POCKETLIST
LTask Management Application

CODE ANALYSIS REPORT

Proposed by:
Mohammad Ibrahim Ali
06177

Basic Functions

- Add Task
- Remove Task
- Sort Task
  - by Name
  - by Starting Date
  - by Deadline
  - Target Hour
  - Urgency

Add Task

Basic Code: List Append Function

```
def addtask(space):
    temp = ['OPD Reminder', '0', '2020/07/12', '02:16:12', '2020/07/13', '14:00:00', '1', '0', '']
    space.append(temp)
```

Complexity

Adding an element to a list would be considered one operation.

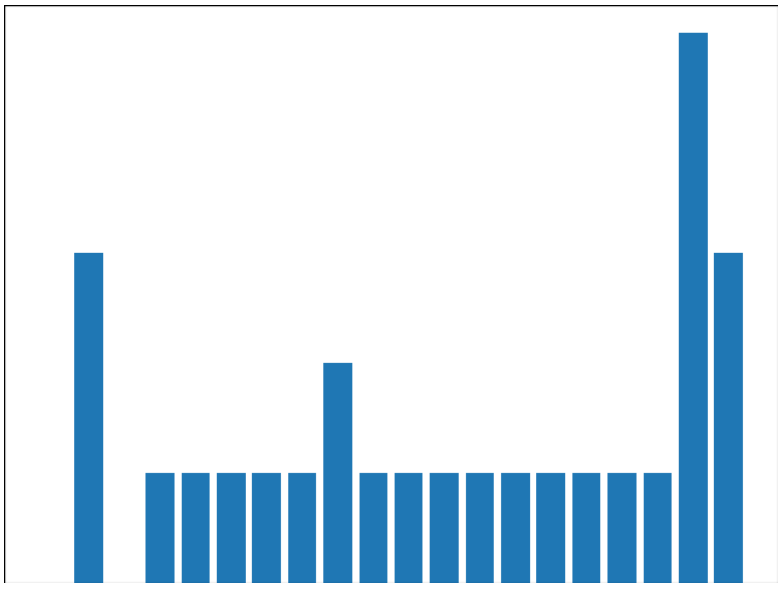
Once the function runs, it performs the operation only one time.

Hence, the big-O time complexity for this function would be:

$$O(1)$$

Performance Graph

The performance graph of the function in terms of the time consumed, for adding 20 elements in the list is given beside.



Remove Task

Basic Code: If-else Checkpoint

```
def removetask(space):
    n = random.randint(0, len(space)-1)
    x = space[n][3]
    x = x.split(':')
    x = ''.join(x)
    x = int(x)
    if space:
        if len(space) == 1:
            del(space[0])
        else:
            del(space[binary(space, 0, len(space), x)])
    else:
        print('There is no task to remove.')
```

Basic Code: Quick-sort Algorithm

```
def partition(lst, x, y, typeindex):
    i = x - 1
    for j in range(x, y):
        if numery2(lst[j][typeindex]) <= numery2(lst[y][typeindex]):
            i += 1
            lst[i], lst[j] = lst[j], lst[i]
    lst[i+1], lst[y] = lst[y], lst[i+1]
    return i + 1

def quicksort(lst, x, y, typeindex):
    if not x < y:
        return
    z = partition(lst, x, y, typeindex)
    quicksort(lst, x, z - 1, typeindex)
    quicksort(lst, z + 1, y, typeindex)
```

Remove Task

Basic Code: Binary Search Algorithm

```
def binary(lst, left, right, x):
  mid = (left + right)//2
  temp = lst[mid][3]
  temp = temp.split(':')
  temp = ''.join(temp)
  temp = int(temp)
  if temp == x:
    return mid
  else:
    if temp > x:
      return binary(lst, left, mid-1, x)
    elif temp < x:
      return binary(lst, mid+1, right, x)
```

Complexity

The if-else checkpoint in the function have all operations that run only one time, once the part of the function is called. Hence, the complexity of the if-else checkpoint is

$$O(1)$$

The Quick-sort Algorithjm , in the worst case, will make partitions of the list by taking the first or the last element as the pivot. In that case, the number of operations would double the number of elements in the list, while getting on each element. hence, the complexity of the quick-sort algorithm would be,

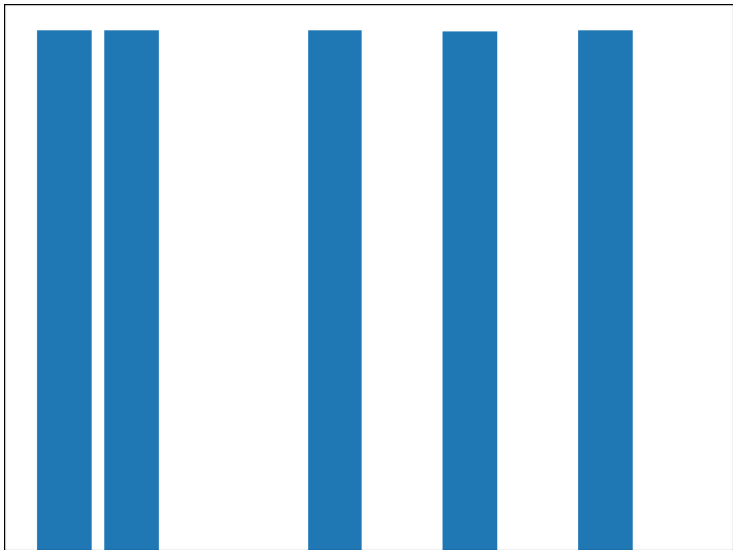
$$O(n^2)$$

The Binary Search Algorithm, in the worst case, will use the divide-and-conquer appraoch. It would repeatedly halve the list, and will choose one of those two halves, In repetitive bisection, the number of operations performed would be a function of logarithm, Hence, the complexity of the binary-search alogirthm would be,

$$O(\log n)$$

Performance Graph

The performance graph of the function in terms of the time consumed, for removing 10 elements in the list is given beside.



Sort Task (by Name)

Basic Code: Quick-sort Algorithm

```
def partition(lst, x, y, typeindex):
  i = x - 1
  for j in range(x, y):
    if lst[j][typeindex] <= lst[y][typeindex]:
      i += 1
      lst[i], lst[j] = lst[j], lst[i]
  lst[i+1], lst[y] = lst[y], lst[i+1]
  return i + 1

def sortbyname(lst, x, y, typeindex):
  if not x < y:
    return
  z = partition(lst, x, y, typeindex)
  sortbyname(lst, x, z - 1, typeindex)
  sortbyname(lst, z + 1, y, typeindex)
```

Sort Task (by Starting Date)

Basic Code: Quick-sort Algorithm

```
def numery1(x):
    x = x.split('/')
    x = ''.join(x)
    x = int(x)
    return x

def partition(lst, x, y, typeindex):
  i = x - 1
  for j in range(x, y):
    if numery1(lst[j][typeindex]) <= numery1(lst[y][typeindex]):
      i += 1
      lst[i], lst[j] = lst[j], lst[i]
  lst[i+1], lst[y] = lst[y], lst[i+1]
  return i + 1

def sortbystartingdate(lst, x, y, typeindex):
  if not x < y:
    return
  z = partition(lst, x, y, typeindex)
  sortbystartingdate(lst, x, z - 1, typeindex)
  sortbystartingdate(lst, z + 1, y, typeindex)
```

Sort Task (by Deadline)

Basic Code: Quick-sort Algorithm

```
def numery1(x):
    x = x.split('/')
    x = ''.join(x)
    x = int(x)
    return x

def partition(lst, x, y, typeindex):
  i = x - 1
  for j in range(x, y):
    if numery1(lst[j][typeindex]) <= numery1(lst[y][typeindex]):
      i += 1
      lst[i], lst[j] = lst[j], lst[i]
  lst[i+1], lst[y] = lst[y], lst[i+1]
  return i + 1

def sortbydeadline(lst, x, y, typeindex):
  if not x < y:
    return
  z = partition(lst, x, y, typeindex)
  sortbydeadline(lst, x, z - 1, typeindex)
  sortbydeadline(lst, z + 1, y, typeindex)
```

Sort Task (by Urgency)

Basic Code: Quick-sort Algorithm

```
def partition(lst, x, y, typeindex):
  i = x - 1
  for j in range(x, y):
    if lst[j][typeindex] <= lst[y][typeindex]:
      i += 1
      lst[i], lst[j] = lst[j], lst[i]
  lst[i+1], lst[y] = lst[y], lst[i+1]
  return i + 1

def sortbyurgency(lst, x, y, typeindex):
  if not x < y:
    return
  z = partition(lst, x, y, typeindex)
  sortbyurgency(lst, x, z - 1, typeindex)
  sortbyurgency(lst, z + 1, y, typeindex)
```

Sort Task (by Target Hour)

Basic Code: Quick-sort Algorithm

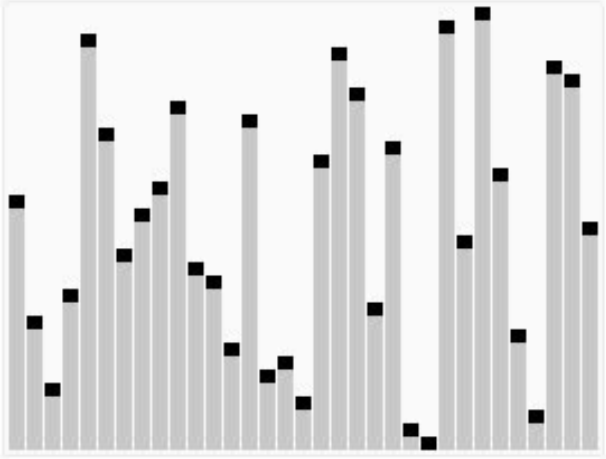
```
def numery2(x):
    x = x.split(':')
    x = ''.join(x)
    x = int(x)
    return x

def partition(lst, x, y, typeindex):
  i = x - 1
  for j in range(x, y):
    if numery2(lst[j][typeindex]) <= numery2(lst[y][typeindex]):
      i += 1
      lst[i], lst[j] = lst[j], lst[i]
  lst[i+1], lst[y] = lst[y], lst[i+1]
  return i + 1

def sortbytargethour(lst, x, y, typeindex):
  if not x < y:
    return
  z = partition(lst, x, y, typeindex)
  sortbytargethour(lst, x, z - 1, typeindex)
  sortbytargethour(lst, z + 1, y, typeindex)
```

Complexity

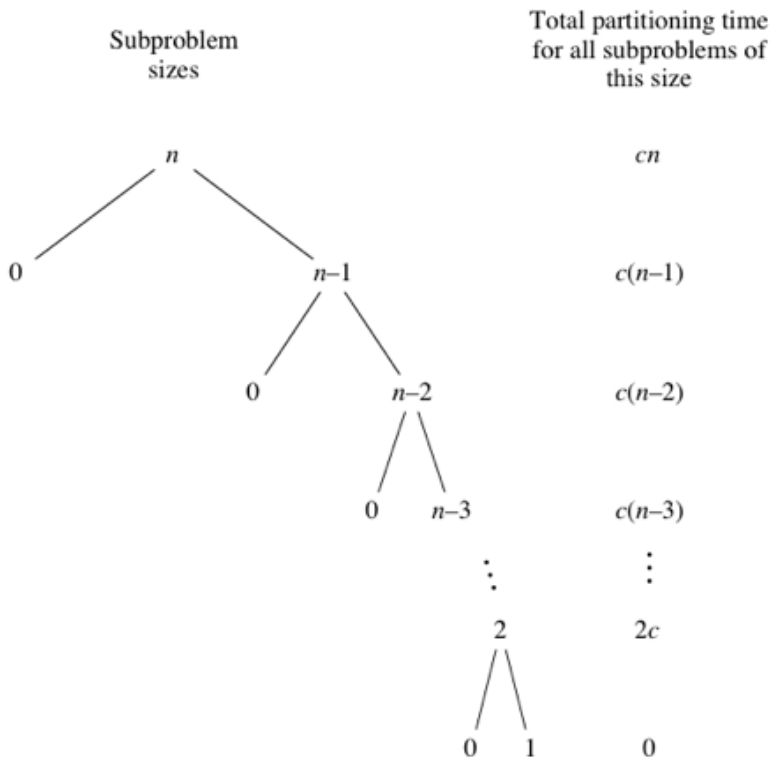
The Quick-sort Algorithm, though more efficient than other complex sorting algorithms, can take a bit more time than usual, if the pivot selection isn't favorable. A visual representation of how quick-sort works is shown below.



Sort Task

Complexity

In the worst case, the selection of pivot that divides the list into two partitions with elements greater than and lesser than the pivot respectively, would be either the first element or the last element. This would infact, generate the tree given here.

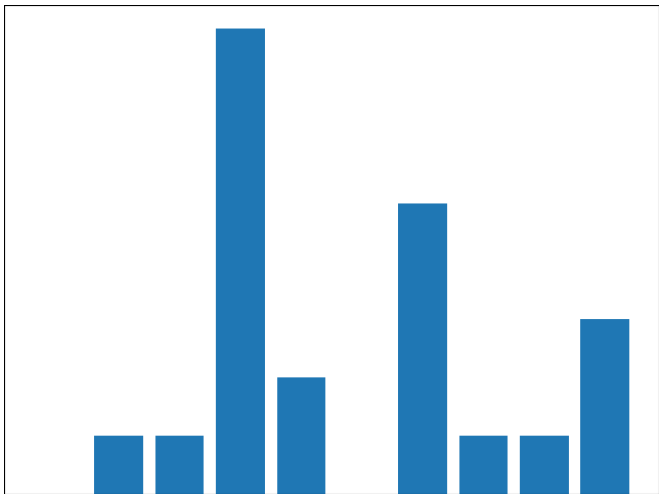


If we make a sequence of number of operations (sorting operation) performed at each iteration, we would get the following sequence. Adding the elements through the sum of series formula would derive the following. We need to subtract 1 from it, since 1 isn't included in the sequence of operations

$$\begin{aligned} cn+c(n-1)+c(n-2)+\ldots+2c &= c(n+(n-1)+(n-2)+\ldots+2) \\ &= c((n+1)(n/2)-1)\ . \end{aligned}$$

In big-Θ notation, quicksort's worst-case running time is $O(n^2)$

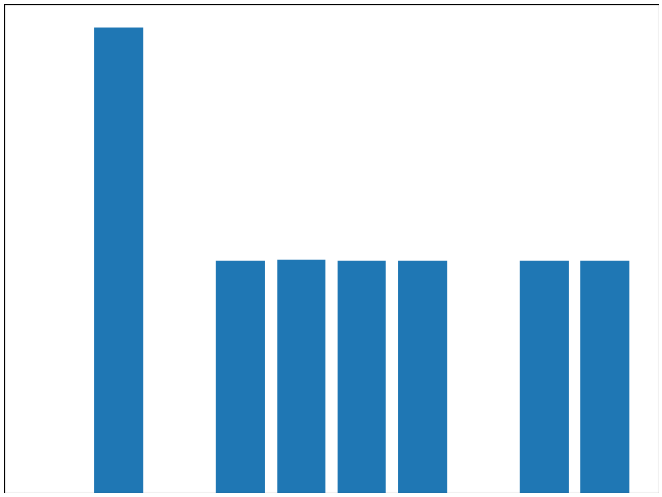
Sort Task (by Name)



Performance Graph

The performance graph of the function in terms of the time consumed, for shuffling the list and then sorting it by the task name for 10 iterations.

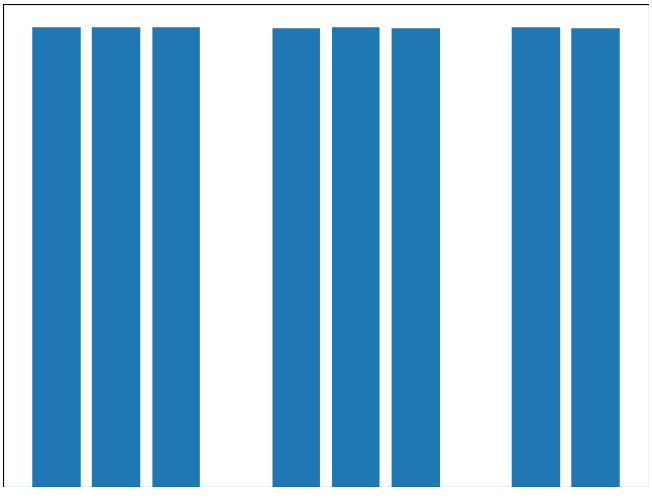
Sort Task (by Starting Date)



Performance Graph

The performance graph of the function in terms of the time consumed, for shuffling the list and then sorting it by the starting date for 10 iterations.

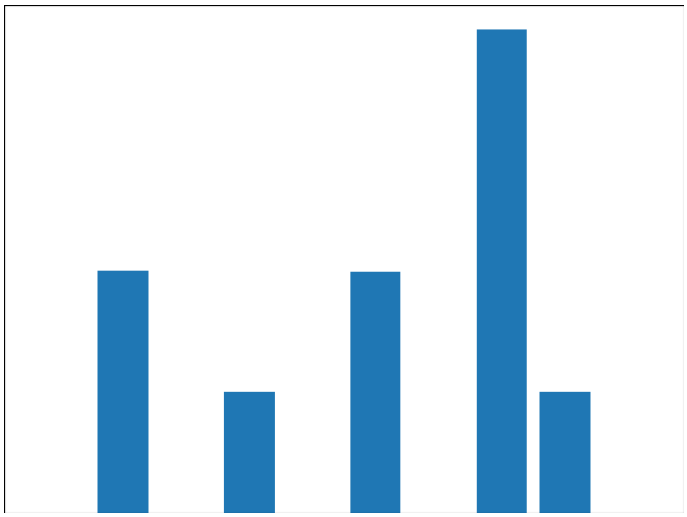
Sort Task (by Deadline)



Performance Graph

The performance graph of the function in terms of the time consumed, for shuffling the list and then sorting it by the deadline for 10 iterations.

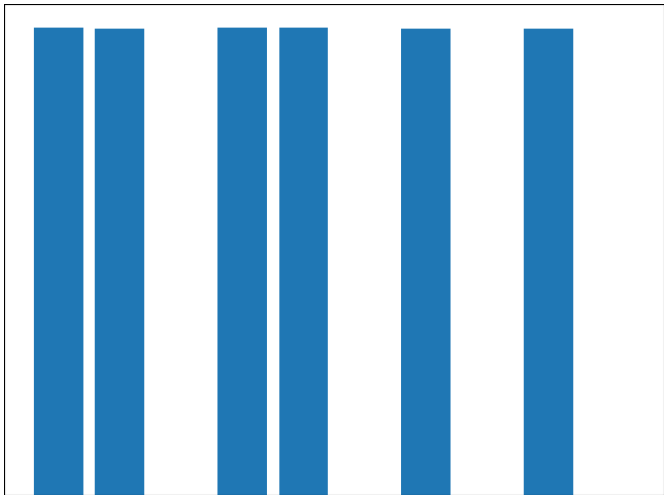
Sort Task (by Target Hour)



Performance Graph

The performance graph of the function in terms of the time consumed, for shuffling the list and then sorting it by the target hour for 10 iterations.

Sort Task (by Urgency)



Performance Graph

The performance graph of the function in terms of the time consumed, for shuffling the list and then sorting it by the urgency for 10 iterations.

References - Documentation

https://stackoverflow.com/questions/58792963/time-complexity-for-adding-elements-to-list-vs-set-in-python#:~:text=The%20time%20complexity%20for%20appending,array%20that%20the%20data%20is

https://en.wikipedia.org/wiki/Quicksort#/media/File:Sorting_quicksort_anim.gif

https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksor

https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/running-time-of-binary-search