4.1 mergesort and quicksort

Tuesday, September 20, 2022

Reminders

- test 1 in lab on Thursday
- lab 4 will be partnered

TODAY

big-O review
selection sort
merge sort
quick sort
runtimes : worst case, best case, expected

Review classes of algorithms and big-O

1. What class grows proportionally to the size of the problem? linear $O(n)$
2. What's the fastest-running class of algorithms? constant $O(1)$
3. What's an example of a quadratic algorithm? $O(n^2)$, for { · for{...}} selection sort, bubblesort
4. Which class of algorithms is the slowest-running? factorial $O(n!)$
5. What class do the fastest sorting algorithms fall into? comparison-based sorting $O(n \log n)$

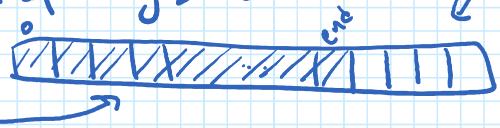
Selection Sort :

pseudocode

```
selectSort(array, size)
    for i = size - 1 down to 1
        indexOfMax = findMax(array, i)
        swap(array, i, indexOfMax)
```

n-1 iterations

```
findMax(array, end) // finds index of largest element in array
    indexOfMax = 0
    for i = 0 to end
        if array[i] > array[indexOfMax]
            indexOfMax = i
```



doesn't look beyond index "end"

return indexOfMax

Observations

- selectionsort is $O(n^2)$
- selectionsort is in-place: it doesn't use extra memory, it just edits the original array

Mergesort

An example of a divide-and-conquer algorithm.

Recursive structure:

- base case, where we solve the problem without recursion
- recursive cases, where we use the same approach on smaller subproblems

CONQUER by combining results from separate subproblems to solve the overall problem.

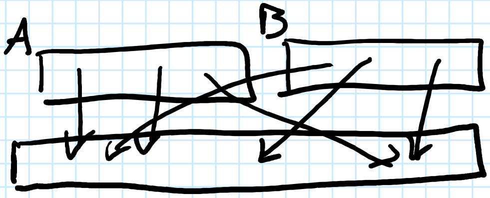
```
mergeSort(array, size):
    if size < 2:                          BASE CASE
        return // you're all done! that array is definitely sorted
    copy the first half of the array into a new array B      }
    copy the second half of the array into a new array C     } divide
    mergeSort(B,size/2)                                      }
    mergeSort(C,size/2)                                      }

    merge(B,size/2,C,size/2, array)       conquer
    // helper function "merge" takes two arrays and puts them
    // back in sorted order in the original array.

merge(A, sizeA, B, sizeB, C):    // A and B are sorted, and C has size big enough
    i=0, j=0, k=0 // indices for A, B, and C, respectively      to store A & B's elements
```

```
merge(A, sizeA, B, sizeB, C):
    i=0, j=0, k=0 // indices for A, B, and C, respectively
    while i<sizeA and j<sizeB:
        if A[i] ≤B[j]:
            C[k++] = A[i++] // this uses the values k and i, then increments each of them
        else
            C[k++] = B[j++]
    while i<sizeA: // to handle the leftover items if they are in A
        C[k++] = A[i++]
    while j<sizeB: // ditto
        C[k++] = B[j++]
```

// A and B are <u>sorted</u>, and C has size big enough to store A & B's elements



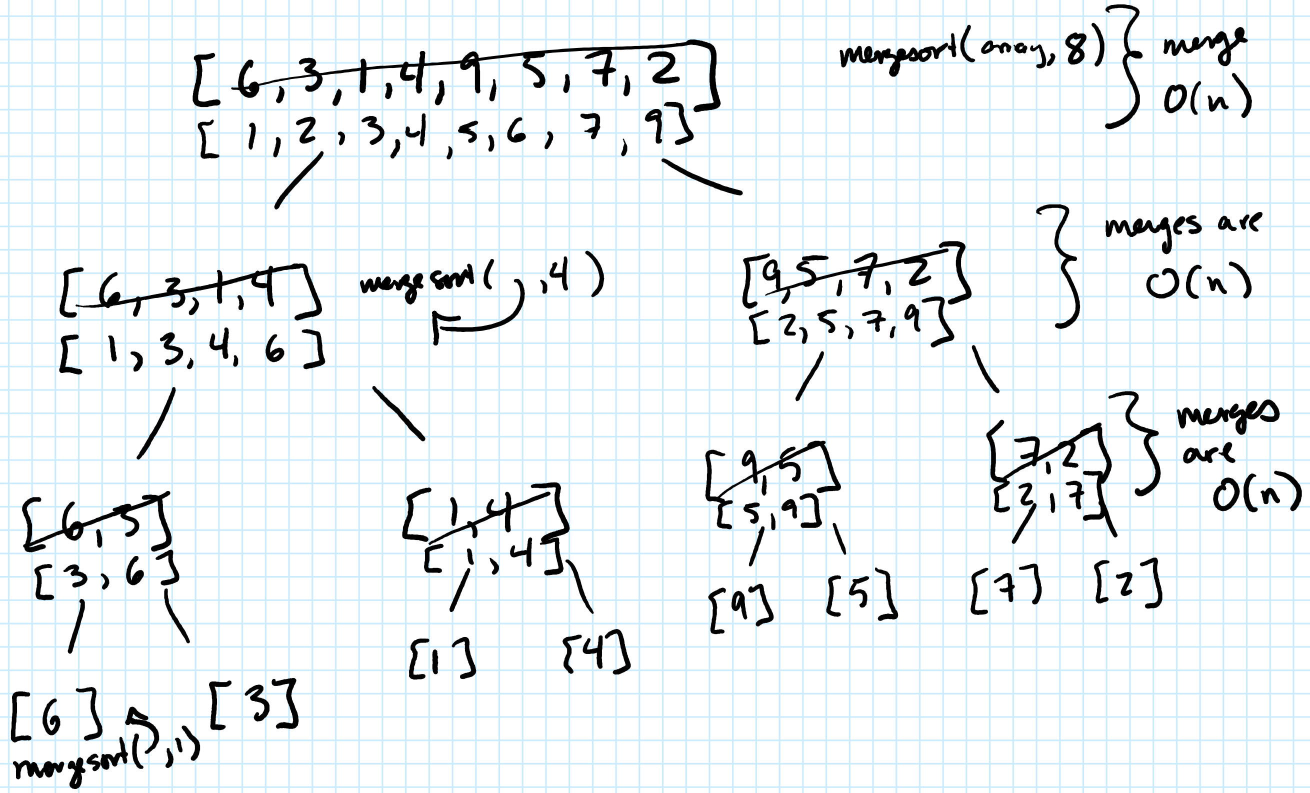
example merge(A, 4, B, 4, C)

	0	1	2	3	4
A:	1	3	4	6	
	~~i~~	~~i~~	~~i~~	~~i~~	i

	0	1	2	3	4
B	2	5	7	9	
	~~j~~	~~j~~	~~j~~	~~j~~	j

sizeA = 4
sizeB = 4

C	1	2	3	4	5	6	7	9	
	0	1	2	3	4	5	6	7	8
	~~k~~	~~k~~	~~k~~	~~k~~	~~k~~	~~k~~	~~k~~	~~k~~	k



runtime: merge(A, sizeA, B, sizeB, C) is O(n)

n = sizeA + sizeB

each level of tree is $O(n)$ work

There are $\log_2 n$ many levels.

Mergesort is $O(n \log n)$.

Observations:

- mergesort is $O(n \log_2 n)$

mergesort uses lots of extra memory to make all the smaller copies of pieces of the array

Can we do better - is there a sorting algorithm that's both fast <u>and</u> in-place?

Quicksort

another divide-and-conquer

To sort the whole array: quicksort(array, 0, size-1)

```
quickSort(array, i, j): // sorts the array[i,...,j]
    if j ≤ i:
        return // region to sort is ≤1 element, so it's definitely sorted!
    k = partition(array, i, j)   <- divide
    quicksort(array, i, k-1)   } conquer
    quicksort(array, k+1, j)   }

partition(array, left, right):
// rearranges array[left,...,right] using a pivot element
// should return the index where pivot element ends up
    pivot = right
    right --
    do:
        if array[left] ≤ array[pivot]:
            left++
        else if array[right] ≥ array[pivot]:
            right--
        else:
            swap(array,left,right)
```

Idea: Keep incrementing 'left' and decrementing 'right' until we find values that should be swapped.

```
        else if array[right] ≥ array[pivot]:
            right--
        else:
            swap(array,left,right)
while (left ≤ right)
swap(array,left,pivot) // put the pivot element into its place
return left
```

should be swapped.

quicksort(, 0, 7)

partition(, 0, 7)

i at 0, j at 7

0 1 2 3 4 5 6 7

[6, 3, 1, 4, 7, 2, 9, 5]

left right ~~right~~ ~~right~~

pivot = 7

after 1st swap

0 1 2 3 4 5 6 7 = pivot

[2, 3, 1, 4, 7, 6, 9, 5]

~~left~~ ~~left~~ ~~left~~ ~~left~~ left ~~right~~ ~~right~~

final array:

0 1 2 3 4 5 6 7

[2, 3, 1, 4, 5, 6, 9, 7]

left right = 4

partition returns 4

i=0 k=4 j=7

quicksort(array, 0, 3)

quicksort(array, 5, 7)

Some observations:

- memory is in-place (never makes extra copies of array)
- After running partition, the array looks like:

$$\underbrace{\left[\text{all values} \le \text{pivot value}\right.}_{\text{not yet sorted}} \quad \boxed{\text{pivot value}} \quad \underbrace{\left.\text{all values} \ge \text{pivot value}\right]}_{\text{not yet sorted}}$$

- Each call to partition takes linear time (with respect to the piece of the array it's called on).