Precourse 2

Problem 1

1. Min Heap

Min heap is a complete binary tree, that for any given node C and its parent P, the key of P is always less or equal key of C.

A heap data structure can be represented as an array in which, for each node `i`, its left and right child will be at `2*i +1` and `2*i+2` position.

In this exercise, we need to implement three functions of heap data structure to ensure the heap's property after each operation.

push

The `push()` function is used to push a new element to a heap while still ensuring heap's property. The procedure of `push()` is described as follow:

```
1. append new value to the end of array
2. start at the end of the array, compare current node with its parents
3. if the parent is greater than current node then swap it
4. repeat step #2 and #3 until the heap property is satisfied or the root
node has been swapped.
```

```
5     def push(self, value):
6         # TODO : FILL IN HERE
7         try:
8             self.heap.append(value)
9             index = len(self.heap ) -1
10            parent = (index - 1)//2
11            while self.heap[parent] > self.heap[index] and index > 0:
12                tmp = self.heap[index]
13                self.heap[index] = self.heap[parent]
14                self.heap[parent] = tmp
15
16                index = parent
17                parent = (index-1)//2
18        except Exception as e:      You, 2 hours ago • add handle exception and
19            print(e)
20
```

With this procedure, we start from the end of array and the index will be divided by 2 each step, so the time complexity of algorithm is O(log n) with n is the size of array.

_heapify() a node

This is an common function which will be used in `pop()` and `heapify` function. This function will compare the current node with its children and swap until it less than its children.

```
20        You, 3 hours ago • add handle exception and fix bug of probl
21    def _heapify(self,index):
22        temp = index
23        left = 2*index +1
24        right = 2*index+2
25        n = len(self.heap)
26        if n ==0 : return
27
28
29        if left < n and self.heap[left] < self.heap[temp]:
30            temp = left
31        if right < n and self.heap[right] < self.heap[temp]:
32            temp = right
33
34        if temp != index:
35            temp_value = self.heap[temp]
36            self.heap[temp] = self.heap[index]
37            self.heap[index] = temp_value
38            self._heapify(temp)
```

The time complexity of this function is O(log n) with n is the size of heap, because the index will be multiplied by 2 at each step until it out of array.

pop

The `pop()` is used to delete the smallest element of an array. The procedure to pop is described as follow:

```
1. get the smallest value of the heap
2. replace the first element by the last element of the heap
3. remove the last element
4. do the _heapify() on the first element
```

```
41      def pop(self):
42          try:
43          # TODO : FILL IN HERE
44              if len(self.heap) == 0:
45                  raise Exception("Heap is empty")
46              value = self.heap[0]
47              self.heap[0] = self.heap[len(self.heap)-1]
48              self.heap.pop()
49              self._heapify(0)
50
51              return value
52          except Exception as e:
53              print(e)
54
```

The time complexity is equal to the _heapify() - O(log n).

heapify

This function is used to transform an array to heap structure. This task can be done by apply `_heapify()` on all none-leaf node in bottom-up order.

```
54
55      def heapify(self):
56          # TODO : FILL IN HERE
57
58          try:
59              for i in reversed(range(0, len(self.heap) //2)):
60                  self._heapify(i)
61
62          except Exception as e:
63              print(e)
64
```

In this function, the `_heapify()` is called n//2 times. So the total time complexity upper bound of heapify is O(n log(n))

Experiment result

```
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem1 % python heap.py
Min heap :  [2, 3, 6, 5, 4, 20, 10, 15, 9, 8]
--------------------
compare result with heapq standard python library
heapify function:
Result of my algorithm: [2, 3, 6, 5, 4, 20, 8, 9, 15, 10]
Result of heapq:        [2, 3, 6, 5, 4, 20, 8, 9, 15, 10]
--------------------
pop function:
Result of my algorithm: [3, 4, 6, 5, 10, 20, 8, 9, 15]
Result of heapq:        [3, 4, 6, 5, 10, 20, 8, 9, 15]
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem1 %
```

In this problem, it ouput to terminal the result of push operation.

Then I compare results of pop and heapify function with a python standard library *heapq*. Both of outputs are the same.

2. Dijkstra with min heap

Firstly, we will define some variables:

- distances[i] is shortest path from source node to node *i*
- mark[i] let us know node *i* is visited or not.
- graph[i][j] is the edge of graph, the distance from node *i* to node *j*.

The dijkstra algorithm can be describe as follow:

1. Mark all node are unvisited and the distances[i] is 0 if *i* is source node else *infinity*.
2. For current node *i*, find all of its unvisited neighbors. For each unvisited neighbor *j* if distances[j] > distances[i] + graph[i][j] then update distances[j] = distances[i] + graph[i][j].
3. When we are done consider all unvisited neighbors, set mark[i] = True so current node *i* is marked visited. This node will never be considered again.
4. If all node of graph is visited the algorithm has finished.
5. Otherwise, select in the unvisited nodes that has the smallest distance from source and set it as current node, then repeat from step #2.

The time complexity of above algorithm is O(V^2) with V is number of vertex or node of graph. This is because we need to do 2 loops, the first loop is find the unvisited node that has smallest distance from source, then the second loop, we need to for all of its neighbors.

The step #5 can be optimized using priority queue - min heap instead of using a for loop to find unvisited node has smallest distance from source. The dijkstra algorithm with min-heap can be describe as follow:

1. Mark all node are unvisited and the distances[i] is 0 if *i* is source node else *infinity*.
2. Create a min heap, each element in heap is a set of 2 value (vertex, distance), vertex is name of node, distance is the distance from source to that node. The distance will be used to compare in heap structure.
3. Push the source element to heap with distance = 0.
4. pop() the heap until get the unvisited node, this step can result in unvisited node has smallest distance from source node. This node is set as current node *i*.

5. for all neighbor *j* of current node *i*, if distances[j] > distances[i] + graph[i][j] then update distances[j] = distances[i] + graph[i][j] and push (j, distance [j]) even if *j* is already in heap.
6. repeat from step 4 until the heap is empty, the algorithm is finished.

To use the previous heap implementation, need to create a node data structure that contains vertex and distance attributes and override compare method. The Node class is implemented as follow:

```python
class Node(object):
    def __init__(self, vertex, distance) -> None:
        self.vertex = vertex
        self.distance = distance

    def __str__(self) -> str:
        return f"{self.vertex} - {self.distance}"
    def __lt__(self, other):
        return self.distance < other.distance

    def __le__(self, other):
        return self.distance <= other.distance

    def __gt__(self, other):
        return self.distance > other.distance

    def __ge__(self, other):
        return self.distance >= other.distance

    def __eq__(self, other):
        return self.distance == other.distance

    def __ne__(self, other):
        return self.distance != other.distance
```

With this Node implementation, when compare each element in heap, it will compare the distance.

The python implementation of Dijkstra:

```python
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    priority_queue = heap.MinHeap()
    mark = {node : 0 for node in graph}
    # TODO : FILL IN HERE
    distances[start] = 0
    priority_queue.push(Node(start,0))
    while(len(priority_queue.heap) > 0):
        current = priority_queue.heap[0]
        priority_queue.pop()
        if mark[current.vertex] == 1: continue
        mark[current.vertex] = 1
        for v, w in graph[current.vertex].items():

            if distances[v] > distances[current.vertex] + w:
                distances[v] = distances[current.vertex] + w
                priority_queue.push(Node(v,distances[v]))

    return distances

```

The time complexity of push and pop element in heap is O(log(V)), and we need to do this process E times with E is number of edge in graph, so the time comlexity of this algorithm is O(E log(V)).

(*) The maximum number of edge in a graph is V^2, so the min heap is only better if the graph is sparse.

The run result is shortest path from source to every node

```
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem1 % python dijkstra.py
Start Node: A
Shortest distances: {'A': 0, 'B': 5, 'C': 3, 'D': 6, 'E': 9, 'F': 11}
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem1 %
```

Problem 2

The idea of this problem is, the index of column in grid is the index of node in tree when apply in order traversal. To solve this problem we will travel the tree inorder then save the level and column number for each node, and use that infomation to calculate the widest level.

Let's define some data structures:

Node

A node structure has the following attribute:

- left: point to left children
- right: point to right children
- value: the name of node
- parent: point to parent node, the parent attribute is used to find the root node when in hard test, we don't know what is root.

Levels

A python dictionary map the level and list of index column in that level.

levels = map< Level , List< ColumnIndex> >

Algorithm

To make algorithm more generic, we need to find the root node in case the root is not the node 1. The algorithm has 3 parts: find root node, do in-order traversal, find the widest_level and max_width.

1. Find root node

This part can start at any node *i*, if parent[i] is None -> *i* is root node, else we continue do the same procedure with parent[i]. Python implementation is described as follow:

```
33      def find_root(node):
34          if node.parent is None:
35              return node.value
36          else:
37              return find_root(node.parent)
```

The worst case of this algorithm is O(n) with n is number of node in tree.

2. In-order traversal

This part, we need to do travel the left child, the current node and the last is right child. At each step we need to append the column index to levels map. The python implementation is as followed

```python
def travel(node, level=1, col = col):
    global Numnode
    if node is None: return
    # print(node.value)
    # node.level = level
    travel(node.left, level=level+1, col=col)

    # node.column = col["value"]
    a = levels.get(level,[])
    a.append(col["value"] )
    levels[level] = a
    col["value"] +=1
    travel(node.right, level=level+1, col=col)
```

At each node, we do a constant number of operation, and we need to travel all nodes, so the time complexity of this algorithm is O(n) with n is total number of node in tree.

3. Find result

Through the tree traversal process, a list column index of a level is in incresing number automatically, so we don't need to order this list again. The process is:

- set max_width = 1, widest_level = 1
- for each level *i* in levels:
  - for each column *col* in levels[i]:
    - if max_width < dist(col) : max_width = dist(col); widest_level = i; with dist(col) is the distance of 2 consecutive column start at *col*

```python
60      for level, info in levels.items():
61          for i in range(1,len(info)):
62              if max_width < info[i] - info[i-1] +1:
63                  max_width = info[i] - info[i-1] + 1
64                  widest_level = level
```

For this step, every node of graph corresponding to a col will be considered once, so the time comlexity is O(n).

So the total complexity of algorithm is O(n).

```
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem2 % python main.py
(4, 5516)
total time run 1000 times algorithm: 3.8678045839769766 s
(base) thuannguyenhoang@Thuans-MacBook-Pro Problem2 %
```

The experiment result run on hard test (the root node is not 1) shows that the algorithm runs 1000 times in 3.86s -> with a tree has 10000 nodes, my algorithm can run in average 4ms.

Problem 3

Optimize inference time

The key to optimize inference time is broadcasting. The original code using for loop and consider each image and explixit broadcast mean to a matrix has the same shape with image.

Here is my solution:

- The array A has shape (n,h,w) -> reshape to (n, h*w)
- calculate mean = np.mean(A, axis =1, keep dims = True), this operation will calculate the mean of each image in *n* images, while keeping the last axis. For that reason, the *mean* array has shape (n,1)
- call np.subtract(A,mean, out=A), this operation will internal broadcasting the axis 1 of mean to axis 1 of A, and then save to result to A. This operation corresponding to subtract each image with its mean value.
- The last operation is do matrix multiply which is the same with original code

Optimize memory

The key to optimize memory is the order of element in array in the memory level. The array *A* is an fortran array - index the elements in column-major, while the default order of numpy is always "C" style - rows major. For this reason, when call A.reshape, it will allocate new memory with "C" order for array A.

To optimize this, just add the option order="F" for reshape function of numpy.

```
81      n,h,w = A.shape
82
83      A = np.reshape(A,(n,h*w), order="F")
84
85      mean = np.mean(A,axis=(1),keepdims=True)
86      A-=mean
87      # np.subtract(A,mean, out=A, order="F")
88      # covs = A@A.T
89      covs = np.matmul(A,A.T)
90      # covs = np.cov(A)*n
91
```

And this code can pass 2 tests

```
================ SNAPSHOT =================
1 memory blocks: 703.1 KiB
    norm_A = np.array(norm_A)
1 memory blocks: 78.1 KiB
    covs = np.matmul(norm_A, norm_A.T)
1 memory blocks: 7.0 KiB
    norm_a = a - mean_a
1 memory blocks: 7.0 KiB
    a = a.flatten()
1 memory blocks: 0.0 KiB
    array = np.array(array, copy=False, subok=subok)
Total Mem: 795.3203125 KiB

================ SNAPSHOT =================
1 memory blocks: 78.1 KiB
    covs = A@A.T
1 memory blocks: 0.8 KiB
    ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
Total Mem: 78.90625 KiB
Success!!
```

Problem 4

In this problem, in order to improve inference time, I need to do filter on all images at the same time using numpy broadcasting.

```
28      images = np.pad(images,((0,0),(padding,padding),(padding,padding)))/255.
29      kernel_H, kernel_W = kernel.shape
30      n,H,W = images.shape
31      out_h = (H-kernel_H)//stride +1
32      out_w = (W-kernel_W) // stride + 1
33      # print(out_h,out_w)
34      output_images = np.zeros((n,out_h,out_w))
35      # kernel = np.array([kernel])
36      for i in range(out_h):
37          for j in range(out_w):
38              # print(i,j)
39              ith = i*stride
40              jth = j*stride
41              #output_images[:,i,j] = np.einsum("jk,ijk->i",kernel,images[:,ith:ith+kernel_H, jth :jth+ker
42              output_images[:,i,j] = \
43                  np.sum(np.multiply(images[:,i*stride:i*stride+kernel_H, \
44                                            j*stride :j*stride+kernel_W],kernel),axis = (1,2))      You,
45
46
```

- Firstly, we need to pad the image with zero padding and normalize image by devide by 255
- Then, calculate the output shape base on original shape, kernel size and stride.
- for each pixel in output array, apply filter on every image at the same time.

This implementation cost **0.78s** for 1 batch 32 images.

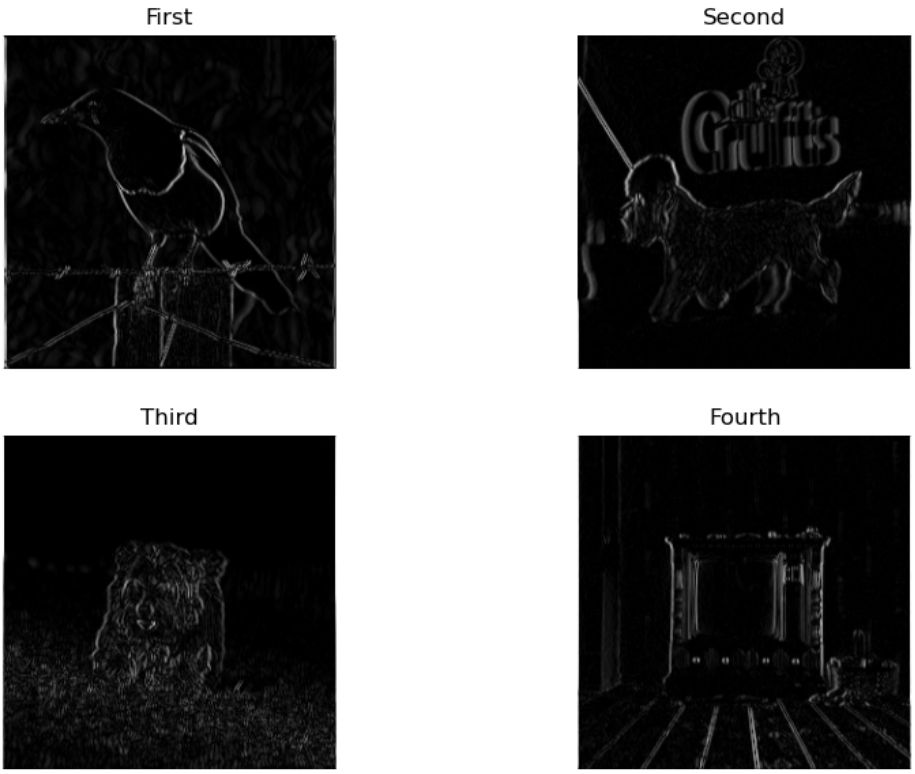
We still can optimize this time by apply numpy einsum

```
output_images[:,i,j] = np.einsum("jk,ijk->i",\
                                 kernel,images[:,ith:ith+kernel_H, \
                                               jth :jth+kernel_W])
```

The einsum operation "jk,ijk->i" will define the sum and multiply operation. "jk,ijk->" mean the kernel has 2 dimensions "jk", the image piece has 3 dimensions "ijk", so "jk,ijk" means the kernel "jk" will be broadcasting to all axis i of image piece and do point-wise multiply. "->i" means the result "ijk" will be sum over the "jk" axis to produce only "i" axis.

The einsum can make the process faster, take **0.56s** for batch 32 images.

When apply only x-filter, the algorithm can recognize the edge in column directions.

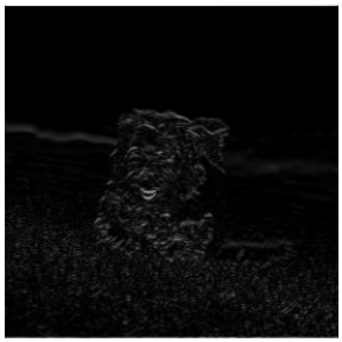


While y-filter can make algorithm recognize edge in row directions.

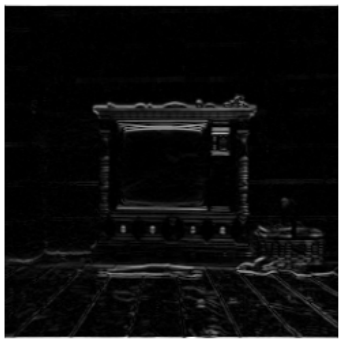
First

Second

Third



Fourth



When combine both result it can show the edge of object

First

Second

Third

Fourth