Python Programming

Problem 1

MinHeap

Generally, the first thing to do is to implement the basic operation of modifying the min heap, the rotate (or the node swap) operation.

```
left_node = 2 * node
right_node = 2 * node + 1

min_node = node
if (
    left_node < len(self.heap)
    and self.heap[min_node - 1] > self.heap[left_node - 1]
):
    min_node = left_node
if (
    right_node < len(self.heap)
    and self.heap[min_node - 1] > self.heap[right_node - 1]
):
    min_node = right_node

if min_node != node:
    self.heap[min_node - 1], self.heap[node - 1] = (
        self.heap[node - 1],
        self.heap[min_node - 1],
    )
    self.rotate(min_node)
```

After that, other operations just get more easier, the push operation is just append that value to the last of the heap and rotate from that node all the way to the root.

```
self.heap.append(value)
node = len(self.heap)
while node > 1:
    parent_node = node // 2
    if self.heap[parent_node - 1] > self.heap[node - 1]:
        self.heap[parent_node - 1], self.heap[node - 1] = (
            self.heap[node - 1],
            self.heap[parent_node - 1],
        )
        node = parent_node
    else:
        break
```

The pop operation, we replace the last node with the root, we remove the root and again rotate down the last level from the root.

```
if len(self.heap) == 0:
      raise IndexError("MinHeap is empty")

top_value = self.heap[0]

self.heap[0] = self.heap[len(self.heap) - 1]
self.heap.pop()

if len(self.heap) > 0:
    self.rotate(1)

return top_value
```

The heapify operation, we do rotate to all the non-leaf nodes.

```
min_heap = heap.MinHeap()
distances = {node: float("inf") for node in graph}

distances[start] = 0
```

```
min_heap.push((start, 0))

while not min_heap.is_empty():
    top_heap = min_heap.pop()
    u, distance_u = top_heap
    if distances[u] != distance_u:
        continue
    for v in graph[u]:
        value_v = distance_u + graph[u][v]
        if distances[v] > value_v:
            distances[v] = value_v
            min_heap.push((v, value_v))

return distances
```

The execution result:

```
➜  getp-precourse git:(main) cd Precourse2/Problem1
➜  Problem1 git:(main) python3 heap.py
Min heap :  [2, 3, 6, 5, 4, 20, 10, 15, 9, 8]
```

Dijkstra

Just go from the min heap(initially only the start node with distance 0 there), iterate through its top value's edges, each edge update the distance wherever it's possible and put the update back to the min heap.

```
min_heap = heap.MinHeap()
distances = {node: float("inf") for node in graph}

distances[start] = 0
min_heap.push((start, 0))

while not min_heap.is_empty():
    top_heap = min_heap.pop()
    u, distance_u = top_heap
```

```
    if distances[u] != distance_u:
        continue
    for v in graph[u]:
        value_v = distance_u + graph[u][v]
        if distances[v] > value_v:
            distances[v] = value_v
            min_heap.push((v, value_v))

return distances
```

Execution result

```
➜  Problem1 git:(main) python3 dijkstra.py
Start Node: A
Shortest distances: {'A': 0, 'B': 5, 'C': 3, 'D': 6, 'E': 9, 'F
```

Problem 2

My idea is starting to determine all nodes's level at every level by DFS.

```
num_node = 1
root_col = margin + 1

if root in lefts:
    left_num_node = build_level_cols(
        lefts[root], level + 1, margin, lefts, rights, level_col
    )
    root_col = left_num_node + margin + 1
    num_node += left_num_node

if level not in level_cols:
    level_cols[level] = []

level_cols[level].append(root_col)
```

```
if root in rights:
    right_num_node = build_level_cols(
        rights[root], level + 1, root_col, lefts, rights, level_
    )
    num_node += right_num_node

return num_node
```

After that, we simply interate through all the nodes at each level and find the answer

```
level_cols = {}
build_level_cols(1, 1, 0, lefts, rights, level_cols)

widest_level, max_width = (0, 0)

for level in level_cols:
    level_max_width = 0
    if len(level_cols[level]) == 1:
        level_max_width = 1
    else:
        for index in range(1, len(level_cols[level])):
            level_max_width = max(
                level_max_width,
                level_cols[level][index] - level_cols[level][ind
            )
    if max_width < level_max_width:
        max_width = level_max_width
        widest_level = level

return widest_level, max_width
```

Execution result

```
➜  Problem2 git:(main) python3 main.py
➜  Problem2 git:(main) cat output.txt
```

```
2 13
```

Problem 3

The key problem here is **keep all result back to A**, so we do not need to allocate any more memory for storing result. There are two things to keep in mind:

- the operation mean and subtraction needs to store back to A (in-place subtraction)

```
mean_a = np.mean(A, axis=(1, 2), keepdims=True)
A -= mean_a
```

- reshape **in-place** 3D matrix `A` to 2D matrix to do matrix multiplication

```
n, w, h = A.shape
reshape_a = A.reshape((n, w * h), order="F")
covs = np.matmul(reshape_a, reshape_a.T)
```

Execution result

```
➜  Problem3 git:(main) python3 main.py

================ 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 = np.matmul(reshape_a, reshape_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

My idea of this problem is to implement a **general** convolution implementation in all case whatever the padding and stride value is.

So first the first is adding zeros value padding to image and iterate though that big image, take median to build the output image.

```
n, image_h, image_w = images.shape
kernel_h, kernel_w = kernel.shape

image_pad_h = (stride * (image_h - 1) + kernel_h - image_h) // 2
image_pad_w = (stride * (image_w - 1) + kernel_w - image_w) // 2

output_h = (image_h + 2 * image_pad_h - kernel_h) // stride + 1
output_w = (image_w + 2 * image_pad_w - kernel_w) // stride + 1

output_images = np.zeros((n, output_h, output_w))

for index, image in enumerate(images):
    padded_image = np.pad(
        image,
        ((image_pad_h, image_pad_h), (image_pad_w, image_pad_w)),
        mode="constant",
        constant_values=0,
    )
    output_image = np.zeros((output_h, output_w))
    for i in range(0, output_h * stride, stride):
        for j in range(0, output_w * stride, stride):
```

```
            output_image[i // stride, j // stride] = np.mean(
                np.multiply(
                    padded_image[i : i + kernel_h, j : j + kerne
                )
            )

    output_images[index] = output_image

return output_images
```

Execution result

```
➜  Problem4 git:(main) python3 main.py
```

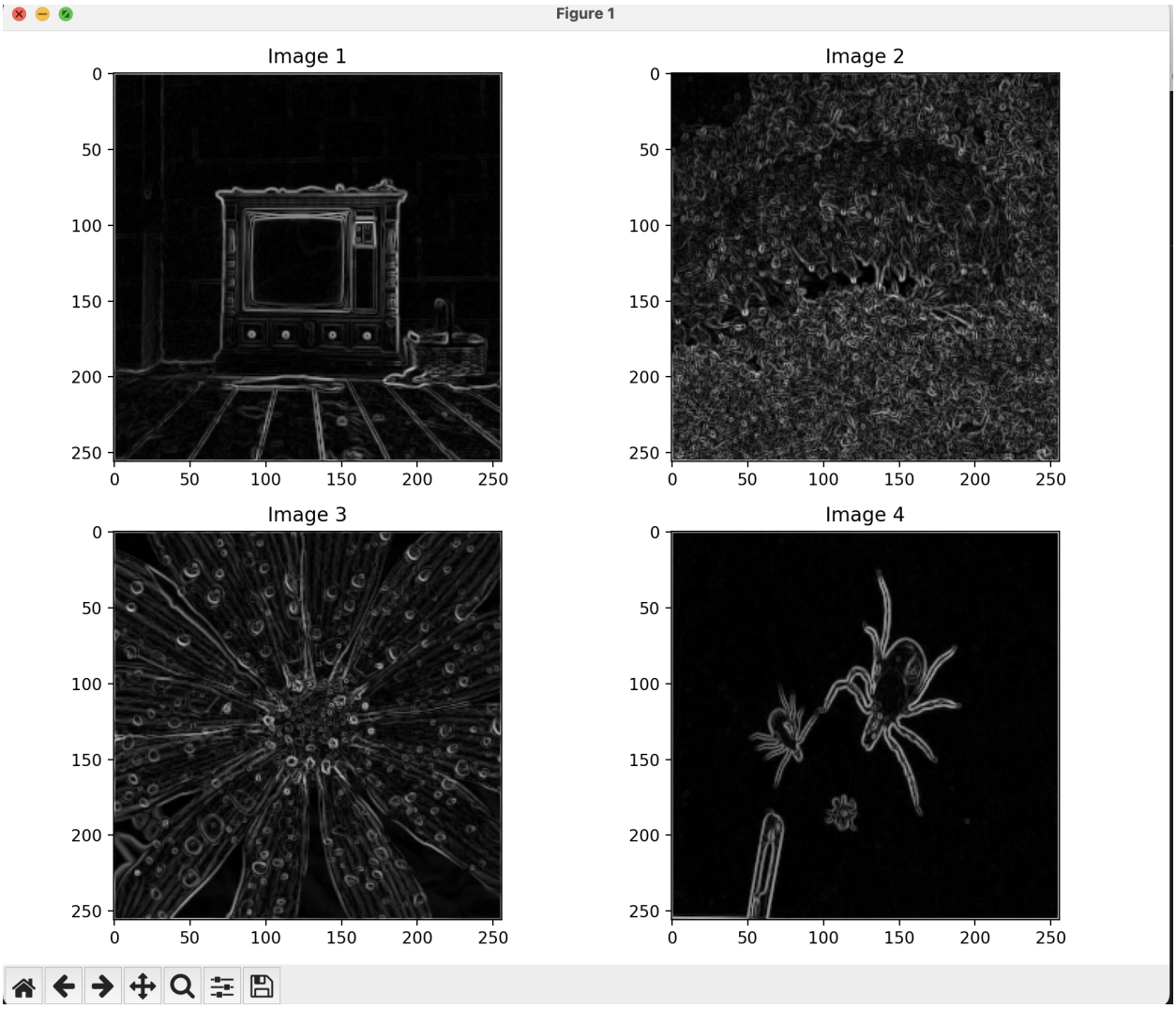
Figure 1
Image 1
Image 2
Image 3
Image 4
0
50
100
150
200
250