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 $\Theta(f+g)$,
$\implies (f+g)c_2 \leq f+g \leq (f+g)c_1$, where c_1 and c_2 are positive constants.
$\implies c_2\max(f,g) \leq c_2(f+g) \leq f+g \leq c_1(f+g) \leq 2c_1\max(f,g)$
$\implies c_2\max(f,g) \leq f+g \leq c_1'\max(f,g)$, 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 $0 \leq f \leq c_1 i$ and $0 \leq g \leq c_2 j$.
$\implies 0 \leq fg \leq c_1c_2ij \implies fg = O(ij)$.

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, $2^g = 2^{\lg n} = n$
$\implies 2^f = O(n^2) \neq O(n) = 2^g$

7. True.

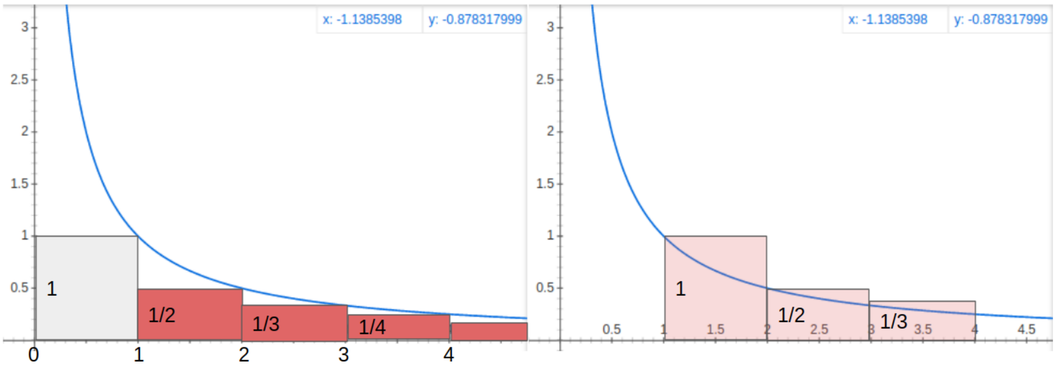
As shown in the left figure, the total area of the red rectangulars is obviously smaller than that under the curve $1/x$(blue line) between $1 < x < n$. Hence, we obtain the upper bound as follows:

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

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

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

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

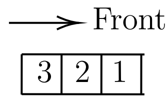
Problem 2

1. Reverse queue with another empty queue

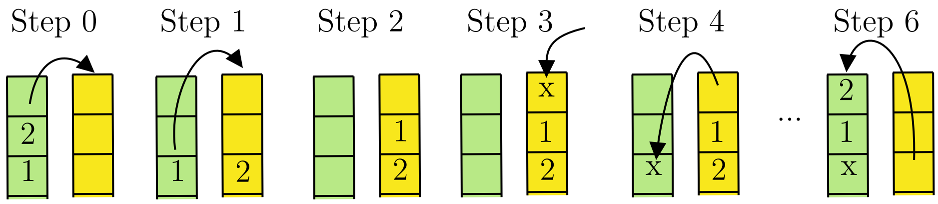
   Input: 1,2,3,4,5
   Output: 5,4,3,2,1
   Q1 = `InitQueue` (1,2,3,4,5);
   Q2 = `InitQueue` ();
   while `GetSize(`*Q1*`)` *is greater than zero* **do**
       while *the last element is at the front of Q1* **do**
           Q1 = `EnQueue(DeQueue(`*Q1*`))`;
       Q2 = `EnQueue(DeQueue(`*Q1*`))`;

2. The # of iterations of outer loop: the size of initial queue, n.
   The # 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. How about $O(1)$-extra space?

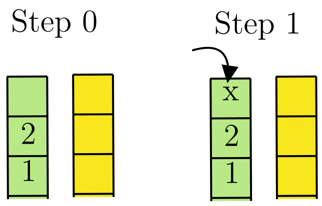
3. Illustration



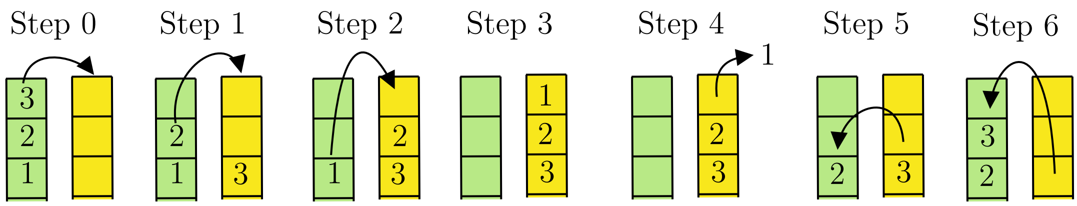
a. `push_front(int x)`



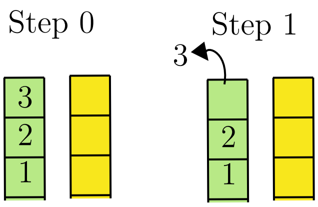
b. `push_back(int x)`



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 $\Theta(n)$ required for n consecutive push operations into dynamic stack.

Problem 3

1. Pseudo code:

```
Input: A is the array containing n integers ∈ {0, 1, . . . , n − 1}
Input: i_0 is the index where the frog is initially located.
List = initalList(A[i_0]);
    /* initialize the list with the single node A[i_0] */
for (i = 0; i < A.Length; i + +) do
    if i == List.End then
            /* List.End is the the last element of List   */
        return [Finite jumps!]
    List = AppendEnd(List, A[List.End]);
        /* append the node A[List.End] to the end of List   */
return [Infinite jumps!]
```

- Time complexity: The time complexity of loop itself is $\Theta(n)$ but `List.End`, which is $\Theta(n)$, is inside the loop. Thus, the effective time complextity is $\Theta(n^2)$.

- Space complexity: $\Theta(n)$ (Why?).

2. Pseudo code:

```
Input: A is the array containing n integers ∈ {0, 1, . . . , n − 1}.
Input: i_0 is the index where the frog is initially located.
Output: j is the length of the jumping loop.
List = initalList(A[i_0]);
    /* initialize the list with the single node A[i_0] */
for (i = 0; i < A.Length; i + +) do
    List = AppendEnd(List, A[List.End]);
        /* append the node A[List.End] to the end of List   */
    for (j = List.Length − 2; j >= 0; j − −) do
        if List(j) == List.End then
                /* List(j) is the j^th element of List   */
            return j
```

- Time complexity: The time complexity of loop itself is $\Theta(n)$ but `List.End`, which is $\Theta(n)$, is inside the loop. Thus, the effective time complextity is $\Theta(n^2)$.

- Space complexity: $\Theta(n)$ (Why?).

3. Given that the array A is a strictly increasing array, $\max(M_{0,i}, M_{i,j}, M_{j,n})$ and $\min(M_{0,i}, M_{i,j}, M_{j,n})$ must be equal to $M_{j,n}$ and $M_{0,i}$, respectively. Moreover, the two elements $A[i-1]$ and $A[j]$ must be as close as possible to make $f(i,j) = M_{j,n} - M_{0,i}$ reach the minimum. We thus simplify the original question as follows: find a index k, such that $M_{k+1,n} - M_{0,k-1}$ is minimized, where $i = k$ and $j = k+1$.

Input: a strictly increasing array, A.
Output: i, j.
for ($m = 1$; $m <$ A.*Length*-1; $m++$) **do**
    F[*m-1*] = `GetMedian(`*m+1*,*n*`)-GetMedian(`*0*,*m-1*`)`;
k=`GetMinIdx(`F`)`+1
return *k*, *k+1*

- Time complexity: The time complexity of loop itself is $\Theta(n)$ but `List.End`, which is $\Theta(n)$, is inside the loop. Thus, the effective time complextity is $\Theta(n^2)$.

- Space complexity: $\Theta(n)$ (Why?).

4.

```
/* Step 1: Get the length of list */
/*========================================*/
Ptr = Head;
Length = 1

while( Ptr->Next != Head )
{
   Ptr = Ptr->Next;
   Length++;
}

/* Step 2: Sorting list with bubble sort */
/*========================================*/
itr_out = 0; /*iteration number of the outer loop*/
Swap = true; /*Do not enter the outer loop if Swap=false,
which can help the time complexity in the worst case down to O(n)*/
Ptr = Head;

while( Ptr->Next != Head && Swap )
{
   Ptr_in = Head;
   Swap = false;
   itr_in = 0;  /*iteration number of the inner loop*/

   while( Ptr_in->Next != Head && itr_in < Length - itr_out - 1 )
   {
```

```
      if ( Ptr_in->Next->Data  <  Ptr_in->Data )
      {
         swap( Ptr_in->Next->Data, Ptr_in->Data );
         Swap = true;
      }
      Ptr_in = Ptr_in -> Next;

      itr_in++;
   }
   Ptr = Ptr->Next;
   itr_out++;
}
```