CS1332 Fall 2018 - Week 10

10/22/18

Topics

I. Iterative Sorts Continued

A. Selection Sort

1. Premise - *selects* the maximum element of the subarray and swaps it with the last element in the subarray (can also be done by selecting the minimum and swapping it with the first element in the subarray)
2. Algorithm
    a) n: loop from end to 1 (marks the end of the subarray)
        (1) loop from 0 to n and find the max element
        (2) swap the max with the element at index n
3. Example (5 iterations) - max in subarray, swapped, **sorted**, instability

idx	0	1	2	3	4	5	6	7	8	
	5	7	8	9	3	6A	4	6B	2	n=8
	5	7	8	2	3	6A	4	6B	9	
	5	7	8	2	3	6A	4	6B	**9**	
	5	7	8	2	3	6A	4	6B	**9**	n=7
	5	7	6B	2	3	6A	4	8	**9**	
	5	7	6B	2	3	6A	4	**8**	**9**	
	5	7	6B	2	3	6A	4	**8**	**9**	n=6
	5	4	6B	2	3	6A	7	**8**	**9**	
	5	4	6B	2	3	6A	**7**	**8**	**9**	
	5	4	6B	2	3	6A	**7**	**8**	**9**	n=5
	5	4	6A	2	3	6B	**7**	**8**	**9**	
	5	4	6A	2	3	**6B**	**7**	**8**	**9**	
	5	4	6A	2	3	**6B**	**7**	**8**	**9**	n=4
	5	4	3	2	6A	**6B**	**7**	**8**	**9**	
	5	4	3	2	**6A**	**6B**	**7**	**8**	**9**	

. . .

4. Qualities - unstable, not adaptive, in-place, best & worst case $O(n^2)$
    a) What is it good for? It minimizes swaps, so it would be the best to use if for some reason writing to memory was very expensive.

B. Cocktail Shaker Sort

1. Premise - performs bubble sort twice, “shakes” the largest to the end and the smallest to the beginning
2. Algorithm
    a) while swaps are still needed
        (1) bubble sort forwards (move the largest to the end)
        (2) bubble sort backwards (move the smallest to the front)
3. Example (one iteration) - in final position

```
Initial array
     5    7    8    9    3    6A   4    6B   2
Bubble sort forwards (intermediate swaps omitted):
     5    7    8    3    6A   4    6B   2    9
Bubble sort backwards (intermediate swaps omitted):
     2    5    7    8    9    3    6A   4    6B
```

4. Qualities - stable, adaptive (with bubble sort optimization), in-place
    a) best case O(n), worst case O(n^2)
5. How is this different from just doing bubble sort? Cocktail shaker moves large and small values quickly. For example, see how long it takes bubble vs. cocktail shaker to sort this array: [2 3 4 5 6 7 8 9 1]

II. Divide & Conquer Sorts

A. Merge Sort

1. Premise - sort subarrays and then *merge* them together
2. Algorithm
    a) Create new arrays to house the left and right halves of the array
    b) Recursively call merge sort on these halves
        (1) Base case: return when the subarray is of length 1
    c) Merge the subarrays
        (1) Have one pointer, i, start at the beginning of the left array and another, j, start at the beginning of the right array
        (2) Compare the data at i with the data at j
            (a) Take whichever is smaller and put it back into the original array, increment the pointer you took from
            (b) If the values are equal, take the one from the left subarray to maintain stability
            (c) If you empty an entire subarray, you can directly add the remaining elements from the other subarray to the original array without performing any comparisons
3. Example (full sort)

```
Dividing   [5    7    8    9    3    6A   4    6B   2]
           [5    7    8    9]   [3   6A   4    6B   2]
           [5    7]   [8   9]   [3   6A]  [4   6B   2]
           [5]  [7]  [8]  [9]  [3]  [6A] [4]  [6B   2]
                                          [6B] [2]
Merging                                   [2    6B]
           [5    7]   [8   9]   [3   6A]  [2   4    6B]
           [5    7    8    9]   [2   3    4    6A   6B]
           [2    3    4    5    6A   6B   7    8    9]
```

4. Qualities - stable, not adaptive, out-of-place
    a) Best & worst case O(n log n) - log n to divide array, n to merge

10/24/18

Announcements

I. How we are counting homework 01?
    A. If you did HW01 and it is higher than your lowest homework grade, that lowest homework grade will be replaced with the grade you got on HW01.
    B. If you did not do HW01, or it is not higher than your lowest homework grade, the average of all your homeworks will be taken, and that average will replace your lowest homework grade.

Topics

I. Divide and Conquer Sorts Continued
    A. Merge Sort Review
        1. See the Saikrishna Slides > Sorting2.pdf > pages 55-57 for the pseudocode we reviewed in class
            a) Page 55
                (1) lines 1-4: copy the two halves of the array
                (2) lines 5-6: perform recursive calls on both halves
                (3) lines 7-9: set indices for merging
            b) Page 56
                (1) merges data into the original array only while both subarrays are not empty
            c) Page 57
                (1) handles emptying the other subarray if one became empty in the loop from page 56
    B. Quicksort
        1. Premise - *quickly* places each element in its final correct position by sorting the data relative to some random pivot value for each iteration
        2. How to do it in-place - the recursive method should take in a `start` and `end` index and pass the same array through each call
        3. Algorithm
            a) calculate the index of your random pivot between indices start and end inclusive → swap the pivot with the element at `start`
            b) initialize i to `start` + 1 and j to `end`
            c) while i has not crossed j
                (1) move i until it has crossed j or the data at i is > pivot
                (2) move j until it has crossed i or the data at j is < pivot
                (3) if i and j have not crossed
                    (a) swap arr[i] and arr[j]
                    (b) increment i, decrement j
            d) recursively call quicksort on left side (`start` to j-1)
            e) recursively call quicksort on the right side (j+1 to `end`)
        4. Example (one iteration) - **pivot**, swapped

```
select the pivot (random index between start & end)
      4     8     6A    2     5     7     9     6B    3
swap the pivot to the beginning
      8     4     6A    2     5     7     9     6B    3
initialize i and j
      8     4     6A    2     5     7     9     6B    3
            i                                         j
move i until it is greater than pivot, or crosses j
      8     4     6A    2     5     7     9     6B    3
            i →   i →   i →   i →   i →   i           j
move j until it is less than pivot, or crosses i
      8     4     6A    2     5     7     9     6B    3
                                          i           j
since i and j have not crossed, we swap i and j
      8     4     6A    2     5     7     3     6B    9
                                          i           j

after a swap, i++ and j--
      8     4     6A    2     5     7     3     6B    9
                                                ij

move i
      8     4     6A    2     5     7     3     6B    9
                                                i →   i
                                                j
i and j have crossed, pivot belongs at j
      6B    4     6A    2     5     7     3     8     9

call quicksort on left (start to j-1) & right (j+1 to end)
      6B    4     6A    2     5     7     3     8     9
      [                 L                 ]           [R]
```

4. Qualities - unstable, not adaptive, in-place
    a) Best case (good pivots) - O(n log n)
    b) Worst case (bad pivots, min or max) - $O(n^2)$

10/26/18

Topics

I. LSD (Least Significant Digit) Radix Sort
    A. See the Saikrishna Slides > Sorting2.pdf > pages 182-183 for the pseudocode
    B. Premise - repeatedly sort integers by each digit starting at the LSD
    C. Algorithm
        1. instantiate an array of linked lists
            a) 9 buckets/LLs for all positives: 0...9
            b) 19 buckets for positive and negative numbers: -9...0...9

2. for as many times as the number of digits (k) in the largest number by magnitude (eg. radix on [1 2 3 -133] would loop 3 times)
    a) iterate through the array and add values to their respective buckets based on which digit we're currently looking at
    b) iterate through the buckets and add the data back into the array

D. Considerations for finding the largest/longest value
    1. Make sure you are finding the largest by MAGNITUDE
    2. If you choose to do Math.abs(), make sure you account for the minimum integer value as well since Math.abs(MIN_VALUE) == MIN_VALUE

E. Considerations for finding the current digit of each value
    1. You'll need to divide the number by something and mod the number by something. HINT: ones digit is obtained through x % 10, tens digit is obtained through (x / 10) % 10 … figure out the rest :)
    2. You should not be using the pow function or any type of pow function to calculate the number you mod or divide by for every number

F. Example (full sort) - the chart represents the buckets, the top of a cell is the front

```
17   743B 672  780  917  743  623  288  432  281  76
```

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
780	281	672 432	743B 743 623			76	17 917	288	

```
780  281  672  432  743B 743  623  76   17   917  288
```

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	17 917	623	432	743B 743			672 76	780 281 288	

```
17   917  623  432  743B 743  672  76   780  281  288
```

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
17 76		281 288		432		623 672	743B 743 780		917

```
17   76   281  288  432  623  672  743B 743  780  917
```

G. Qualities - stable, not adaptable, out of place
    1. Best & worst case O(kn) - k is the number of digits in the largest number

II. K-Select (Quickselect)

A. Premise - you want to find a specific value in an array, eg. the smallest, second smallest, third smallest, etc. where you would know which index it would be at if the array was sorted
    1. For example, the smallest element will always be at index 0, the second smallest at index 1, etc.; the kth smallest element is at index k-1

B. Algorithm - italicized steps are the exact same as quicksort
    1. *calculate the index of your random pivot between indices start and end inclusive → swap the pivot with the element at `start`*
    2. *initialize i to `start` + 1 and j to `end`*
    3. *while i has not crossed j*
        a) *move i until it has crossed j or the data at i is > pivot*
        b) *move j until it has crossed i or the data at j is < pivot*
        c) *if i and j have not crossed*
            (1) *swap arr[i] and arr[j]*
            (2) *increment i, decrement j*
    4. if j is equal to k-1 → return j, we found it!
    5. if j > k-1 → recursively call quicksort on left side (`start` to j-1)
    6. if j < k-1 → recursively call quicksort on the right side (j+1 to `end`)

C. Qualities - unstable, in place
    1. Best case O(n) - different from best case quicksort because instead of recursing on both halves for each iteration, you only recurse on one half
    2. Worst case O(n^2) - bad pivots