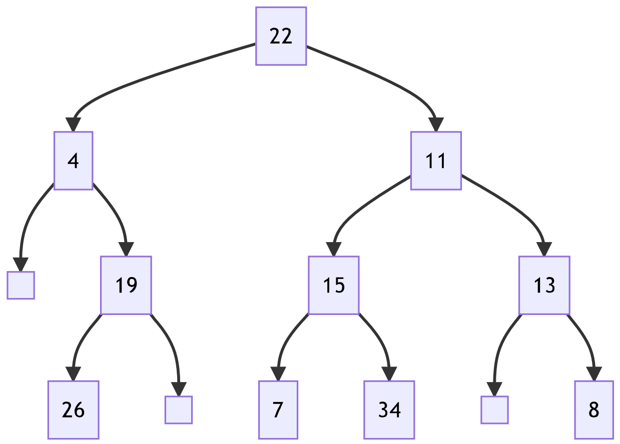
# dsa HW2

## 1

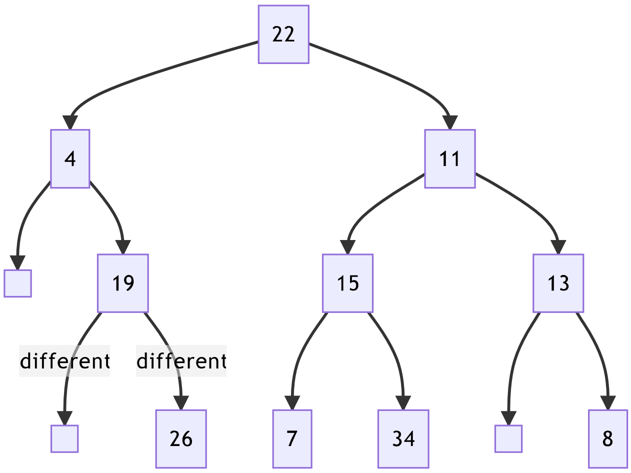
## 2

可以
概念是從preorder 按照順序放入節點，透過inorder 判斷位置，每當inorder list節點以左的元素都放入成功，即可拋棄該節點inorder list以左的資訊，新的節點放在該節點的右子節點
步驟是首先看preorder的第一個元素得知為22
得知tree head是22
有了第一個點後，看inorder 22 的位置得知在其左邊的node都在22 左子樹
接著將4放到22左子樹，看4在inorder 之中的位置，4以前的值是左子樹，4以右到父節點22之前是右子樹, 發現4沒有左子樹，可以得知19是4的右節點，接下來確認19只有左子樹，26放到19左子樹
接下來以此類推，透過觀察preorder節點在inorder list 中父節點的右邊還是左邊決定是放在左子節點還是右子節點



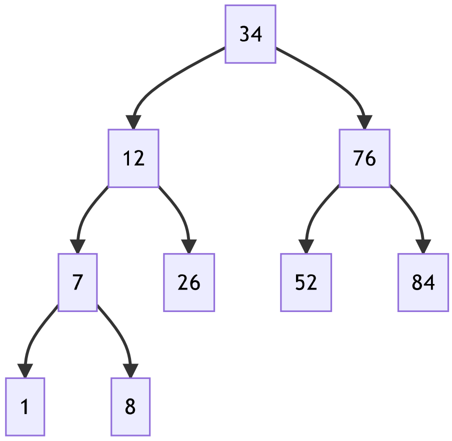

## 3

不行，會使很多左子樹跟右子樹位置不明確，以圖證明，第二題的tree 跟第三題的tree在微調node 26 位置後都符合題三的treversal result，因此可能有多種正確結果如果只根據preorder 跟 postorder




## 4

[1, 7, 8, 12, 26, 34, 52, 76, 84]




## 5

```
function construct_shortest_BST(arr, s, e)

    if s > e:
        return NIL
    mid := ceil((e + s)/ 2 )
    node := new node address
    node->num := arr[mid]
    node->left := construct_shortest_BST(arr, s, mid-1])
    node->right := construct_shortest_BST(arr, mid+1, e])
    return node
```

## 6

time complexity = pigean_search line 2 for loop N * pigean_search line 4 for loop M * pigean_search line 6 for loop P
O(N * M * P)
extra space complexity = pigean_store line 1 two

dimension array M*P array + pigean_search line 1 one dimension array N*
*O(N + M*P)

### 7

time complexity = Spotteddove_search line 2 for loop N * Spotteddove_search line 6 for loop P
O(N * P)
extra space complexity = extra space complexity for storing bugs root is min(26, P)*26 + Spotteddove_search line 1 one dimension array N
O(N)

### 8

I’ll choose Spotteddove method, it’s obvious that it’s time complexity and extra-space complexity spend less than previous one.
Although it take more extra-space when M is small.However, I prefer wasting some space when the task is simple rather than spending enormous time and space when the task is complicate.

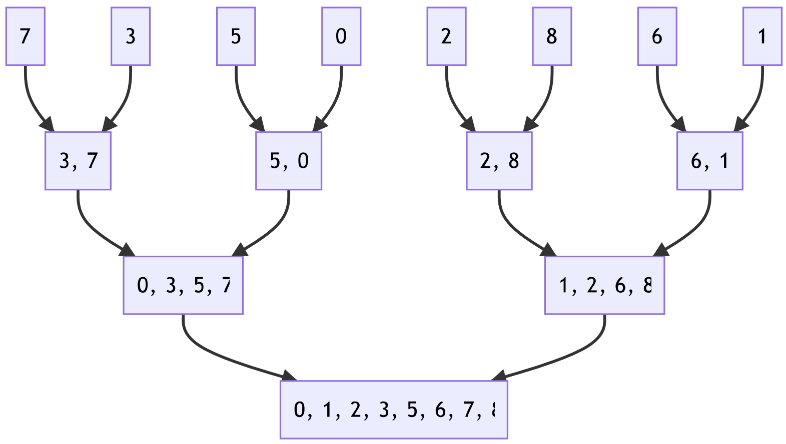
## 2

### 1

```
funtion Button-up_merge_sort(arr, n)
    n := size of arr
    for i:=1; i<n; i*=2
        for j:=1; j<=n; j+=2i

            mid := min(n, j+i)
            end := min(n, j+2i)
            s1, s2, e1, e2 := j, mid, mid, end
            tmp := array with size [end - j]
            idx := 1
            while(s1 < e1 and s2 < e2)
                if arr[s1] <= arr[s2]:
                    tmp[idx] := arr[s1]
                    s1++
                else
                    tmp[idx] := arr[s2]
                    s2++
            while(s2 < e2)
                tmp[idx] := arr[s2]
                s2++
            while(s1 < e1)
                tmp[idx] := arr[s1]
                s1++
            for k := 0 to end - j - 1
                arr[j+k] := tmp[k + 1]
    return arr

```



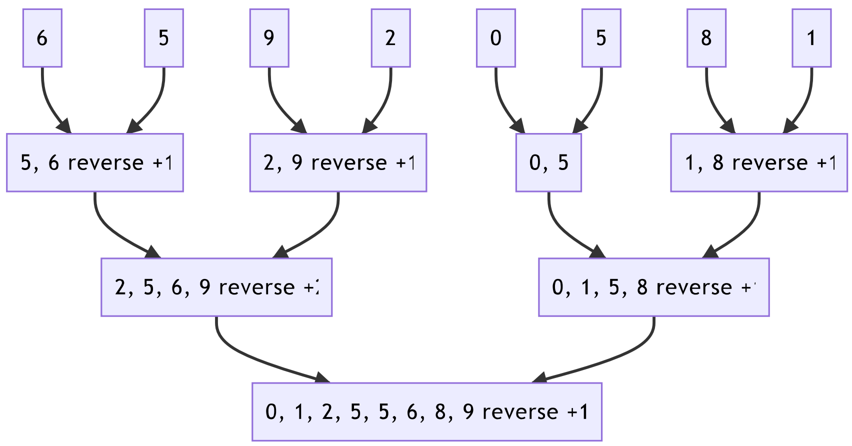

## 2

```
funtion count_number_of_reversion(arr, n)
    n := size of arr
    cnt := number of reversion
    for i:=1; i<n; i*=2
        for j:=1; j<=n; j+=2i

            mid := min(n, j+i)
            end := min(n, j+2i)
            s1, s2, e1, e2 := j, mid, mid, end
            tmp := array with size [end - j]
            idx := 1
            while(s1 < e1 and s2 < e2)
                if arr[s1] <= arr[s2]:
                    tmp[idx] := arr[s1]
                    s1++
                else
                    tmp[idx] := arr[s2]
                    s2++
                    cnt += e1 - s1
            while(s2 < e2)
                tmp[idx] := arr[s2]
                s2++
            while(s1 < e1)
                tmp[idx] := arr[s1]
                s1++
            for k := 0 to end - j - 1
                arr[j+k] := tmp[k + 1]
    return cnt

```

在merger sort過程，每發成一次先加入後面陣列的元素，Reversion cnt += 前面陣列剩餘的元素



total reversion = 17

## 3

```
1  funtion count_number_of_reversion(arr, n)
2      cnt := number of reversion
3      for i:=1; i<n; i*=2
4          for j:=1; j<=n; j+=2i
5              mid := min(n, j+i)
6              end := min(n, j+2i)
7              s1, s2, e1, e2 := j, mid, mid, end
8              tmp := array with size [end - j]
9              idx := 1
10             while(s1 < e1 and s2 < e2)
11                 if arr[s1] <= arr[s2]:
12                     tmp[idx] := arr[s1]
13                     s1++
14                 else
15                     tmp[idx] := arr[s2]
16                     s2++
17                     cnt += e1 - s1
18             while(s2 < e2)
19                 tmp[idx] := arr[s2]
20                 s2++
21             while(s1 < e1)
22                 tmp[idx] := arr[s1]
23                 s1++
24             for k := 0 to end - j - 1
25                 arr[j+k] := tmp[k + 1]
26     return cnt
27
```

time complexity = line 4 for loop [$log_2N$] * line 5 for loop [n/2i] * while loop from line 10 to line 23 [2i]
O($Nlog_2N$)
extra space complexity = line 8 one dimension array with at most size N
O(N)

## 4

```
funtion count_number_of_candidate(arr, n)
    arr := Button-up_merge_sort(arr, n) // overload operator that arr[i] <= arr[j]
    KD := K/D ratio list of arf
    cnt := (n - 1)n/2 // number of candidate
    cnt -= count_number_of_reversion(KD, n)
    return cnt
```

time complexity = Button-up_merge_sort O($Nlog_2N$) + count_number_of_reversion O($Nlog_2N$)
O($Nlog_2N$)
extra space complexity = Button-up_merge_sort O($N$) + count_number_of_reversion O($N$)
O($N$)
O(N)

### 5

```
funtion binary_search_k-th(arr1, arr2, n, k)
    s1, e1, s2, e2 = 1, n, 1, n
    while 2n <= k
        m1, m2 = floor(s1 + e1)/2 , floor(s2 + e2)/2
        if (arr[m1] + arr[m2] > k)
            if m1 > m2
                e1 = m1 - 1
            else
                e2 = m2 - 1
        else if (arr[m1] + arr[m2] < k)
            if m1 < m2
                s1 = m1 + 1
            else
                s2 = m2 + 1
        else
            return max(m1, m2)
    return cnt

```

time complexity = while loop $2log_2N$
O($log_2N$)
extra space complexity = s1, m1, e1, s2, m2, e2 ($6$)
O($1$)

### 6

time complexity of partition fuction is $2N$. However, pi that partition return is always be lastest - 1.
Hence, NeonSort will call recursive fuction with parameter (match, oldest, lastest - 1), and (match, lastest, lastest), only NEONSort with parameter (match, oldest, lastest - 1) will continuely call partition fuction.
Considering all the above, the time complexity of NeonSort is 2N*(N)/2 = N^2
O($N^2$)

## 3

### 1

result = [74, 16, 10, 11, 6, 7, 4, 6, 8, 3]

### 2

result = [74, 16, 10, 11, 6, 7, 8, 4, 6, 3]

## 3

21964800
用程式把每種可能計算，同時進行剪枝減少進行時間，array由前至後插入值0-13，如果插入值>於其index/2的值或該值已被插入過，中斷for迴圈，否則填入新值，只有一路填完15格的case納入計算，max_heap可能性應等於min_heap可能性

```
cnt = 0
def count (arr, deph):
    cnt = 0
    for i in range(14):
        past = arr.copy()
        if i in past[0:deph]: continue
        if deph == 15 :
            if i > past[deph//2]:
                return 0
            else:
                past.append(i)
                return 1
        if i > past[deph//2]:
            continue
        past.append(i)
        cnt += count(past, deph+1)
    return cnt
print(count([-1, 14], 2))
```

## 4

min heap
for 1-15 計算每個點大於其後面的值的數量(inverse)
第一個ㄘㄥ無庸置疑 沒有inverse
[1]
第二個值
最大可以是9，小於整個右子樹中的值，有2^3-1個inverse
而另一個便只能是2-noinverse
總共7
[1, 9, 2]
第三層
可以是[1, 9, 2, 13, 10, 6, 3]
inverse 數分別是 小於3到1個子樹 = 3(2^2-1) + 2(2^2-1) + 1(2^2-1)
總共18
最後一層
可以是[1, 9, 2, 13, 10, 6, 3, 15, 14, 12, 11, 9, 8, 5, 4]
以此類推
7*(2*1-1*) + *6*(2*1-1) + ... + 1(2*1-1)
總共 7
加總為32

## 5

```
1 function minimum_cost_fusing(arr)
2     cost := 0
3     build arr to min_heap            O(nlogn)
4     for i := 1; i < arr.length; i++: O(n)
5         a := REMOVE-SMALLEST(arr)    O(1)
6         b := REMOVE-SMALLEST(arr)    O(1)
7         cost += a + b
8         arr.insert(a + b)            O(logn)
9
```

the only extra-space the algoruthm is cost which is O(1)
and the time complexity is as mention in pseudo code
is = nlogn(build min heap) + n(for loop in line 4) *(
logn(heap.insert) + 1(heap.remove smallest) )
= O(nlogn)

## 6

```
funtion Button-up_merge_sort(array_list, M, N, new)

    for i:=N; i<MN; i*=2
        for j:=1; j<=MN; j+=2i

            mid := min(MN, j+i)
            end := min(MN, j+2i)
            s1, s2, e1, e2 := j, mid, mid, end
            idx := 1
            while(s1 < e1 and s2 < e2)
                if array_list[s1/M+1][s1%M] <= array_list[s2/M+1][s2%M]:
                    new[idx] := array_list[s1/M+1][s1%M]
                    s1++
                else
                    new[idx] := array_list[s2/M+1][s2%M]
                    s2++
            while(s2 < e2)
                new[idx] := array_list[s2/M+1][s2%M]
                s2++
            while(s1 < e1)
                new[idx] := array_list[s1/M+1][s1%M]
                s1++
            for k := 0 to end - j - 1
                array_list[(j+k)/M+1][(j+k)%M] := new[k + 1]
    for i:=0; i<M; i+=1
        for j:=1; j<=N; j+=1:
            new[i*M+j] = array_list[i][j]

    return new

```

it is equavalent to botton which have been process logN round
Hence BigO = O(mn (log (mn) - log (n))) = O(mn (log (m)))
and extra space = O(1) because new is not included in the extra space