Advanced Sorting

AP Computer Science

Sorting

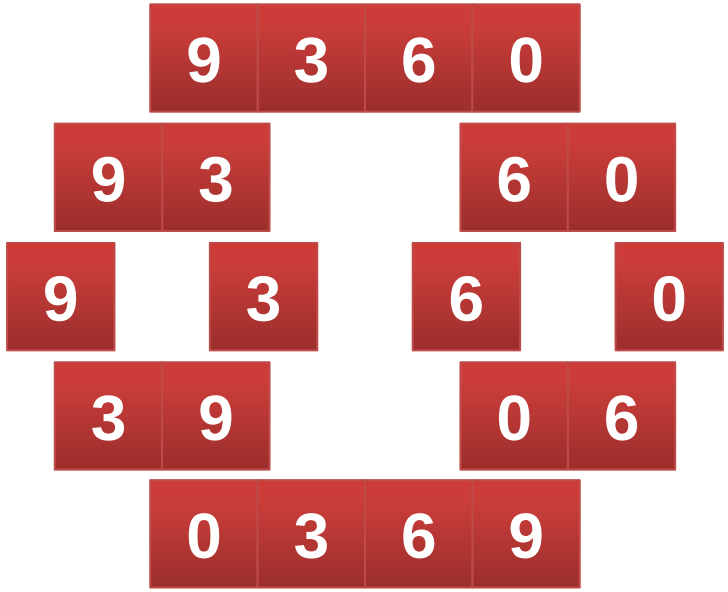
- Remember all of the sorting algorithms we already covered?
  - Bubble, selection, insertion, bogo, etc.
- On average, how many computational steps did it take to sort a list of size ***n*** using one of these sorts?
  - **n^2** computational steps
- We can do much better than that!

Merge sort

Merge sort

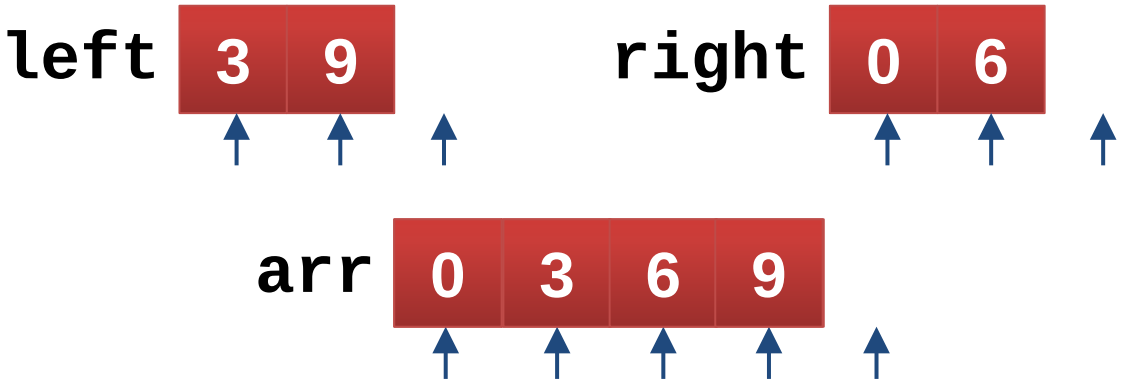
- Merge sort is classified as a "divide and conquer" algorithm
- Here is the basic idea:
  - Divide the unsorted list into 2 sublists
  - Recursively call merge sort on both sublists
    - ... which creates two more sublists for each call!
    - Repeats until there is only one element in the sublist
  - Merge the two sublists back together (and maintain sorted order), then return it

Merge sort: step-by-step



s([0])	→	[9][3]
s([6,0])	→	merge(s([9]), s([3])) → [3,9]
s([9,3,6,0])	→	merge(s([9,9]), s([0,0]))
The Stack	→	[0,3,6,9]

Merge: step-by-step



- Make an index pointer for **`left`**, **`right`**, and **`arr`**, e.g. **`l = 0`**, **`r = 0`**, **`i = 0`**.
- Add the minimum of **`left`** and **`right`** into **`arr`**, then update the index pointers.
- Repeat until you reach the end of **`arr`**.

Merge sort code

```
int[] mergeSort(int[] arr):
  if (arr.length <= 1): return arr // base case
  // recursively sort two sub arrays ("divide")
  int mid     = arr.length / 2
  int[] left  = mergeSort(arr[0..mid])
  int[] right = mergeSort(arr[(mid+1)..arr.length])
  // merge the left and right arrays ("conquer")
  int l = 0, r = 0
  for (int i = 0; i < arr.length; i++):
    // if at the end of left or right array
    if (r >= right.length):       arr[i] = left[l],  l++
    else if (l >= left.length):   arr[i] = right[r], r++
    // find the minimum of the left and right array
    else if (left[l] < right[r]): arr[i] = left[l],  l++
    else:                         arr[i] = right[r], r++
  // return the merged and sorted array
  return arr
```

Quick sort

Quick sort

- Quick sort is also classified as a "divide and conquer" algorithm, splitting up the list just like before
- Here is the basic idea:
  - Pick an element, called a **pivot**, from the list
  - Create two sublists
    - One contains all elements ≤ **pivot**
    - The other contains all elements > **pivot**
  - Recursively call quicksort on both sublists
  - Concatenate the left sublist with the **pivot** and the right sublist, then return it

Quick sort: step-by-step

s([3])	→	[] [3]
s([9])	→	s([]) + [0] + s([3]) → [0,3]
s([9,3,6,0])	→	s([0,0]) + [6] + s([9])
The Stack	→	[0,3,6,9]

Quick sort code

```
int[] quickSort(int[] arr):
  // base case
  if (arr.length <= 1): return arr

  // select a pivot and create two sublists ("divide")
  // NOTE: pivot can be any element (I'll use the middle)
  int pivot = arr[arr.length / 2];
  int[] left = [], right = []
  for (int element : arr):
    if (element <= pivot):
      left.add(element)
    else:
      right.add(element)

  // recursively sort each sublist and return the
  // concatenated result ("conquer")
  return quickSort(left) + [pivot] + quickSort(right)
```

Why do we care?

- Quick sort and merge sort provide more efficient ways to sort large lists

- Remember, it took **n^2** computational steps to complete the old school routines
  - Now we only take **n*log(n)** steps (on average) using our "divide and conquer" algorithms

- Now we will revisit our "Sorting Efficiently" labs to test out the new sorts!