dsa HW1

problem 1

1.

Let the algorithm be $f'(n)$, c be some positive constant
$f'(n) = O(n^2)$ means
$f'(n)N \leq cn^2 \quad \forall n \geq n_0$
$f'(n) = O(n^3)$ belong to $O(n^2)$
if and only if
$\exists n_0, \exists c_1, c_2 > 0 \quad f'(n) \leq c_1 n^3 \leq c_2 n^2 \quad \forall n \geq n_0$
However
$c_1 n^3 \leq c_2 n^2 \quad \forall n \geq n_0$
$\implies \frac{c_1}{c_2} \lim_{n \to +\infty} n^3 \leq \lim_{n \to +\infty} n^2$
$\implies \frac{c_1}{c_2} \lim_{n \to +\infty} \frac{n^3}{n^2} \leq 1$
$\implies \infty \leq 1$
$\implies$ contradiction
Hence $f'(n) = O(n^3)$ doesn't belong to $O(n^2)$

2.

Big O notation represents the worst-case time complexity of an algorithm. The worst-case scenario is the situation in which the algorithm takes the longest time to solve the problem. In the case of binary search, the worst-case scenario is when the target number is not in the array. In this case, the worst case time complexity function is the length of the array, which is $O(n)$. However, the expected time complexity of binary search is $O(\log n)$. It may result in a "Time Limit Exceeded" error due to the excessive time taken to solve the problem.

3.

By defenition
$f(n) = \Theta(g(n))$ if
$\exists n_0, \exists c_1, c_2 > 0 \quad c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$
$\implies \lim_{n \to +\infty} c_1 g(n) \leq f(n) \leq c_2 g(n)$
$g(n) = n^2$

$\implies \lim_{n\to+\infty} c_1n^2 \leq f(n) \leq c_2n^2$
$\implies \lim_{n\to+\infty} c_1n^2 \leq f(n), \lim_{n\to+\infty} f(n) \leq c_2n^2$
$\implies \lim_{n\to+\infty} f(n) \geq c_1n^2, \lim_{n\to+\infty} f(n) \leq c_2n^2$
$\implies \lim_{n\to+\infty} \frac{f(n)}{n^2} \geq c_1, \lim_{n\to+\infty} \frac{f(n)}{n^2} \leq c_2$

Since both c1 and c2 are positive constants, it follows that $\lim_{n\to+\infty} \frac{f(n)}{n^2}$ exists and is a positive constant.

4.

By defenition
$f(n) = O(g(n))$ if
$\exists n_0, \exists c > 0 \quad f(n) \leq cg(n) \quad \forall n \geq n_0$
$f(n) = \lg n, g(n) = \sqrt{n}$
$\implies \exists n_0, \exists c > 0 \quad \lg n \leq c\sqrt{n} \quad \forall n \geq n_0$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \lg n \leq c\sqrt{n}$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{\lg n}{\sqrt{n}} \leq c$

By L'Hôpital's rule

$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{\frac{1}{n\ln 10}}{\frac{1}{2}n^{-1/2}} \leq c$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{1}{\frac{1}{2}n^{-1/2}n\ln 10} \leq c$
$\implies \exists c > 0 \quad 0 \leq c$

Hence we prove that $\lg n = O(\sqrt{n})$

5.

By defenition
$f(n) = O(g(n))$ if
$\exists n_0, \exists c > 0 \quad f(n) \leq cg(n) \quad \forall n \geq n_0$
$f(n) = \Sigma_{i=1}^{n} i^n, g(n) = n^n$
$\implies \exists n_0, \exists c > 0 \quad \Sigma_{i=1}^{n} i^n \leq cn^n \quad \forall n \geq n_0$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \Sigma_{i=1}^{n} i^n \leq cn^n$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{\Sigma_{i=1}^{n} i^n}{n^n} \leq c$

devide both side be n^n

$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{\frac{(1)^n}{n^n}+\frac{2^n}{n^n}...+\frac{(n-1)^n}{n^n}+1}{1} \leq c$
$\implies \exists c > 0 \quad \lim_{n\to+\infty} \frac{0+0+...+0+1}{1} \leq c$
$\implies \exists c > 0 \quad 1 \leq c$

Since there exists at least one constant c that c>0 and c>1,hence we prove that $f(\Sigma_{i=1}^{n}) = O(n^n)$

6.

$|\ln(f(n)) - \ln(g(n))| = O(1)$

$\implies \exists n_0, \exists c > 0 \quad \lg(f(n)) - \lg(g(n)) \leq$(not equal but lower equal) $c \quad \forall n \geq n_0$

$\implies \exists n_0, \exists c > 0 \quad \lg \frac{f(n)}{g(n)} \leq c \quad \forall n \geq n_0$

$\implies \exists n_0 \in R, \exists c > 0 \quad \frac{f(n)}{g(n)} \leq 2^c \quad \forall n \geq n_0$

$\implies \exists n_0, \exists c > 0 \quad f(n) \leq 2^c g(n) \quad \forall n \geq n_0$

Take $c' = 2^c$(not $\frac{1}{2^c}$)

$\implies \exists n_0, \exists c' > 0 \quad f(n) \leq c' g(n) \quad \forall n \geq n_0$

$\implies f(n) = O(g(n))$

problem 2

1.

```
function binary_search_miss(arr[0....n-1], integer l=0,integer r=n-1):
    m = floor((r+l)/2)
    if((m is 0 or n-1 and A[m] is 1 or n number) //
        or (A[m]+1 == A[m+1] and A[m]-1 == A[m-1] )):
        else if (A[m]-A[l] != m-l):
            return binary_search_miss(arr, l, m-1)
        else if (A[r]-A[m] != r-m):
            return binary_search_miss(arr, m+1, r)
    else:
        if (m is edge):
            if m = 0:
                return 0
            if m = n-1:
                return n
        else:
            if A[m]+1 != A[m+1]
                return A[m]+1
            if A[m]-1 != A[m-1]:
                return A[m]-1
            return m
    return NIL

function find_k_missing_value(arr[0....n-1], integer  k=2):
    M = [0...k] # missing list
    new_arr = arr
    while (k > 0):
        M[k] = binary_search_miss(arr, 0, len(new_arr)):
        intsert M[k] to new_arr
        k--
    return M
```

Time complexity $\approx 2 \log n$, which is similar to binary search.

2.

```
function is_paired(arr[0....n-1]):
    integer l, r = 0, 1
    integer s = null # integer to skip
    m = floor((r+l)/2)
    bool pair = True
    integer pair_cnt = 0
    while (j<=n-1):
        if(arr[i]*2 == arr[i]):
            i += 1
            j += 1
            pair_cnt += 1
        else if (arr[i]*2 <= arr[j]):
            i += 1
        else if (arr[i]*2 >= arr[j]):
            j += 1
        if (i == j):
            pair = False
            break
    if (pair_cnt != n/2):
        pair = False
        break
    return pair
```

Time complexity $O(n)$, while loop with array len n
Space complexity $O(1)$. only serveral fix spaces are required above

3.

```
function merge_link_list(A, B):
    start = null # start of merged list
    if (A==null):
        return B
    else if (B==null):
        return A
    else if (A->id <= B->id):
        start = A
    else:
        start = B
        B = A
        A = start
    # 333333333333
    while ( B != null):
        if(A->nxt == null or A->nxt->id > B->id):
            tmp = A->nxt
            A = B
            B = B->nxt
            A->nxt = tmp
        if(A == null):
            A = A->nxt
    return start
```

Time complexity $O(n)$, while loop with link list len n,
Space complexity $O(1)$ only one extra space start.

4.

```
function sort_link_list(A):
    start = A
    end = A # end node
    while (end->nxt != null):
        end = end->nxt
    while (A->nxt != null):
        if(A->nxt->id < 0):
            tmp = end->nxt
            end->nxt = A->nxt
            end->nxt->nxt = tmp
            A->nxt = A->nxt->nxt
    return start
```

Time complexity $O(n)$ while loop with link list len n,
Space complexity $O(1)$start end and temp three extra space.

problem 3

1.

5 3 4 * + 8 2 3 + * - 1 9 6 * + 5 / -
一般人類使用的方法會將要預先計算的地方用括弧框起來，在看到一個運算子的時候還會先確認後面有沒有優先及更高的運算子，複雜一點的式子可能會需要反覆確認多變，較沒效率

2.

527 × +634 − ×486 × +4/ − −
從左至右讀取，當看到運算子的時候，將前兩個數字做運算子運算，並放回陣列中，以此類推

3.

1. check if stack 0 empty or stack 1 empty
   stack 1 empty -> pop 3 from stack 0 and push it to stack 1

2. check if stack 0 empty or stack 1 empty
   both not empty -> compare top of stack 0 and top of stack 1
   find top of stack 1 greater -> check if top of stack 2 empty or greater than top of stack 1
   stack 2 empty -> pop 3 from stack 1 and push it to stack 2

3. check if stack 0 empty or stack 1 empty
   stack 1 empty -> pop 2 from stack 0 and push it to stack 1

4. check if stack 0 empty or stack 1 empty
   both not empty -> compare top of stack 0 and top of stack 1
   find top of stack 1 greater -> check if top of stack 2 empty or greater than top of stack 1
   top of stack 2 lower -> pop 2 from stack 1 and push it to stack 2

5. check if stack 0 empty or stack 1 empty
   stack 1 empty -> pop 1 from stack 0 and push it to stack 1

6. check if stack 0 empty or stack 1 empty
   both not empty -> compare top of stack 0 and top of stack 1
   find top of stack 0 greater -> check if top of stack 2 empty or greater than top of stack 0
   top of stack 2 lower than top of stack 0

Find that we can’t move all elements to stack 2 and sort them to the ascending order, for we can’t pop stack 2 and top of stack 2 lower than stack 0

4.

```
function sort_stack(stack0):
    while not stack0.isempty:
        if (not stack2.isempty) and stack0.top > stack2.top:
            return impossible
        if stack1.isempty or stack0.top >= stack1.top:
            stack1.push(stack0.top)
            stack0.pop()
        else:
            stack2.push(stack1.top)
            stack1.pop()
    while not stack2.isempty:
        stack2.push(stack1.top)
        stack1.pop()
    return stack2
```

5.

先逐一計算腳踏車的間距，由左至右，逐一push到一個queue中
第一台車停時無條件取第一台車與第二台車之中，

將上述距離除二後存在queue
第二台車來時比較第二台車與第三台車間距與上面站存的第一台車與第二台車距離除二
取第二台車與第三台車間距，將之除二後存到queue,
第三台車來時比較第三台車與第四台車間距與queue最前面得職
發現queue head值較大，得知她得的間距為queue head /2 = 2.5
在push
得到[5, 2]

6.

```
function sort_stack(Hall, m):
    Hall_queue = compute distance and push to queue
    new_queue = empty queue
    length = 0
    for i := 1 to m:
        if (new_queue.head == Nil or Hall_queue.head > new_queue.head)
            length =  Hall_queue.head/2
            new_queue.push(Hall_queue.head/2)
            new_queue.push(Hall_queue.head/2)
            Hall_queue.pop_head
        else
            length =  new_queue.head/2
            new_queue.push(new_queue.head/2)
            new_queue.push(new_queue.head/2)
            new_queue.pop_head

    return length
```

time complexity = O(m+n) (for loop+initial queue)
space complexity = O(m+n) (length N for Hall_queue + 2m length for new queue in worst case)