, $S(m) = \Theta(m \log m)$. So $T(n) = S(\log n) = \Theta(\log n \log \log n)$.

(c) Let $m = \log_3 n$ and $R(m) = T(3^m)$, then $R(m) = T(3^m) = 3T(3^{m/3}) + \Theta(3^m) = 3R(m/3) + \Theta(3^m)$. $R(m)$ fits the standard form of Master Theorem: $R(m) = aR(m/b) + f(m)$, where $a = 3, b = 3, f(m) = 3^m$. Since $f(m) = \Omega(m^{\log_3(3)+\epsilon})$ and $3f(m/3) = 3^{m/3+1} \le 3^m = f(m)$, the conditions for case 3 hold. The solution is $R(m) = \Theta(3^m)$. $T(n) = R(\log_3 n) = \Theta(n)$.

Problem 4

The following algorithm takes (A, l, r) and returns $(total, left, right, sum, left\text{-}left,\ left\text{-}right,\ left\text{-}sum,\ right\text{-}left,\ right\text{-}right,\ right\text{-}sum)$ where

- $total = \sum_{i=l}^{r} A[i]$,
- left, right, sum: $A[left, right]$ is the maximun subarray of $A[l, r]$, and $sum = \sum_{i=left}^{right} A[i]$,
- left-left[1], left-right, left-sum: the maximum subarray that must include the left end,
- right-left, right-right[1], right-sum: the maximum subarray that must include the right end.

[1]left-left and right-right can be inferred from l and r. We keep them just for consistency.

And has a running time of $T(n) = 2T(n/2) + O(1)$, or $T(n) = O(n)$.

```
FindMaximumSubarray($A, l, r$)
if $l = r$ then
    return $(A[l], l, r, A[l], l, r, A[l], l, r, A[l])$
$mid = \lfloor (l+r)/2 \rfloor$
$(l\text{-}tot, l\text{-}l, l\text{-}r, l\text{-}sum, !l, l\text{-}l\text{-}r, l\text{-}l\text{-}sum, l\text{-}r\text{-}l, !mid, l\text{-}r\text{-}sum) =$ FindMaximumSubarray$(A, l, mid)$
$(r\text{-}tot, r\text{-}l, r\text{-}r, r\text{-}sum, !mid+1, r\text{-}l\text{-}r, r\text{-}l\text{-}sum, l\text{-}r\text{-}l, !r, r\text{-}r\text{-}sum) =$ FindMaximumSubarray$(A, mid+1, r)$
$tot = l\text{-}tot + r\text{-}tot$
$cross\text{-}sum = l\text{-}r\text{-}sum + r\text{-}l\text{-}sum$
if $l\text{-}sum \geq r\text{-}sum$ and $l\text{-}sum \geq cross\text{-}sum$ then        // Choose the maximum subarray of left, right and cross
    $sum = l\text{-}sum$
    $left = l\text{-}l$
    $right = l\text{-}r$
else if $r\text{-}sum \geq l\text{-}sum$ and $r\text{-}sum \geq cross\text{-}sum$ then
    $sum = r\text{-}sum$
    $left = r\text{-}l$
    $right = r\text{-}r$
else
    $sum = cross\text{-}sum$
    $left = l\text{-}r\text{-}l$
    $right = r\text{-}l\text{-}r$
if $l\text{-}l\text{-}sum \geq l\text{-}tot + r\text{-}l\text{-}sum$ then        // Choose the maximum subarray that must include the left end
    $left\text{-}sum = l\text{-}l\text{-}sum$
    $left\text{-}right = l\text{-}l\text{-}r$
else
    $left\text{-}sum = l\text{-}tot + r\text{-}l\text{-}sum$
    $left\text{-}right = r\text{-}l\text{-}r$
if $r\text{-}r\text{-}sum \geq r\text{-}tot + l\text{-}r\text{-}sum$ then    // Choose the maximum subarray that must include the right end
    $right\text{-}sum = r\text{-}r\text{-}sum$
    $right\text{-}left = r\text{-}r\text{-}l$
else
    $right\text{-}sum = r\text{-}tot + l\text{-}r\text{-}sum$
    $right\text{-}left = l\text{-}r\text{-}l$
return $(tot, left, right, sum, l, left\text{-}right, left\text{-}sum, right\text{-}left, r, right\text{-}sum)$
```

Problem 5

(a)

```
BinaryGCD($a, b$)
if $a = 0$ then
    return $b$
if $b = 0$ then
    return $a$
if $a$ is even and $b$ is even then
    return $2 \times$ BinaryGCD($a/2, b/2$)
else if $a$ is even and $b$ is odd then
    return BinaryGCD($a/2, b$)
else if $a$ is odd and $b$ is even then
    return BinaryGCD($a, b/2$)
else
    if $a \geq b$ then
        return BinaryGCD($(a-b)/2, b$)
    else
        return BinaryGCD($(b-a)/2, a$)
```

(b) The worst-case running time of the binary GCD algorithm is $\Theta(n^2)$.

First we show that it is $O(n^2)$. Let T(m) denote the running time of the binary GCD algorithm for two integers, where m is the total number of bits in the inputs. In each step, we either divide one of the numbers by 2, or subtract one number from the other and then divide by 2. Each operation reduces the size of at least one of the numbers by at least 1 bit. Thus, we have: $T(m) \leq T(m-1) + O(m)$. Or $T(m) = O(m^2)$. The running time when a and b are n-bit integers is $T(2n) = O(n^2)$.

Then we show that it is $\Omega(n^2)$. Constructing a sequence c_k where $c_0 = 0, c_1 = 1$ and $c_k = c_{k-1} + 2c_{k-2}$ for $k \geq 2$. It can be proved that $c_k = \frac{2^k - (-1)^k}{3}$ and c_k has $k-1$ bits when $k \geq 2$. Let $a = c_{n+1}$ and $b = c_n$. The calls to BinaryGCD will be: $\text{GCD}(c_{n+1}, c_n) \to \text{GCD}(c_{n-1}, c_n) \to \text{GCD}(c_{n-1}, c_{n-2}) \to \text{GCD}(c_{n-3}, c_{n-2}) \to \cdots \to \text{GCD}(c_1, c_0) = \text{GCD}(1, 0)$. A $\text{GCD}(c_k, c_{k-1})$ takes $\Theta(k)$ time. So the total time is $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$.

Problem 6

(a) Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$. Then $A^2 = \begin{bmatrix} a \times a + b \times c & (a+d) \times b \\ (a+d) \times c & b \times c + d \times d \end{bmatrix}$. Five multiplications $a \times a, b \times c, d \times d, (a+d) \times b, (a+d) \times c$ are used to compute A^2.

(b) Assume $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, where A, B, C, D are $n/2 \times n/2$ matrices. Then $M^2 = \begin{bmatrix} AA + BC & AB + BD \\ CA + DC & CB + DD \end{bmatrix}$ The recursive subproblems includes not only squaring but general matrix multiplications. And the matrix multiplication is not necessarily commutative. So we cannot use this divide-and-conquer method to reduce the number of multiplications.

(c) $AB + BA = (A+B)^2 - A^2 - B^2$. So we can compute $AB + BA$ through 3 matrix squaring in $3S(n)$ and 2 matrix addition in $O(n^2)$. The total time is $3S(n) + O(n^2)$.

(d) $AB + BA = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$.

(e) First we construct $A = \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix}$ in $O(n^2)$ time. Then we compute $AB + BA$ in $3S(2n) + O((2n)^2)$ time. Finally we extract XY from the result in $O(n^2)$ time. The total time is $3S(2n) + O(n^2)$.

If $S(n) = O(n^c)$. Then $c \geq 2$. Because any general squaring algorithm must read all n^2 entries of the input matrix (suppose it does not read some $A_{i,j}$, then it will give the same output on, i.e.,

$I^2 = I \neq (I + 1_{i,j})^2 = I + 1_{i,j} + (1_{i,j})^2)$. So the time for computing XY is $3S(2n) + O(n^2) = 3O((2n)^c) + O(n^2) = O(n^c)$.