Data Structure and Algorithm
Homework #1

=== Homework Reference Solution ===

***Problem* 1.** Basic Matrix Operation

- Convert infix expression to postfix expression (Please refer to the lecture note "Stacks and Queues" p.24)
  (1) Read the operands or operators from the left.
  (2) After reading an operand, put it into the postfix expression array.
  (3) After reading an operator, compare the precedence of the operator and the operator at the top of the stack:
    i. If the operator that is being read has a larger precedence, push it into the stack.
    ii. Otherwise, pop the operator(s) from the stack, and put them into the postfix expression array until the stack is empty or the operator that is being read has a larger precedence compared to that of the operator at the top of the stack.
  (4) The postfix expression array is the converted postfix expression.

- Matrix operation on postfix expression
  (1) Read the symbols or operation from the left.
  (2) After reading an operand, push the corresponding matrix into the stack.
  (3) After reading an operator, pop two matrices from the stack, calculate them with the corresponding operation, and push the result matrix back into the stack.
    i. If A+B, the dimensions of A and B should be the same, otherwise output "error"
    ii. If A*B, the number of columns of A should equal to the number of rows of B, otherwise output "error"
  (4) At the end of the execution, there is only one matrix in the stack. Output the matrix.

***Problem* 2.** Asymptotic Notation

2.1

(a) Please refer to the lecture note "Basic 1" p.42

(b) $p(n) = n^d\left(a_d + \sum_{i=0}^{d-1} a_i \frac{1}{n^{d-i}}\right), a_d > 0$

There exists positive constants n_0 , $c = (a_d - \sum_{i=0}^{d-1}|a_i|)$ such that

$$0 \le cn^k \le \left(a_d + \sum_{i=0}^{d-1} a_i \frac{1}{n^{d-i}}\right)n^d = p(n) \ \ for\ all\ n \ge n_0$$

(c) By Theorem 3.1 in textbook(Cormen) p.48, if k=d, $p(n) = O(n^k), p(n) = \Omega(n^k), so\ p(n) = \Theta(n^k)$

(d) By (3.1) in textbook(Cormen) p.51, $\lim_{n\to\infty} \frac{p(n)}{n^k} = 0$ (By L'Hôpital's rule), $\therefore p(n) = o(n^k)$

(e) Similar to (d), $\lim_{n\to\infty} \frac{p(n)}{n^k} = \infty$ (By L'Hôpital's rule), $\therefore p(n) = \omega(n^k)$

2.2 (Using base 2 for all *log* operations)

Reference Solution:

$n^{\frac{1}{\log n}}, \sqrt{logn}, log^2 n, 2^{\sqrt{2logn}}, \sqrt{2}^{logn}, nlogn = \log(n!), 4^{logn} = n^2, n^3, n^{loglogn} = (logn)^{logn}, (\frac{3}{2})^n, e^n, n!$

Brief proof:

(1) $n^{\frac{1}{\log n}} = 2^{logn\frac{1}{logn}} = 2$

(2) Let $t = \sqrt{logn}, \sqrt{logn} = t, log^2 n = t^4, 2^{\sqrt{2logn}} = 2^{\sqrt{2}t}, \sqrt{2}^{logn} = \sqrt{2}^{t^2}$

(3) $\sqrt{2}^{logn} = \sqrt{n}$

(4) $\log(n!) = \Theta(nlogn)$ ($\int_1^n logx\,dx = nlogn - n + 1$, For more information, please refer to: http://www.cs.sfu.ca/CourseCentral/307/jmanuch/lec/factorial.pdf)

(5) $\lim_{n\to\infty} \frac{e^n}{(\frac{3}{2})^n} = \infty$

(6) $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$ by Stirling's approximation

***Problem* 3.** Time Complexity

3.1

There are *m* elements in A and B, i.e., there are $\sqrt{m}$ columns and rows in A and B. Let C = AB, $C_{ij} = \sum_{k=0}^{\sqrt{m}-1} A_{ik}B_{kj}$, there are *m* elements, and each element needs approximately $\sqrt{m}$ multiplications and additions. Therefore, total time complexity is $O(m\sqrt{m})$.

3.2

Algorithm

```
for i = 0 to m-1
    for j = 0 to m-1
        C[i][j] = ci+i+j;
            D[i][j] = di+i-j;
    end
end
for i = 0 to m-1
    for j = 0 to m-1
            for k = 0 to m-1
                E[i][j] = C[i][k] * D[k][j];
        end
    end
end
```

Time complexity:

The loop in the first half runs for m^2 times, and the loop in the second half runs for m^3 times. Therefore, the total time complexity is $O(m^3)$.

3.3

Although we use the same algorithm to multiply two matrices, the input size "*m*" is defined differently in 3.1 and 3.2. We express the complexity of an algorithm with its input size, and therefore when the input size is defined differently two identical algorithms would appear to have different time complexity expression.

***Problem* 4.** Stacks and Queues

4.1
Given two queues, Queue1 and Queue2, and we have to implement a stack using these two queues. There are many approaches to solve this question, and the following are two of these approaches.

Approach 1:

Push a new element X:

Just enqueue X into Queue1.

Pop:

1. Dequeue all the items in Queue1 and enqueue them into Queue2 until Queue1 has only one element.
2. Dequeue the last item in Queue1 and store it in a temporary variable.
3. Dequeue all items in Queue2 and enqueue them back into Queue1.
4. Return the pre-stored temporary variable.

Approach 2:

Push a new element X:

If the Queue1 is empty, just enqueue X into Queue1.

Otherwise, enqueue X into Queue2. Then, dequeue all the items in Queue1 and enqueue them into Queue2. Finally, dequeue all the items in Queue2 and enqueue them into Queue1.

Pop:

Just dequeue and return the item from Queue1.

4.2
Given two stacks, Stack1 and Stack2, and we have to implement a queue using these two stacks. There are many approaches to solve this question, and the following are two of these approaches.

Approach 1:

Enqueue a new element X:

Just push X into Stack1.

Dequeue:

1. Pop all the items in Stack1 and push them into Stack2 until Stack1 has only one element.
2. Pop the last item in Stack1 and store it in a temporary variable.
3. Pop all the items in Stack2 and push them back into Stack1.
4. Return the pre-stored variable.

Approach 2:

Enqueue a new element X:

If Stack1 is empty, directly push X into Stack1.

Otherwise, pop all the items in Stack1 and push them into Stack2. Then push the X into Stack1. Finally, pop all the items in Stack2 and push them back into Stack1.

Dequeue:

Just pop and return the item from Stack1.

4.3
Given a stack and a queue, we can use following algorthm to determine if the stack and the queue has the same elements with same inserted order.

Algorithm isEquivalent(stack S, queue Q)

```
int size = 0;
while(S.isEmpty() == FALSE){
```

```
    Q.enqueue(S.pop());
    size += 1;
  }
  for (i = 0; i < size; i++){
    S.push(Q.dequeue());
  }
  for (i = 0; i < size; i++){
    if (S.isEmpty() == TRUE or Q.isEmpty() == TRUE){
          return FALSE;     // Q and S are different.
      }
      else if (S.pop() != Q.dequeue()){
          return FALSE;     // Q and S are different.
      }
  }
  return TRUE; // Q and S have the same elements in the same order.
```

Time complexty = O(*N*) + O(*N*) + O(*N*) = O(*N*), because all for loops consume at most O(*N*) time.

Additional Space = O(*1*), because there is only one additional integer being used.

***Problem* 5.** 2-SUM and 3-SUM

5.1

Algorithm 2-SUM brute-force

```
int M = 0;
for (i = 0; i < n - 1; i++){
    for (j = i + 1; j < n; j++){
        if (a[i] + a[j] == 0) M += 1;
    }
}
```

Time complexity:

If the number of elements of the given array is *N*, the brute-force algorithm has to do the "if (a[i] + a[j] == 0)" for $\frac{N(N-1)}{2}$ times. Thus the time complexity is O(N^2).

5.2

Algorithm 2-SUM binary search

```
int M = 0;
for (i = 0; i < n - 1; i++){
    // do the binary search to find if there exists an element with value –a[i]
in array a.
    if (binarySearch(a, -a[i]) == TRUE) M += 1;
}
```

Time complexity:

The time complexity of binary search is O(*log N*). Above algorithm has to do the binarySearch() *N* times if the number of elements in array a is *N*. Thus the time complexity of the algorithm is O(*N log N*).

5.3

The worst case: there does not exist any combination of *x, y, z* to satisfy $a_x + a_y + a_z = 0$, or the indices *x, y, z* correspond to the last three elements in array a. In this case, we have to execute the while loop for *N*-2 times if the number of elements in array a is *N*, and each while loop has to take O(*N*) time. Thus, the time complexity in the worst case is O(N^2).

5.4

We can use the concept of problem 5.3 to design the linear time complexity algorithm to solve problem 5.1.

Algorithm 2-SUM linear time algorithm

```
int M = 0;
i = 0;
j = n - 1;
while (i < j) {
    if (a[i] + a[j] == 0) {
        M += 1;
        i += 1;
        j -= 1;
    else if (a[i] + a[j] > 0) {
        j -= 1;
```

```
        }
        else {
            i += 1;
        }
    }
```

Time complexity:

The above algorithm executes "if… else if… else…" for at most $N - 1$ times, and thus the time complexity of the algorithm is O(N) (linear time).