Problem 1

1. $\Theta(n) = n$

# of iterations	n
0	0
1	1
2	2
3	3
4	4

2. $\Theta(n) = \lg(\lg n)$

# of iterations	n	$\lg(n)$	$\lg(\lg(n))$
0	2^1	1	0
1	2^2	2	1
2	2^4	4	2
3	2^8	8	3
4	2^{16}	16	4

3. $\Theta(n) = 4^n$

Tree level	# of calls	Input number
0	4^0	n
1	4^1	$n-1$
$\vdots$	$\vdots$	$\vdots$
$n-2$	$4^{n-87506055}3^{87506055-2}$	2
$n-1$	$4^{n-87506055}3^{87506055}$	1

$4^{n-87506055}3^{87506055} = \Theta(n) \implies 4^n = \Theta(n)$

4. True.

Obviously $f+g$ itself belongs to the set $\Theta(f+g)$,
$\implies (f+g)c_2 \le f+g \le (f+g)c_1$, where c_1 and c_2 are positive constants.
$\implies c_2\max(f,g) \le c_2(f+g) \le f+g \le c_1(f+g) \le 2c_1\max(f,g)$
$\implies c_2\max(f,g) \le f+g \le c_1'\max(f,g)$, where $c_1' = 2c_1$
$\implies f+g = \Theta(\max(f,g))$.

5. True.

$f = O(i)$ and $g = O(j)$
$\implies$ There are two positive constants such that $0 \le f \le c_1 i$ and $0 \le g \le c_2 j$.
$\implies 0 \le fg \le c_1c_2ij \implies fg = O(ij)$.

6. False.

If $f = 2\lg n$ then $g = \lg n$
$\implies 2^f = 2^{2\lg n} = 2^{\lg n^2} = n^2$
On the other hand, $2^g = 2^{\lg n} = n$
$\implies 2^f = O(n^2) \ne O(n) = 2^g$

7. True.

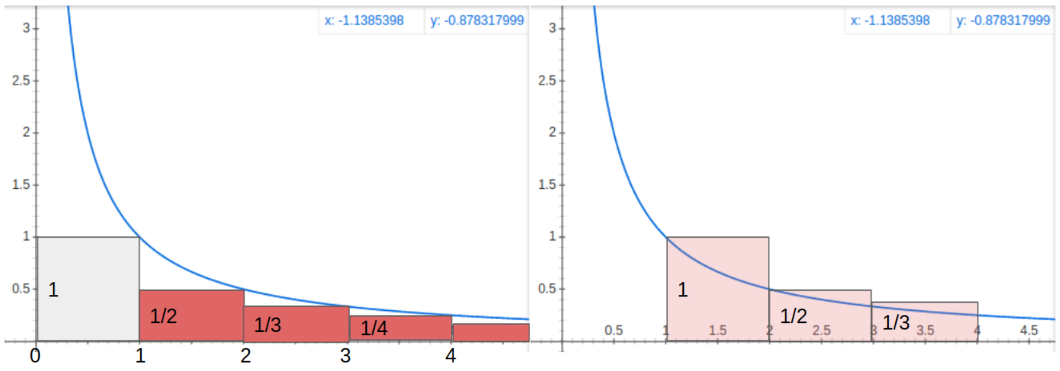
As shown in the left figure, the total area of the red rectangles is obviously smaller than that under the curve $1/x$(blue line) between $1 < x < n$. Hence, we obtain the upper bound as follows:

$$\sum_{k=1}^{n} \frac{1}{k} < 1 + \int_1^n \frac{1}{x} dx = 1 + \log n \tag{1}$$

Similarly, the lower bound can be obtained from the right figure:

$$\sum_{k=1}^{n} \frac{1}{k} > \int_1^{n+1} \frac{1}{x} dx = \log(n+1) \tag{2}$$

Finally, since both lower and upper bounds have the asymptotic form $\log n$, we have $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg n)$.



8. True.

   On the basis of approximation as follows:

$$\lg n! = \sum_{k=1}^{n} \lg k \sim \int_1^n \lg x dx = n \lg n - n + 1, \tag{3}$$

   we have $\lg n! = \Theta(n \lg n)$.

9. $f(n) = \Theta(n(\lg n)^2)$.

$$\begin{aligned}
f(n) &= 2f\left(\frac{n}{2}\right) + n \lg n \\
&= 2\left[2f\left(\frac{n}{2^2}\right) + \frac{n}{2} \lg\left(\frac{n}{2}\right)\right] + n \lg n \\
&\vdots \\
&\sim \overbrace{2 \times \cdots \times 2}^{\sim \lg n \text{ times}} \times f(1) + \overbrace{(n \lg 1) + \cdots + \left(n \lg\left(\frac{n}{2^2}\right)\right) + \left(n \lg\left(\frac{n}{2}\right)\right) + (n \lg n)}^{\sim \lg n \text{ times}} \qquad (4) \\
&\sim 2^{\lg n} + n \lg\left(\frac{n^{\lg n}}{2^{\lg n + \cdots + 2 + 1}}\right) \\
&\sim n + n(\lg n)^2 - n \lg\left(2^{\lg n \times (\lg n + 1)/2}\right) \\
&\sim n + n(\lg n)^2 - n\left[0.5 \lg n \left(\lg n + 1\right)\right] \\
&\sim n + 0.5n(\lg n)^2 - 0.5n \lg n.
\end{aligned}$$

   Hence, $f(n) = \Theta(n(\lg n)^2)$.

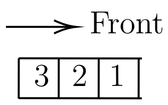
Problem 2

1. Reverse queue with another empty queue

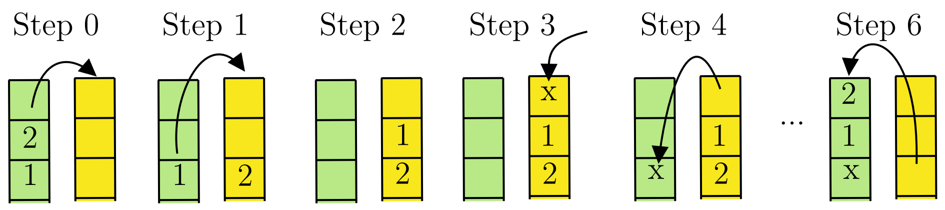
```
Input: 1,2,3,4,5
Output: 5,4,3,2,1
Q1 = InitQueue (1,2,3,4,5);
Q2 = InitQueue ();
while GetSize(Q1) is greater than zero do
    while the last element is at the front of Q1 do
        Q1 = EnQueue(DeQueue(Q1));
    Q2 = EnQueue(DeQueue(Q1));
```

2. The number of iterations of outer loop: the size of initial queue, n.
   The number of iterations of inner loop: $n-k$, where k is the # of iterations of the outer loop at k-th times.
   Hence, the time complexity is $n + (n-1) + \cdots + 1 = 0.5n(n+1) = O(n^2) -$ time.
   We do not use the additional third array or queue to reverse queue. Thus extra-space complexity is $O(1)$.

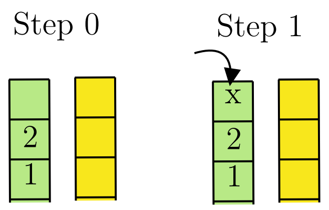
3. Illustration



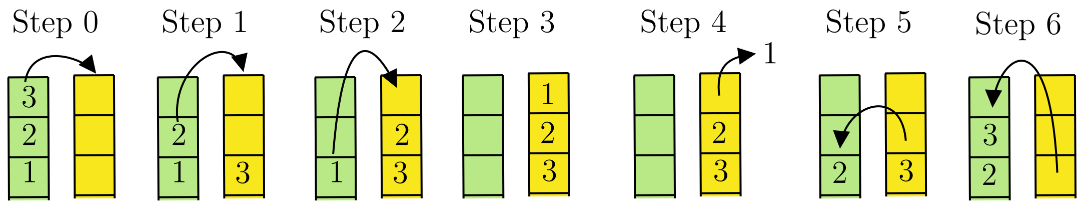
a. `push_front(int x)`



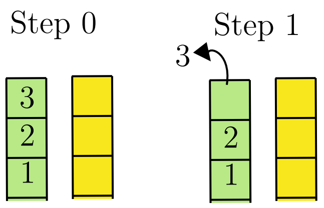
b. `push_back(int x)`



c. `pop_front()`



d. `pop_back()`



4. `push_front()`: $O(n^2) - time$

   Pop all elements from the left stack n times, and push all elements, including x, back $n+1$ times. Hence, we have $O(n^2)$-time.

5. `push_back()`: $O(1) - time$

   No matter how many elements in the stack are, we only have to push one time. Hecce, $O(1)$-time.

6. `pop_front()`: $O(n^2) - time$

   Pop all elements from the left stack n times, push the front element, and then push all elements, including x, back n times. Hence, we have $O(n^2)$-time.

7. `pop_back()`: $O(1) - time$

   No matter how many elements in the stack are, we only have to pop one time. Hecce, $O(1)$-time.

8. Dynamic stack

Assuming a is the size of initial stack, and q is the number of `enlarge()` calls. We have $a3^q = n$ or $q = \log_3(n/a)$. On the other hand, If $t = c_0 m$ is the time required for single `enlarge()` calls with m elements, then the total time required for all reallocations is

$$c_0 a + c_0 a 3^1 + c_0 a 3^2 + \cdots + c_0 a 3^{q-1} = 0.5 c_0 a \left(3^{q+1} - 1\right) = \Theta(3^q). \tag{5}$$

Substituting $q = \log_3(n/a)$ into Eq.(5) yields the time complexity $\Theta(n)$ required for n consecutive push operations into dynamic stack.

Problem 3

1. Code:

```c
int main(){

  int A[6] = {2,0,1,3,4,3}; // The array given by statement
  int i=3;                  // initial position of the frog
  int j=0;                  // counter of loop

  while(1){

    int next = A[i];

    if ( A[i] == i ){
      printf("Frog will stop!\n");
      break;
    }
    else if( j == 6 ){
      printf("Frog will keep jumping forever!\n");
      break;
    }

    i = next;
    j++;
  }

  return 0;

}
```

   - Time complexity is $\Theta(n)$ since we loop over the array `A[i]`. If the counter `j` equals to the length of the array `A[i]`, then we stop the loop and think the frog will keep jumping forever.
   - Extra-space complexity is $\Theta(1)$ because we do not use the helper array to keep track where the frog passes.

2. Code:

```c
int main(){

  int A[8] = {1,0,4,2,3,1,4,6};   // The array given by statement
  int B[8] = {0};                 // helper array to keep track of where the frog pass
  int i = 2;                      // Initial position of the frog
  int j = 0;                      // loop counter



  while(1){

    if (B[i] != 1){
      B[i] = 1;
      j++;
    }else{
      printf("The length of the loop is %d\n", j);
      break;
    }

    i = A[i];
  }

  return 0;
}
```

- Time complexity is $\Theta(n)$ since we loop over the array `A[i]`. We stop the loop and return the counter `j` which represent the length of the loop when the helper array `B[i]` equals to 1.
- Extra-space complexity is $\Theta(n)$ because the frog may enter the infinite loop from non-initial position, and thus we need the helper array `B[i]` to keep track where the frog passes.

3. find the minimum of $f(i, j)$

   - Time complexity is $\Theta(n)$.
   - Extra-space complexity is $\Theta(1)$.

   Since the array is strictly increasing, the two indice `i` and `j` must be as close as possible so that $f(i, j)$ can reach the minimum. i.e. $\texttt{j} = \texttt{i} + 1$. Using the constraint $\texttt{j} = \texttt{i} + 1$, we simply loop the index `i` over the array and compare the current $f(i, j)$ with the previous one to find the minimum $f(i, j)$.

4. Circularly linked list (CLL):

   a) Loop over the CLL from the head find the node, say `A`, having the maximum data.
   b) Loop over again the CLL from the node `A->next` instead of `A`.
   c) During the loop, if the next data is smaller than the current data, we remove the next node from the CLL and store the data in the next node in helper array. Note that the size of helper array is expected to be 2 as the CLL only have 2 decreasing.

d) After the second loop, we finally loop over the CLL and insert the data into the proper position.
e) In summary, we loop the CLL 3 times and use one helper array with length 2. Thus, the time complexity is $O(n)$, and the extra-space complexity is $O(1)$.