CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

Mergesort

Revised based on textbook author's notes.

Review

- **sorting** – the process of arranging a collection of items such that each item and its successor satisfy a prescribed relationship.
  - **sort key** – values on which items are ordered.
  - items arranged in ascending or descending order.

Sorting Algorithms

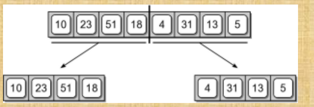
- Can be divided into two categories:
  - **comparison sorts**
    - items are arranged by performing pairwise logical comparisons between two sort keys.
  - **distribution sorts**
    - distributes the sort keys into intermediate groups based on individual key values.

Merge Sort

- Uses a divide and conquer strategy to sort the keys stored in a sequence.
  - Keys are recursively divided into smaller and smaller subsequences until 1 element.
  - These individual elements are in order by themselves
  - Subsequences are merged back together.

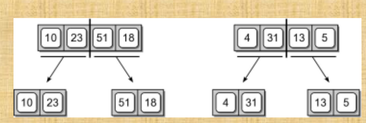
Merge Sort – Divide

- Starts by splitting the original sequence in the middle to create two subsequences of **approximately** equal size.



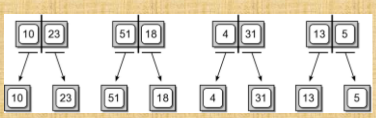
Merge Sort – Divide

- The two subsequences are then split in the middle.



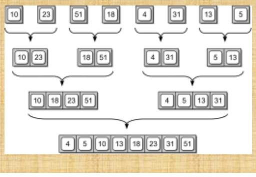
Merge Sort – Divide

- The subdivision continues until there is a single item in the sequence.



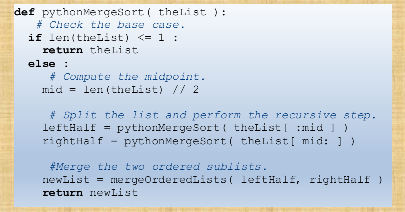
Merge Sort – Conquer

- After the sequences are split, they are merge back together, two at a time to create sorted sequences.



Merge Sort Code #1

- A simple implementation for sorting a Python list.



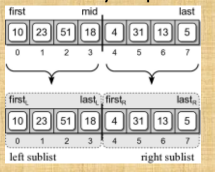
```
def pythonMergeSort( theList ):
   # Check the base case.
  if len(theList) <= 1 :
    return theList
  else :
     # Compute the midpoint.
    mid = len(theList) // 2

     # Split the list and perform the recursive step.
    leftHalf = pythonMergeSort( theList[ :mid ] )
    rightHalf = pythonMergeSort( theList[ mid: ] )

     #Merge the two ordered sublists.
    newList = mergeOrderedLists( leftHalf, rightHalf )
    return newList
```

Merge Sort – Improved Version

- The previous version:
  - only works with Python lists.
  - the splitting creates new physical lists.
  - uses the slice operation which is time consuming.

Merge Sort – Improved Version

- We can improve the implementation:
  - using virtual subsequences.
  - that works with any sequence.



Merge Sort Code #2

- An improved version of the merge sort.

```
def recMergeSort( theSeq, first, last, tmpArray ):
    # Check the base case.
  if first == last :
    return
  else :
     # Compute the mid point.
    mid = (first + last) // 2

     # Split the sequence and perform the recursive step.
    recMergeSort( theSeq, first, mid, tmpArray )
    recMergeSort( theSeq, mid+1, last, tmpArray )

     # Merge the two ordered subsequences.
    mergeSeq( theSeq, first, mid+1, last, tmpArray )
```

Merging Sorted Sequences

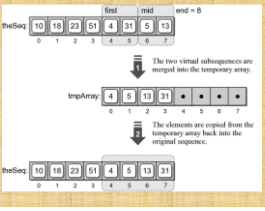
```
def mergeSeq( theSeq, left, right, end, tmpArray ):
  a = left
  b = right
  m = 0

  while a < right and b <= end :
    if theSeq[a] < theSeq[b] :
      tmpArray[m] = theSeq[a]
      a += 1
    else :
      tmpArray[m] = theSeq[b]
      b += 1
    m += 1
             :
             :
```

Merging Sorted Sequences

```
           :
           :

  while a < right :   # in parallel with first while
    tmpArray[m] = theSeq[a]
    a += 1
    m += 1

  while b <= end :    # in parallel with the two whiles
    tmpArray[m] = theSeq[b]
    b += 1
    m += 1

  for i in range( end - left + 1 ) :
    theSeq[i+left] = tmpArray[i]
```

Merge Sort – Temporary Array

- A temporary array is used to merge two virtual subsequences.


first
mid
end = 8
theSeq:
10
18
23
51
4
31
5
13
0
1
2
3
4
5
6
7
1
The two virtual subsequences are merged into the temporary array.
tmpArray:
4
5
13
31
•
•
•
•
0
1
2
3
4
5
6
7
2
The elements are copied from the temporary array back into the original sequence.
theSeq:
10
18
23
51
4
5
13
31
0
1
2
3
4
5
6
7

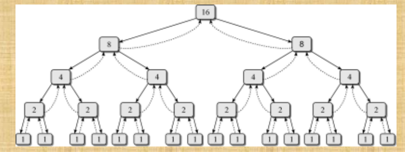
Wrapper Functions

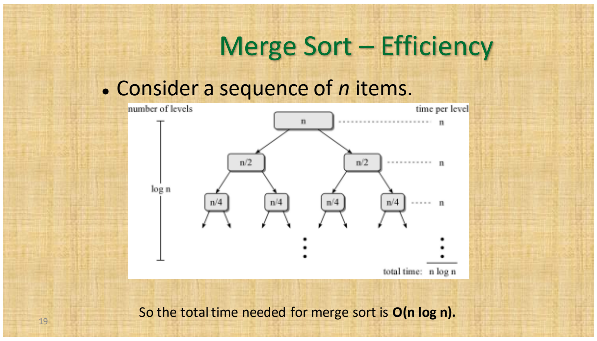
- A function that provides a simpler and cleaner interface for another function.
  - Provides little or no additional functionality.
  - Commonly used with recursive functions that require additional arguments.

```
def mergeSort( theSeq ):
  n = len( theSeq )
  tmpArray = Array( n )
  recMergeSort( theSeq, 0, n-1, tmpArray )
```

Merge Sort – Efficiency

- We need to determine the number of invocations and the time required by each function.





Merge Sort – Efficiency

- Consider a sequence of *n* items.

So the total time needed for merge sort is **O(n log n).**

19