[heapq](https://docs.python.org/3/library/heapq.html#module-heapq) — Heap queue algorithm

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which heap[k] <= heap[2\*k+1] and heap[k] <= heap[2\*k+2] for all k, counting elements from zero.

heapq.**nlargest**(*n*, *iterable*, *key=None*)

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, key=str.lower). Equivalent to: sorted(iterable, key=key, reverse=True)[:n].

heapq.**nsmallest**(*n*, *iterable*, *key=None*)

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, key=str.lower). Equivalent to: sorted(iterable, key=key)[:n].

n Python, it is available using “**heapq**” module. The property of this data structure in Python is that each time the **smallest of heap element is popped(min heap)**. Whenever elements are pushed or popped, **heap structure in maintained**. The heap[0] element also returns the smallest element each time.

**Let’s see various Operations on heap :**

* **heapify(iterable)** :- This function is used to**convert the iterable into a heap** data structure. i.e. in heap order.
* **heappush(heap, ele)** :- This function is used to **insert the element** mentioned in its arguments into heap. The**order is adjusted**, so as **heap structure is maintained**.
* **heappop(heap)** :- This function is used to **remove and return the smallest element** from heap. The**order is adjusted**, so as **heap structure is maintained**.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Python code to demonstrate working of  # heapify(), heappush() and heappop()    # importing "heapq" to implement heap queue  import heapq    # initializing list  li = [5, 7, 9, 1, 3]    # using heapify to convert list into heap  heapq.heapify(li) |

# printing created heap

print ("The created heap is : ",end="")

print (list(li))

# using heappush() to push elements into heap

# pushes 4

heapq.heappush(li,4)

# printing modified heap

print ("The modified heap after push is : ",end="")

print (list(li))

# using heappop() to pop smallest element

print ("The popped and smallest element is : ",end="")

print (heapq.heappop(li))

**Output :**

The created heap is : [1, 3, 9, 7, 5]

The modified heap after push is : [1, 3, 4, 7, 5, 9]

The popped and smallest element is :

|  |
| --- |
| # Python code to demonstrate working of  # nlargest() and nsmallest()    # importing "heapq" to implement heap queue  import heapq    # initializing list  li1 = [6, 7, 9, 4, 3, 5, 8, 10, 1]    # using heapify() to convert list into heap  heapq.heapify(li1)    # using nlargest to print 3 largest numbers  # prints 10, 9 and 8  print("The 3 largest numbers in list are : ",end="")  print(heapq.nlargest(3, li1))    # using nsmallest to print 3 smallest numbers  # prints 1, 3 and 4  print("The 3 smallest numbers in list are : ",end="")  print(heapq.nsmallest(3, li1)) |

**Output :**

The 3 largest numbers in list are : [10, 9, 8]

The 3 smallest numbers in list are : [1, 3, 4]

K’th Largest/Smallest Element in an array

edit

play\_arrow

brightness\_4

|  |
| --- |
| ''' Python3 code for k largest elements in an array'''    def kLargest(arr, k):      # Sort the given array arr in reverse      # order.      arr.sort(reverse = True)      # Print the first kth largest elements      for i in range(k):          print (arr[i], end =" ")    # Driver program  arr = [1, 23, 12, 9, 30, 2, 50]  # n = len(arr)  k = 3  kLargest(arr, k)    # This code is contributed by shreyanshi\_ |

**Top K Frequent Elements**

Given a non-empty array of integers, return the ***k*** most frequent elements.

**Example 1:**

**Input:** nums = [1,1,1,2,2,3], k = 2

**Output:** [1,2]

**Example 2:**

**Input:** nums = [1], k = 1

**Output:** [1]

class Solution:

def topKFrequent(self, nums: List[int], k: int) -> List[int]:

d={}

for i in range(len(nums)):

if nums[i] in d:

d[nums[i]]+=1

else:

d[nums[i]]=1

sorted\_d=(sorted(d.items(), key=lambda item: item[1]))

return [k[0] for k in sorted\_d[-k:]]

**Merge k sorted arrays | Set 2 (Different Sized Arrays)**

Input: k = 3

arr[][] = { {1, 3},

{2, 4, 6},

{0, 9, 10, 11}} ;

Output: 0 1 2 3 4 6 9 10 11

Input: k = 2

arr[][] = { {1, 3, 20},

{2, 4, 6}} ;

Output: 1 2 3 4 6 20

from heapq import merge

# function for meging k arrays

def mergeK(arr, k):

    l = arr[0]

    for i in range(k-1):

        # when k = 0 it merge arr[1]

        # with arr[0] here in l arr[0]

        # is stored

        l = list(merge(l, arr[i + 1]))

    return l

# for printing array

def printArray(arr):

    print(\*arr)

# driver code

arr =[[2, 6, 12 ],

    [ 1, 9 ],

    [23, 34, 90, 2000 ]]

k = 3

l = mergeK(arr, k)

printArray(l)

**Output:**

Merged array is

1 2 6 9 12 23 34 90 2000

ef mergeKLists(self, lists: List[ListNode]) -> ListNode:

nodes = []

for lst in lists:

while(lst):

nodes.append(lst.val)

lst = lst.next

heapq.heapify(nodes)

head = n = ListNode()

while(nodes != []):

n.next = ListNode(heapq.heappop(nodes))

n = n.next

return head.next

**Top K Frequent Words**

Given a non-empty list of words, return the *k* most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency, then the word with the lower alphabetical order comes first.

**Example 1:**

**Input:** ["i", "love", "leetcode", "i", "love", "coding"], k = 2

**Output:** ["i", "love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**Example 2:**

**Input:** ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

**Output:** ["the", "is", "sunny", "day"]

**Explanation:** "the", "is", "sunny" and "day" are the four most frequent words,

with the number of occurrence being 4, 3, 2 and 1 respectively.

Given a non-empty list of words, return the *k* most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency, then the word with the lower alphabetical order comes first.

**Example 1:**

**Input:** ["i", "love", "leetcode", "i", "love", "coding"], k = 2

**Output:** ["i", "love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**Example 2:**

**Input:** ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

**Output:** ["the", "is", "sunny", "day"]

**Explanation:** "the", "is", "sunny" and "day" are the four most frequent words,

with the number of occurrence being 4, 3, 2 and 1 respectively.

def topKFrequent(self, words: List[str], k: int) -> List[str]:

cnts = Counter(words)

cnts = sorted(cnts.items(), key=lambda x: (-1\*x[1], x[0]))

return [x[0] for x in cnts[:k]]

from collections import Counter

Here is a simple example , that shows the working of Counter module.

from collections import Counter

list1 = ['x','y','z','x','x','x','y', 'z']

print(Counter(list1))

Output:

Counter({'x': 4, 'y': 2, 'z': 2})

**>>> from collections import Counter**

**>>> m = Counter(l)**

**>>> print(m)**

**Counter({'I': 2, 'am': 2, 'there': 1})**

**>>> cnts = sorted(m.items(), key=lambda x: (-1\*x[1], x[0]))**

**>>> print(cnts)**

**[('I', 2), ('am', 2), ('there', 1)]**

**>>>**

**Connect n ropes with minimum costs**

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to the sum of their lengths. We need to connect the ropes with minimum cost.

For example, if we are given 4 ropes of lengths 4, 3, 2, and 6. We can connect the ropes in the following ways.   
1) First, connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6, and 5.   
2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.   
3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.

**Algorithm:**

1. Create a min-heap and insert all lengths into the min-heap.
2. Do following while the number of elements in min-heap is not one.
   1. Extract the minimum and second minimum from min-heap
   2. Add the above two extracted values and insert the added value to the min-heap.
   3. Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost.

# Python3 program to connect n

# ropes with minimum cost

import heapq

def minCost(arr, n):

    # Create a priority queue out of the

    # given list

    heapq.heapify(arr)

    # Initializ result

    res = 0

    # While size of priority queue

    # is more than 1

    while(len(arr) > 1):

        # Extract shortest two ropes from arr

        first = heapq.heappop(arr)

        second = heapq.heappop(arr)

        #Connect the ropes: update result

        # and insert the new rope to arr

        res += first + second

        heapq.heappush(arr, first + second)

    return res

if \_\_name\_\_ == '\_\_main\_\_':

    lengths = [ 4, 3, 2, 6 ]

    size = len(lengths)

    print("Total cost for connecting ropes is " +

          str(minCost(lengths, size)))

**Output:**

Total cost for connecting ropes is 29

**Complexity Analysis:**

* **Time Complexity:**O(nLogn), assuming that we use a O(nLogn) sorting algorithm.   
  Note that heap operations like insert and extract take O(Logn) time.
* **Auxiliary Complexity:** O(n).

**K Closest Points to Origin**

We have a list of points on the plane.  Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order.  The answer is guaranteed to be unique (except for the order that it is in.)

**Input:** points = [[1,3],[-2,2]], K = 1

**Output:** [[-2,2]]

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

**Input:** points = [[3,3],[5,-1],[-2,4]], K = 2

**Output:** [[3,3],[-2,4]]

(The answer [[-2,4],[3,3]] would also be accepted.)

# Python3 program for implementation of

# above approach

# Function to return required answer

def pClosest(points, K):

    points.sort(key = lambda K: K[0]\*\*2 + K[1]\*\*2)

    return points[:K]

# Driver program

points = [[3, 3], [5, -1], [-2, 4]]

K = 2

print(pClosest(points, K))

##### **python solution using max\_heap | O(nlogk)| simple solution**

import heapq,math

from collections import defaultdict

class Solution:

def close\_point(self,a,b):

ans= math.sqrt(a\*a+ b\*b)

return ans

def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:

heap=[]

d =defaultdict(tuple)

heapq.heapify(heap)

for each\_element in points:

a,b = each\_element

ans = self.close\_point(a,b)

d[(a,b)]=ans

for k,v in d.items():

heapq.heappush(heap,(-1\*v,k))

if len(heap)>K:

heapq.heappop(heap)

#print(heap)

return [list(each[1]) for each in heap]

**s**