DSA Non-Programming Problem Solving

Problem 1 - Complexity

> Reference for drawing math graph:https://www.desmos.com/calculator

1. The loop will iterate k times which $1+2+3...+k=\frac{(1+k)*k}{2}\leq n$

   $k^2+k\leq 2n\Rightarrow k^2+k-2n\leq 0\Rightarrow k\leq \frac{-1+\sqrt{1+8n}}{2}$(positive is not in consonance)
   $=\Theta(\sqrt{n})$

2. The loop will iterate k times which $m^{(2^k)}<n$

   $m^{(2^k)}<n\Rightarrow 2^k<\log_m n\Rightarrow k<log_2(log_m n)\Rightarrow k<log_2(\frac{logn}{logm})\Rightarrow k<log_2(\log n)-log_2(logm$

3. The recursion will cause two condition:

   - n <= 87506055, the total step is 3^n
   - n > 87506055, the total step is $4^{n-87506055}3^{87506055}$

   We only have to consider the condition that n is large enough because 3^n is true in a limited n value, so the total step $4^{n-87506055}3^{87506055}=\Theta(4^n)$

4. By the definition of Θ , $q(n)=\Theta(k(n))$ exist $c_1\in\mathbb{N}^+\wedge c_2\in\mathbb{N}^+$ fulfill
   $c_1k(n)\leq q(n)\leq c_2k(n)$ for $n\geq N$

   max$(f(n),g(n))\leq f(n)+g(n)\leq 2max(f(n),g(n))$

   So, $f(n)+g(n)=\Theta(max(f(n),g(n)))$ for $n\in\mathbb{N}^+$

   $c_1=1,c_2=2,k(n)=max(f(n),g(n)),q(n)=f(n)+g(n),n\in\mathbb{N}^+$

   Q.E.D

5. By the definition of $\mathbb{O}$, $q(n)=\mathbb{O}(k(n))$ exist $c\in\mathbb{N}^+$ fulfill $q(n)\leq ck(n)$ for $n\geq N$

   $f(n)=\mathbb{O}(i(n))$ exist c_1 that $f(n)\leq c_1i(n)$ for $n\geq N_1$

   $g(n)=\mathbb{O}(j(n))$ exist c_2 that $g(n)\leq c_2j(n)$ for $n\geq N_2$

   $f(n)*g(n)\leq c_1i(n)c_2j(n)=c_1c_2i(n)j(n)$

   So,$f(n)*g(n)=\mathbb{O}(i(n)j(n))$ for $n\geq$ max(N_1,N_2)

   $k(n)=i(n)*j(n),c=c_1c_2,q(n)=f(n)*g(n),n=$ max(N_1,N_2)

   Q.E.D

6. By the definition of $\mathbb{O}$, $q(n)=\mathbb{O}(k(n))$ exist $c\in\mathbb{N}^+$ fulfill $q(n)\leq ck(n)$ for $n\geq N$

   $f(n)=\mathbb{O}(g(n))$ exist c_1 that $f(n)\leq c_1g(n)$ for $n\geq N_1$

   $2^{f(n)}\leq 2^{c_1g(n)}$

   so, $2^{f(n)}=\mathbb{O}(2^{c_1g(n)})$

   $c=1,q(n)=2^{f(n)},k(n)=2^{c_1g(n)},N=N_1$

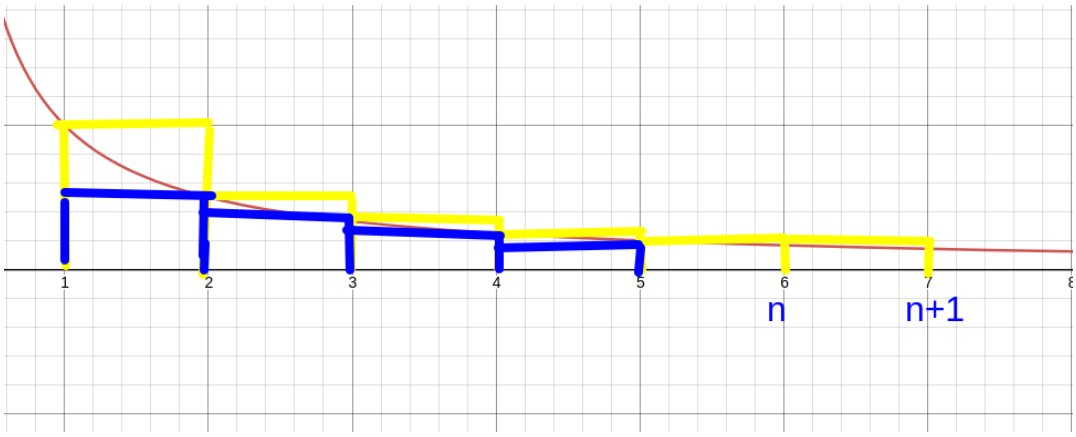
   So $2^{f(n)}=\mathbb{O}(2^{g(n)})$ if and only if $c_1=1$

   Q.E.D

7. By the definition of Θ , $q(n) = \Theta(k(n))$ exist $c_1 \in \mathbb{N}^+ \wedge c_2 \in \mathbb{N}^+$ fulfill
   $c_1 k(n) \leq q(n) \leq c_2 k(n)$ for $n \geq N$

   $ln(n) = \Theta(lg(n)) = \int_{x=1}^{n} \frac{1}{x} dx$

   Compare with sigma and integral in below figure



   The area below the red line is the area of $y = \frac{1}{x}$ which means $\int_1^{n+1} \frac{1}{x} dx$

   And the area of yellow line is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} \ldots + \frac{1}{n}$ which means $\Sigma_{x=1}^{n} \frac{1}{x}$

   The area of blue line is $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots + \frac{1}{n+1}$ which means $\Sigma_{x=2}^{n+1} \frac{1}{x} dx$

   Obviously, we can observe that $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots + \frac{1}{n+1}$ which means
$\Sigma_{x=2}^{n+1} \frac{1}{x} \leq \int_1^{n+1} \frac{1}{x} dx \leq \Sigma_{x=1}^{n} \frac{1}{x}$

$\int_1^n \frac{1}{x} dx < \int_1^{n+1} \frac{1}{x} dx < \Sigma_{x=1}^{n} \frac{1}{x} = 1 + \Sigma_{x=2}^{n} \frac{1}{x} < 1 + \Sigma_{x=2}^{n+1} \frac{1}{x} \leq 1 + \int_1^{n+1} \frac{1}{x} dx \leq 2 \int_1^n \frac{1}{x} dx$
for $n \geq \frac{e+\sqrt{e^2+4e}}{2}$

   So $\ln n = \int_1^n \frac{1}{x} dx \leq \Sigma_{k=1}^{n} \frac{1}{k} \leq 2 \int_1^n \frac{1}{x} dx = 2 \ln n$

   So we proved that $\Sigma_{k=1}^{n} \frac{1}{k} = \Theta(\lg n)$ for $n \geq \frac{e+\sqrt{e^2+4e}}{2}$

   $c_1 = 1, c_2 = 2, k(n) = \lg n, q(n) = \Sigma_{k=1}^{n} \frac{1}{k}, N = \frac{e+\sqrt{e^2+4e}}{2}$

   Q.E.D

8. By the definition of Θ , $q(n) = \Theta(k(n))$ exist $c_1 \in \mathbb{N}^+ \wedge c_2 \in \mathbb{N}^+$ fulfill
   $c_1 k(n) \leq q(n) \leq c_2 k(n)$ for $n \geq N$

   $\frac{n}{2} lg(\frac{n}{2}) = lg(\frac{n}{2}) + lg(\frac{n}{2}) + lg(\frac{n}{2}) \ldots$(total $\frac{n}{2}$ terms)
   $<= lg(1) + lg(2) + lg(3) \ldots + lg(n) = \lg(n!) \leq \lg(n^n) = n \lg(n)$ for $n \in \mathbb{N}^+$

   So we proved that $\lg(n!) = \Theta(n \lg n)$ for $n \in \mathbb{N}^+$

   $c_1 = \frac{1}{2}, c_2 = 1, q(n) = \lg(n!), k(n) = n ln(n), N = \mathbb{N}^+$

   Q.E.D

9. By the definition of Θ , $q(n) = \Theta(k(n))$ exist $c_1 \in \mathbb{N}^+ \wedge c_2 \in \mathbb{N}^+$ fulfill
   $c_1 k(n) \leq q(n) \leq c_2 k(n)$ for $n \geq N$

   $f(n) = 2(f([\frac{n}{2}]) + n \lg n) = 2(2(f([\frac{n}{4}] + \frac{n}{2} \lg(\frac{n}{2}))) + n \lg n) = 2(n \lg n + n \lg(\frac{n}{2}) + n \lg(\frac{n}{4}) \ldots + 1)$
   , total $\log_2 n$ terms.

   $\frac{\log_2 n}{2}(n \lg n) = 2(\frac{\log_2 n}{2})(n \lg(\sqrt{n})) = 2(n \lg \sqrt{n} + n \lg \sqrt{n} + n \lg \sqrt{n} \ldots + n \lg \sqrt{n}) \leq 2(n \lg n + n l$

$f(x) = 2(nlgn + nlg(\frac{n}{2}) + nlg(\frac{n}{4}) \ldots +1) \leq 2(n\lg n + n\lg n \ldots +n\lg n) = 2log_2 n(n\lg n)$ for$n \in \mathbb{N}^+$

So, $\frac{nlog_2(n)\lg n}{2} \leq f(x) \leq 2nlog_2(n)\lg n$ for$n \in \mathbb{N}^+$

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

$c_1 = \frac{1}{4}, c_2 = 1, q(n) = f(x), k(n) = n(\lg n)^2, N = \mathbb{N}^+$

Q.E.D

10. By the definition of $\mathbb{O}$, $q(n) = \mathbb{O}(k(n))$ exist $c \in \mathbb{N}^+$ fulfill $q(n) \leq ck(n)$ for $n \geq N$

    Because $f_k(n) = \mathbb{O}(n^2)$, so for each $f_k(n)$, there exist a c_k that $f_k(n) \leq c_k n^2$ for $n >= N_k$

    So
    $\Sigma_{k=1}^{n} f_k(n) = f_1(n) + f_2(n) \ldots + f_n(n) \leq c_1 n^2 + c_2 n^2 \ldots + c_k n^2 = (c_1 + c_2 \ldots + c_k)n^2$ for $n \geq \max(n_k)$

    $\Sigma_{k=1}^{n} f_k(n) = \mathbb{O}(n^2) = \mathbb{O}(n^3)$(Because $n^3 \geq n^2$ when $n >= 1$) for $n \geq \max(n_k)$

    $c = (c_1 + c_2 + c_3 \ldots + c_k), q(n) = \Sigma_{k=1}^{n} f_k(n), k(n) = n^2, N = \max(n_k)$
    $\Sigma_{k=1}^{n} f_k(n) = \mathbb{O}(n^2) = \mathbb{O}(n^3)$(Because $n^3 \geq n^2$ when $n >= 1$) for $n \geq \max(n_k)$

    Q.E.D

11. $n = k_1 m + r_1 \geq m + r1 > r_1 + r_1 = 2r_1, n > 2r_1, \frac{n}{2} > r_1$

    $m = r_1 k_2 + r_2 \geq r_1 + r_2 > 2r_2, m > 2r_2, \frac{m}{2} > r_2$

    Consider the worst case that we only decrease the value of m and n by divide two each times, and the recursion stop when m or n equals to 0, so we can get log_2(min(m,n)) $= k$ while k is the step we takes.

    So the time complexity is O(lg(m+n)) because log_2(min(m,n)) $= O$(lg(min(m,n))) $= O(lg(m+n))$

    Q.E.D

Problem 2 - Stack/Queue

> Reference for pseudo code: https://michaelchen.tech/blog/how-to-write-pseudocode/
>
> Reference for Amortized Time Complexity:https://medium.com/@satorusasozaki/amortized-time-in-the-time-complexity-of-an-algorithm-6dd9a5d38045
>
> Reference for how I draw picture:https://magiclen.org/kolourpaint/
>
> Reference of case of time complexity: https://zhuanlan.zhihu.com/p/113526827

1.
```
procedure reverse(source, help):
    while source.size() > 2:
        help.enqueue(source.front())
        source.dequeue()
    source.enqueue(source.front())
    source.dequeue()
    end while
    totalEle <- help.size()
    for i (0 to totalEle - 1) do
        for j (0 to help.size() - 2) do
```

```
            help.enqueue(help.front())
            help.dequeue()
        end for
        source.enqueue(help.front())
        help.dequeue()
    end for
```

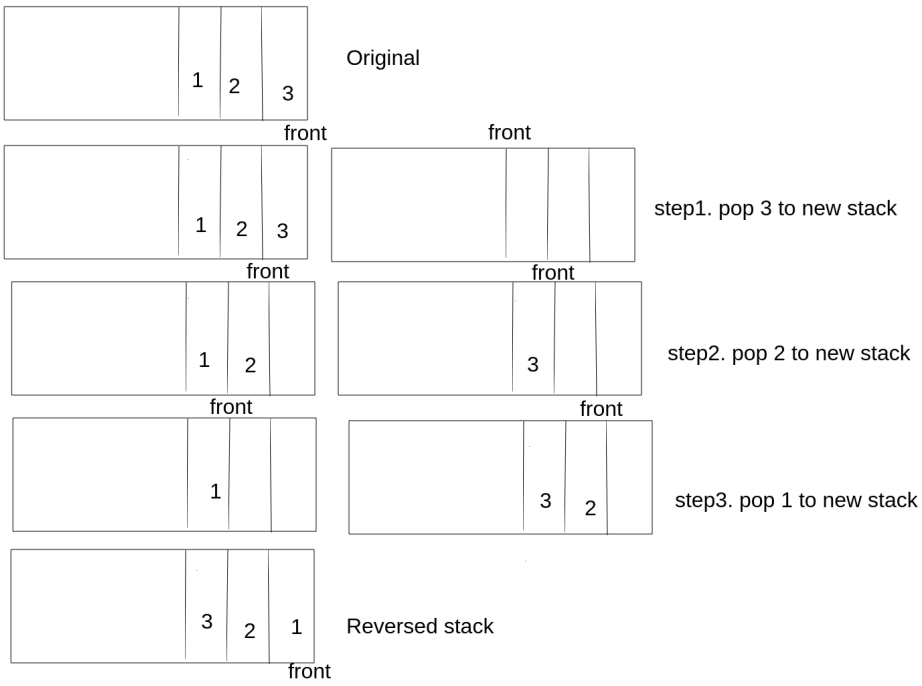
2. Prove:

   while in line 2 iterates $n-2$ and times

   for in line 4-6 iterates $n-2+(n-3)+(n-4)...+1=\frac{(n-2)(n-1)}{2}$

   total iterates $n-2+\frac{(n-2)(n-1)}{2}=\frac{n(n-2)}{2}=\mathbb{O}(n^2)$

3. A stack has a property that when we pop all the element from one stack and push to another stack, the order of the elements of the stack will be reversed. Below is an easy example.



With this property, we can use two stacks(assume being called with front and back) as two sides of a deqeu, and use a variable called dir(Here we use value 1 for front, -1 for back) to record the present direction. When we want to change the direction of operation, we only have to push all element from one to another, then we can let the bottom be the top of the stack, the top be the bottom of the stack. Thus we successfully operates on two sides of the deque.

Here is the pseudo code

```
front and back are two stack and dir is an int
deq is a deque
   struct deque:
    front
    back
    dir
end struct

procedure push_front(x):
```

```
    if dir == -1 then
        while back.size() > 0:
            front.push(back.front())
            back.pop()
        end while
        dir = 1
    end if
    front.push(x)
   end procedure

procedure pop_front():
    if dir == -1 then
        while back.size() > 0:
            front.push(back.front())
            back.pop()
        end while
        dir = 1
    end if
    set num to front.front()
    front.pop()
    return num
end procedure

procedure push_back(x):
    if dir == 1 then
        while front.size() > 0:
            back.push(front.front())
            front.pop()
        end while
        dir = -1
    end if
    back.push(x)
end procedure

procedure pop_back(x):
    if dir == 1 then
        while front.size() > 0:
            back.push(front.front())
            front.pop()
        end while
        dir = -1
    end if
    set num to back.front()
    back.pop()
    return num
end procedure
```

4. First, the average case(average time complexity) is $O(n)$ because we have two case, one is operate in the same way ($O(1)$), two is operate in the different way ($O(n)$), so the average case is $\frac{O(n)+O(1)}{2} = O(n)$.

   Second, consider the worst case, we operate in the different direction each time. So the worst time complexity is $O(n)$

5. Same as the above, we have two choice with two different way and the average time complexity is $\frac{O(n)+O(1)}{2} = O(n)$.

And the worst time complexity is $O(n)$ when we operate on different direction each time.

6. Same as the above, we have two choice with two different way and the average time complexity is $\frac{O(n)+O(1)}{2} = O(n)$.

   And the worst time complexity is $O(n)$ when we operate on different direction each time.

7. Same as the above, we have two choice with two different way and the average time complexity is $\frac{O(n)+O(1)}{2} = O(n)$.

   And the worst time complexity is $O(n)$ when we operate on different direction each time.

8. In consecutive push operations, when the stack isn't full, the time complexity of each push is $O(1)$, but when the stack is full, the time complexity is $O(n)$. So in n times consecutive push, we use $O(1)$ n-1 times and $O(n)$ one time, so the amortized time complexity is $O(1)$.

Problem 3 - Array / Linked Lists

1. Step: used an array called **visited** with size n to record if we visit the point and **pos** to record the current position

   1. Initialize **visited** with 0
   2. start to run by setting **pos** = initial position
   3. Assign the value of **visited[pos]** = 1
   4. Check if **A[pos]** == **pos**, that is , we stop
   5. Iterate to the next position by set **pos** = **A[pos]**
   6. check if **visited[pos]** == 1, that is, it will run forever, if not, go step 3.

   Why: Because if a point is visited, we will visit again when following the steps from the visited node, so it creates a loop if we visit a position that we have visited.

   Time complexity: $O(n)$ is worst case if we find it stop or run forever in the last unrepetitive node.

   Space complexity: $O(n)$ because we use an extra array with size n to record.

2. Step: use an array called **steps** with size n to record the steps when we visit the position and **cur** record the current steps and **pos** to record the current position

   1. Initialize **steps** with zero
   2. start to run by setting **pos** = initial position, **cur** = 1
   3. Check if **steps[pos]** != 0, that is, the length is **cur** - **steps[pos]** and end iteration
   4. set **steps[pos]** = **cur**
   5. Add **cur** with one to represent the total step
   6. Iterate to the next position by set **pos** = **A[pos]** and go step 3

   Why: If the **steps[pos]** !=0 means we visited the position and the value is last step when we visit that position. So we can minus the current step with the last step(steps[pos]), then we can find the length of loop because the length of the loop is the interval of the same value appear in a loop.

   Time complexity:$n + 1 = O(n)$ is worst case if all nodes are in the loop.

   Space complexity: $n = O(n)$ because we use an extra array with size n to record.

3. Step: Use two variable called **min** to record the index of $M_{0,i}$ and **max** to record the index of $M_{j,n}$ and **val** to record the min value of $f(i,j)$ and **ans[2]** to record the i,j of min value

    1. Loop with variable **min** start with 1 while **min** < n
    2. Loop inside step 1 with variable **max** start with m -1 while 2**min** < 2**max** - m
    3. if **A[max]** - **A[min]** < **val** ,**val** = **A[max]**- **A[min]**, **ans** = {2**min**, **m** - 2**max**}
    4. Decrease **max** with 1 and go step 3
    5. Increase **min** with 1
    6. After iteration, **ans** is the minimized $f(i,j)$index

    Why: Because the function is strictly increasing, so max($M_{0,i}, M_{i,j}, M_{j,n}$) must appear in $M_{j,n}$, and so is min($M_{0,i}, M_{i,j}, M_{j,n}$) appear in $M_{0,i}$

    The minimized $f(i,j)$ must the smallest $M_{j,n}$ or largest $M_{0,i}$ or both.

    So we start by letting $M_{j,n}$ is the biggest and $M_{0,i}$ is the smallest, In iteration by decrease $M_{j,n}$ and increase $M_{0,i}$ we can guarantee that we consider the all possible value and record the minimal number with **val** and index by **ans**.

    i,j in $f(i,j)$ is 2**min** and 2**max** - m because $\frac{0+i}{2} = \min$, and $\frac{m+j}{2} = \max$, and i < j, so 2**min** < 2**max**- m.

    Time complexity: $O(n^2)$ because we do a double loop with index n

    Space complexity: $O(1)$ because we only use 5(3 + 2) extra variable.

4. Step: Use a variable **iter** to record the current node and **prev** to record the preview node and **Dec** to represent the current decreasing nodes, **move** to record the node you want to move and **move_prev** to record the prev node of move.

    1. Initialize **iter** with the header of L and **prev** with NIL and **Dec** to 0
    2. iterate in the list, if you find **iter.val** > **iter.next.val** and **Dec** == 0, record it with **move** and **move_prev** for prev node and increase **Dec** with 1 and continue the loop
    3. if the **Dec** == 1 and **iter.val** > **iter.next.val**, we can compare **move_prev** and **move** with **prev** and **iter** and record the small one in **move_prev** and **move** and increase **Dec** with 1
    4. If **move** exist and **Dec** == 2 and **iter.val** > **move.val** , set **prev_move.next** to **move.next** and Insert the node **move** in the back of **iter** then break iteration
    5. Set **prev** to **iter** and **iter** to **iter.next** and back to step 2.

    Why: Because there are only two decreasing node which means other nodes' value will less or equal than its next node, so we can move the smaller decreasing node to the right place(iter.val > move.val) to make sure it preserve the original order of other nodes' but still exist one decreasing node at the same time.

    We don't move the bigger decreasing node because it may be the node with biggest value, so there is no other way to insert because nothing bigger than it's val.

    Time complexity: $2n = O(n)$ we at most spend n to find the decreasing node and n to find the place to insert it to the right place.

    Space complexity: $O(1)$ because we only use 4 additional variable.