Algorithm Project On Patience Sort

Md.Ahnaf Rashid ID - 1931289 sec -2
Syeda Abida Sultana 1931160 sec-2
Jahedul Islam -1910343 sec -1
Group-16

History of Patience Sort

Patience Sort is a sorting Algorithm which rather then being well known for sorting is well known for finding the Longest Increasing Subsequence of a given set of data say in form of sequence of numbers in an array or just a given sequence of numbers. The inspiration behind this algorithm was from the card game **patience** where the rules were to keep making piles of cards which will remain in a sorted order in the pile.

Patience sort has many and rather unclear traces of its origins as original publications were hard to find regarding the very first implementation of the the Algorithm. However The name Patience sort was given by C.L Mallows and he credited its invention to A.S.C Ross in the 1960s. So we conclude it was invented in the 1960s.

Aldous and Diacons mentioned that it was used as finifng the longest increasing subsequence by Hammersley.

Much Later Robert W.Floyd recognized this as a sorting algorithm which was analyzed by Mallows. The floyds game was developed by Floyd and Donald Knuth according to available sources based on the same principle of patience sort.

A small Overview overall and a bit about how it works

As mentioned in the history section Patience sort was inspired by the card game patience. Now to provide a short summary the way it works as a sorting algorithm is say we have a few numbers like 7 2 4 9 5 3 6 12 11 8

We start making piles and the rules of making piles is such that we can make a pile using the first number we see in this case 7 and on top of it we can keep all numbers less than 7 , if it is not less than 7 we make a new pile with that greater number and keep following this process and keep stacking up numbers less then the first one in that pile like so

```
     3     8
2    5    11
7  4 9  6 12
```

Now as we see the piles here on top we use either priority queue after forming the piles or any other method to find the smallest numbers in piles, removing it from the pile and storing it in a list or an array of sorts, and doing so we will get a sorted sequence of numbers from the piles for in this case which is 2 3 4 5 6 7 8 9 11 12 . Note that a special rule we will always try to fill the leftmost pile first which implies it uses the greedy strategy.

```
IMPORT bisect and heapq
DEFINE FUNCTION sort(inputarray):
  SET piles TO [ ]
  FOR x IN inputarray:
    SET new_pile TO [x]
    SET i TO bisect.bisect_left(piles, new_pile)
    IF i != len(piles):
      piles[i].insert(0, x)
    ELSE:
      piles.append(new_pile)

 FOR i IN range(len(inputarray)):
    SET small_pile TO piles[0]
    SET inputarray[i] TO small_pile.pop(0)
    IF small_pile:
      heapq.heapreplace(piles, small_pile)
    ELSE:
      heapq.heappop(piles)
 CONTINUE till not piles
```

Pseudocode

Firstly here we will be using two python library modules named heapq and bisect Where heapq will make an implementation of the priority queue algorithm and the bisect module will help in locating an insertion point in which if we pass a data the list will be sorted.

The bisect module will help us in creating the piles and heapq will help in retrieving the smallest number from the pile and then we will put them in the list which will finally make a sorted list after placing all of them.

Implementation based on psuedocode

Based On pseudocode the code will be divided and explained into two parts. One where I create the piles and next where i find the minimum numbers from the pile and sort them in a list. The code is given below : -

We take demo input input = [5,12.5,-16.3,5,-4,6]

Pile Creating Part

```
import bisect

import heapq
def patiencesort(inparr):

    piles = [ ]

    for p in inparr:

        new_pile = [p]

        q = bisect.bisect_left(piles, new_pile)

        if q!=len(piles):

            piles[q].insert(0, p)

        else:

            piles.append(new_pile)
```

Sorting part

```
  for q in range(len(inparr)):

    small_pile = piles[0]

    inparr[q] = small_pile.pop(0)

    if small_pile:

        heapq.heapreplace(piles, small_pile)

    else:

        heapq.heappop(piles)

  assert not piles
```

Passing data

```
input = [5,12.5,-16.3,5,-4,6]

patiencesort(input)

print(input)
```

Explaining the pile creation part

```python
import bisect
import heapq
def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

```

```python
input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
```

Here we take the pythons list named input which is going to be our set of numbers which we will sort

We pass the input to our function patiencesort where it is passed to the variable named inparr which now has the input list

list-inparr

0	1	2	3	4	5
5	12.5	-16.3	5	-4	6

```python
1 import bisect
2 import heapq
3 def patiencesort(inparr):
4     piles = []
5     for p in inparr:
6         new_pile = [p]
7         q = bisect.bisect_left(piles, new_pile)
8         if q!=len(piles):
9             piles[q].insert(0, p)
10        else:
11            piles.append(new_pile)
12
```

In line 4 we create a new list named piles which will be an empty list for now

We run a for loop p upto the last place of inparr which will serially iterate and store the value of inparr list starting from index 0 to 5 to the new list called new_pile. In the first run new_pile stores 5 as it is the data of the first index in inparr

list-inparr

0	1	2	3	4	5
5	12.5	-16.3	5	-4	6

Note that python list is very beneficial as it not only can store single values but can also store another list in its list whose concept we are using to make the piles. Note that python list is also dynamic so we can add remove delete at will.

Now remember from last part we stored the value of p in a new list named new pile like so

```
6        new_pile = [p]
```

We have now the new list new_pile

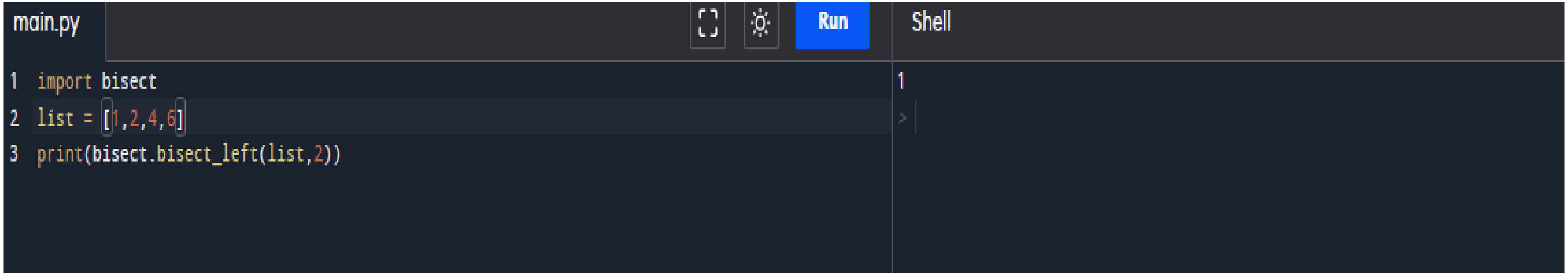
new_pile - 0

5

Now I move On to line 7 where we will be using module bisect and use bisect.bisect_left to find and return the position of the list at which if the element is inserted a sorted order will be maintained .
More explanation next page-

```
1  import bisect
2  import heapq
3  def patiencesort(inparr):
4      piles = []
5      for p in inparr:
6          new_pile = [p]
7          q = bisect.bisect_left(piles, new_pile)
8          if q!=len(piles):
9              piles[q].insert(0, p)
10         else:
11             piles.append(new_pile)
12
```

To understand how bisect works take an example code -



Here say we have a list 1 2 4 6 having index position of 0 1 2 3 respectively
Say we want to insert 2 now where in the left most region possible would we put 2 in the list so that the list remains sorted

The answer is at position 1 as it will give 1 2 2 4 6 having index 0 1 2 3 4
This concept was based on binary search

Back To code

Now back in line 7 we have piles and new_pile in bisect.bisect_left with piles being where we want to insert and new_pile being the element we want to insert.

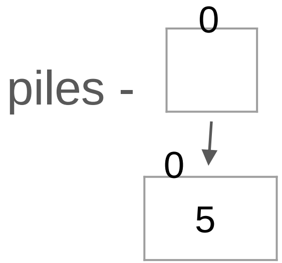
If we remember, piles was initially created as empty list so bisect.left will find that if new_pile is placed in 0th index the pile will be sorted

So q = 0
Now since piles is still an empty list so its size is 0 so in line 8 the else loop will initiate
And by piles.append(new_piles)
We will add new_pile which had the value 5 into the list piles

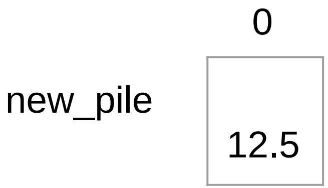
Now piles has one data which is 5

```
1  import bisect
2  import heapq
3  def patiencesort(inparr):
4      piles = []
5      for p in inparr:
6          new_pile = [p]
7          q = bisect.bisect_left(piles, new_pile)
8          if q!=len(piles):
9              piles[q].insert(0, p)
10         else:
11             piles.append(new_pile)
12
```

```
23  input = [5,12.5,-16.3,5,-4,6]
24  patiencesort(input)
```



Now we continue the loop now p has value 12.5 from inparr
We now have new_pile=12.5

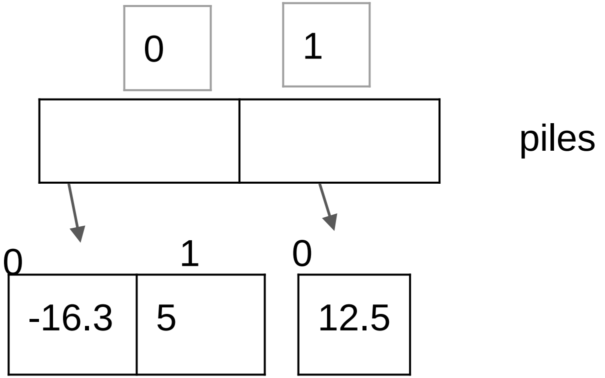


We now execute the line 7 bisect.bisect left and check **where would we place 12.5 so that the list named piles remains sorted ?**
Since 12.5 is greater than 5 it will be place in position / index one(1) so q = 1

Now since q = 1 and length of piles is also one the if condition if q!=len(piles): will not execute, else condition will proceed and add 12.5 at the end of the list named piles using append

Now we continue the loop again now p denotes to -16.3

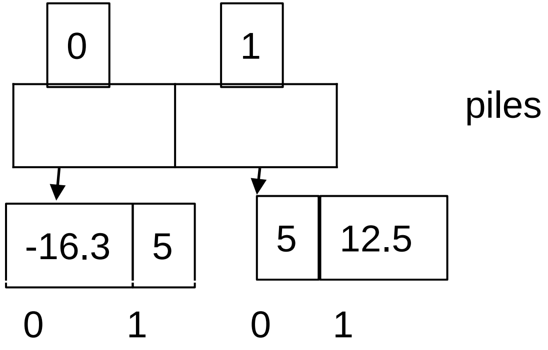
Store p in new_pile
Check where -16.3 can be place so that piles will be sorted
Since -16.3< 5 So at index 0
So q= 0 and len(piles) = 2
If loop will execute insert p = -16.3 at 0 th index of piles[q] i.e piles[0].insert(0,-16.3)




```
import bisect
import heapq
def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
```

Now moving on p denotes to 5.
We store in new_pile.
Check using bisect.left and find we can place it on index 1 so q =1 and length of piles len(piles) = 2
If loop will execute and I insert data at piles[1].insert(0,5)




```python
import bisect
import heapq
def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
```

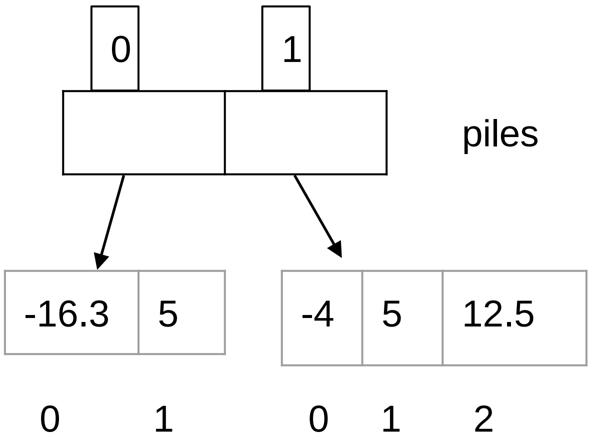
Move on to p denotes to -4 of inparr

We store -4 in new_pile

Check in bisect we get q = 1

len(piles) = 2 so they are not equal

If condition will execute and i will insert -4 into piles[1].insert(0,-4)at 0th index




```python
import bisect
import heapq
def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
```

Move on to the last number 6

new_pile = 6

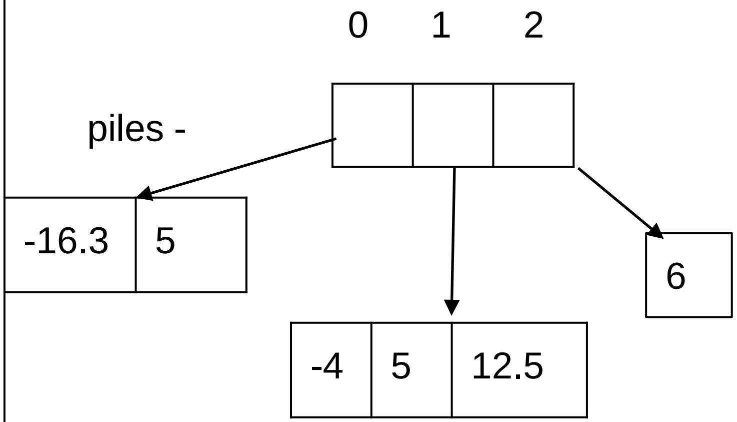
Check bisect_left , we get q = 2

And len(piles) = 2

So if condition wont check out

We go to else loop and append the new_pile =6 at the end of piles

So we get




```python
import bisect
import heapq
def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
```

There are no more elements in the list the first loop terminates and the pile creation is done.

Now for the sorting part…..

```
for q in range(len(inparr)):
    small_pile = piles[0]
    inparr[q] = small_pile.pop(0)
    if small_pile:
        heapq.heapreplace(piles, small_pile)
    else:
        heapq.heappop(piles)
assert not piles
```

Here we are using heapq.heapreplace and heqpq.heappop()

The heqpq module which applies the priority queue algorithm where and follows minheap property i.e smallest element is at the root.

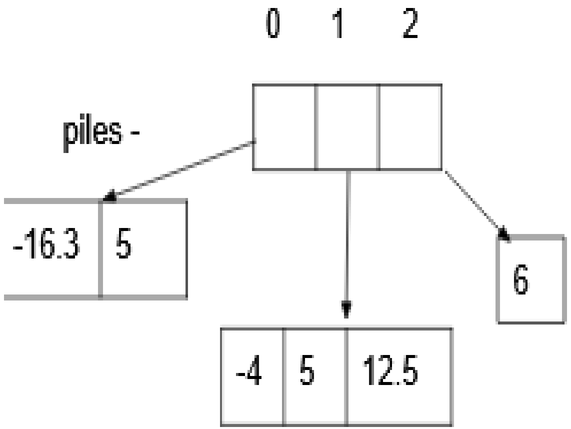
Heapq.heapreplace is used here to return the smallest item form heap, here heap is piles and to push the item on heap for in this case is small_pile back on the heap

Heap.pop pushes the item to the heap

```python
    for q in range(len(inparr)):
        small_pile = piles[0]
        inparr[q] = small_pile.pop(0)
        if small_pile:
            heapq.heapreplace(piles, small_pile)
        else:
            heapq.heappop(piles)
    assert not piles
```

We run a loop q equal to the length of our inparr list

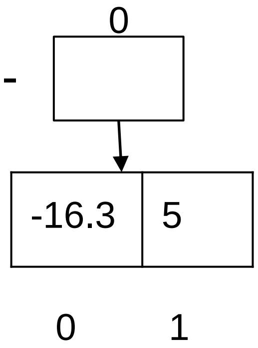
We take a variable small_pile and store the data of the 0th index of piles i.e piles[0]




```
for q in range(len(inparr)):
    small_pile = piles[0]
    inparr[q] = small_pile.pop(0)
    if small_pile:
        heapq.heapreplace(piles, small_pile)
    else:
        heapq.heappop(piles)
assert not piles
```

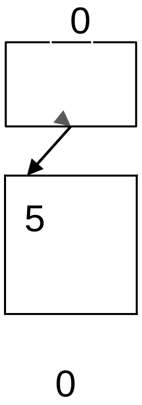
So now small_pile has

small_pile -




```python
for q in range(len(inparr)):
    small_pile = piles[0]
    inparr[q] = small_pile.pop(0)
    if small_pile:
        heapq.heapreplace(piles, small_pile)
    else:
        heapq.heappop(piles)
assert not piles
```

Now at inparr[q] where initially q =0 we store small_pile.pop(0) we remove the 0th index store it in the 0th index of inparr



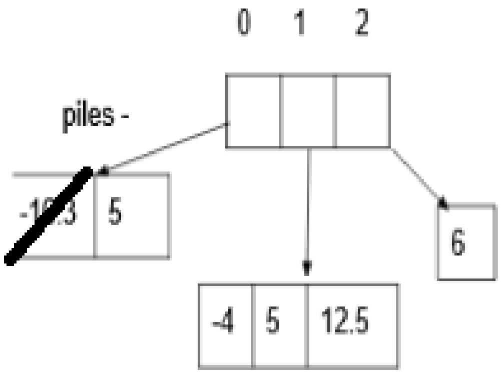
small_pile

inparr -

-16.3	12.5	-16.3	5	-4	6

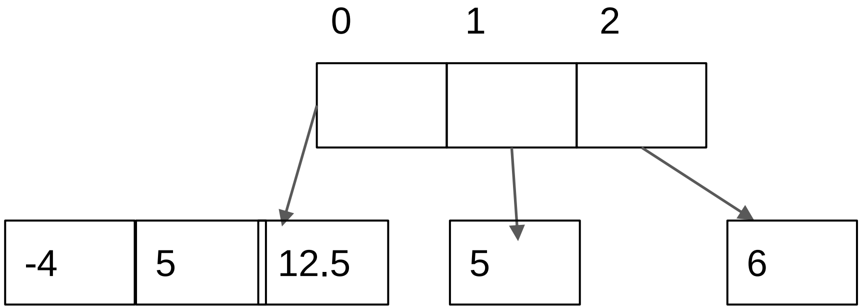
Now heaprelplace finds the smallest item and locates in which pile it is

The next smallest item is in 1th index of piles




```python
for q in range(len(inparr)):
    small_pile = piles[0]
    inparr[q] = small_pile.pop(0)
    if small_pile:
        heapq.heapreplace(piles, small_pile)
    else:
        heapq.heappop(piles)
assert not piles
```

Now since we removed the first smallest digit the heap has changed and the new pile in index 0 will be

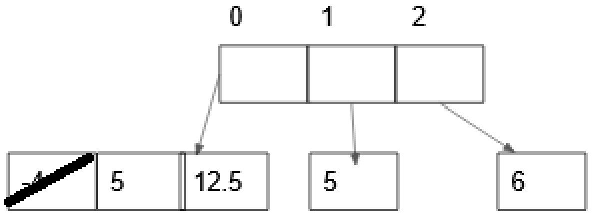


Now the smallest value is again in the root

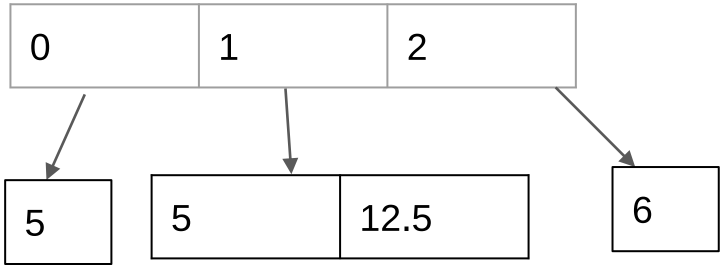
We then run the loop with remaining length and get

-16.3	-4	-16.3	5	-4	6

Now we again find the next smallest number in pile using heapq.heapreplace and find the least number to be in



The 1th index of pile
We then bring it to root

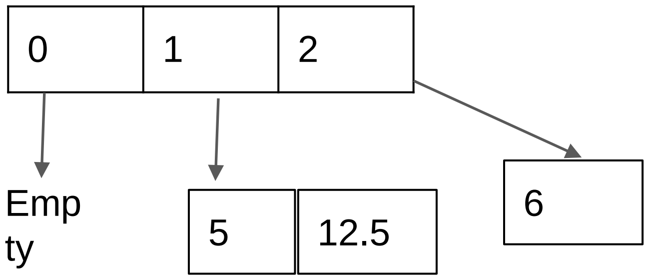


We then again store the pile[0] in small_pile , pop small_pile and store in inparr[q]

Which gives us

-16.3	-4	5	5	-4	6

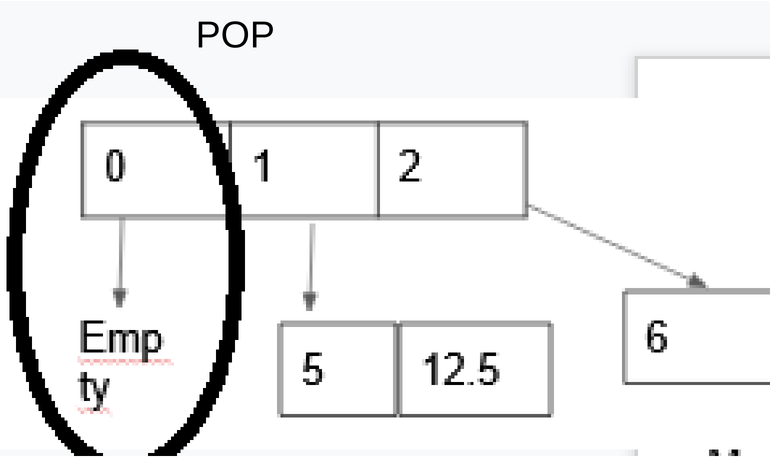
Moving on we get here an emptly list now



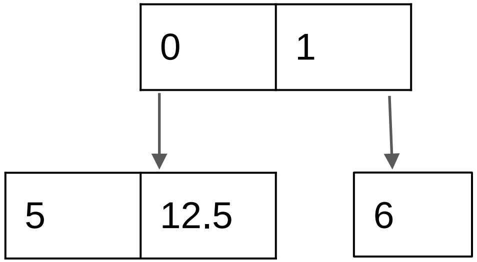
Since empty the else loop will initiate

```
if small_pile:
    heapq.heapreplace(piles, small_pile)
else:
    heapq.heappop(piles)
```

What this else loop will do is pop the empty list with 0th index



Then we will have



We thus continue this process till there are no more piles left and while we keep storing pop of small_pile in inp array we will get a sorted list

-16.3	-4	5	5	6	12.5

```
IMPORT bisect and heapq
DEFINE FUNCTION sort(inputarray):
  SET piles TO [ ]
  FOR x IN inputarray:
    SET new_pile TO [x]
    SET i TO bisect.bisect_left(piles, new_pile)
    IF i != len(piles):
      piles[i].insert(0, x)
    ELSE:
      piles.append(new_pile)

 FOR i IN range(len(inputarray)):
    SET small_pile TO piles[0]
    SET inputarray[i] TO small_pile.pop(0)
    IF small_pile:
      heapq.heapreplace(piles, small_pile)
    ELSE:
      heapq.heappop(piles)
  CONTINUE till not piles
```

n times

logn times as it uses binary search

nlogn

n times

O(1)

logn for priority queue

nlong

So the time complexity stands as nlogn+nlogn

= 2nlogn

=O(nlogn)

Points To note

- Not stable , does not maintain the order the input is given in it generally depends whether that number entered into the first / leftmost piles first

-It is inplace as at most the piles creation , the running in sequence and storing takes the same space as sample input

- It is online as it does not know what type of data it will deal with before executing

-Yes it is adaptive , depending on the sortedness or size of input ,the size or shape of piles and runtime may change

-Can work with positive , negative ,floating(neg and pos) , integer numbers (neg and pos)

-Follows greedy paradigm

Pros and Cons

Pros

-has running time of nlogn and sometimes n when input is sorted

- used in finding longest increasing subsequence
- is used in playing the card game patience

-can sort data from a piles of data

Cons

- hard to implement
- too many variations and not worth the hassle
- better and easier codes available

-not that much used

Practical Uses

- This algorith was used as process control according to wiki and further research led to this algorithm being used in a site called Bazaar as version control.Besides this according to a paper in microsoft a modified version of the patience sort is being used in ata processing systems and applications - reference https://www.microsoft.com/en-us/research/uploads/prod/2018/04/impatience-icde18.pdf main.py

Run

```python
import bisect
import heapq


def patiencesort(inparr):
    piles = []
    for p in inparr:
        new_pile = [p]
        q = bisect.bisect_left(piles, new_pile)
        if q!=len(piles):
            piles[q].insert(0, p)
        else:
            piles.append(new_pile)

    for q in range(len(inparr)):
        small_pile = piles[0]
        inparr[q] = small_pile.pop(0)
        if small_pile:
            heapq.heapreplace(piles, small_pile)
        else:
            heapq.heappop(piles)
    assert not piles


input = [5,12.5,-16.3,5,-4,6]
patiencesort(input)
print(input)
```

Shell

```
[-16.3, -4, 5, 5, 6, 12.5]
>
```