Sorting

cs151

Nov 9

Midterm 2

average 80.1   std dev 12

	Average deduction
1	-3.9
2	-3.3
3	-0.5
4	-4
5	-1.2
6	-2.7
7	-1.8
8	-2.1

Time

80 minutes + 20 for startup/submit

4 people ran more than 10 minutes long

Handling

Sorting

- public Comparable[] sort(Comprable[] arra)
  - change the order of the items in arra
  - All examples will use integers but same statements apply to any Comparable object
  - ideally, do this "in place".
    - That is do not use any extra memory

- First 3 sort techniques we have already discussed

Sorting

Offer N followed by Poll N is sorting!!!!

- PQ on unordered == Selection Sort
- PQ on ordered == Insertion Sort
- PQ on Heap == Heap Sort

Selection Sort

- Selection-sort:
  - in place algorithm given an array with N items:
    - step 1: find the min from 0..(N-1) in array and swap with item in position 0
    - step 2: find min from 1..(N-1) in array and swap with item in position 1.
    - etc
- priority queue implemented with an unsorted array / arrayList / …
- Time:
  - $O(n^2)$
    - In terms of priority Q, can split this into two phases
      - insertion == O(N)
      - polling == $O(N^2)$

Selection Sort — Example

Phase 1 Inserting	Inserting		
a	7	(7)	1
b	4	[7,4]	1
…			
g		[7,4,8,2,5,3,9]	
Phase 2	Polling		
a	[2]	[7,4,8,5,3,9]	search=4, shift=3
b	[2,3]	[7,4,8,5,9]	search=5, shift=1
c	[2,3,4]	[7,8,5,9]	search=2 shift=3
d	[2,3,4,5]	[7,8,9]	search=3, shift=1
e	[2,3,4,5,7]	[8,9]	search=1, shift=2
f	[2,3,4,5,7,8]	[9]	search=1, shift=1
g	[2,3,4,5,7,8,9]	[]	search=1

Insertion Sort

- Insertion-sort
  - in-place algorithm
    - public Comparable[] sort(Comprable[] arra)
    - Step 0: start with item in position 0. Now the items in positions 0..0 are sorted
    - Step 1: look at item in position 1. Compare it to item in 0. If p1 is smaller, then swap. the items in position 0..1 are sorted with respect to each other
    - Step 2: determine where item in p2 should go in sorted list 0..N. If needed, For instance, bigger than 0 but smaller than 1. Make a space: save p1 into tmp. Shifting p1 into p2. Then put tmp into p1. Now the item in 0..2 are sorted.
    - Step N:

- Priority queue implemented with a sorted array/ ArrayList / …
- Time:
  - $O(n^2)$
  - In terms of PQ
    - Add:$O(n^2)$
    - Remove: $O(n)$
  - Generally faster than selection sort

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,7,8)
(e)	5	(2,4,5,7,8)
(f)	3	(2,3,4,5,7,8)
(g)	9	(2,3,4,5,7,8,9)

Phase 2 — polling

(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Heap Sort

- Heap-sort:
  - Insertion — no more than $\log_2(n)$ steps per insertion
  - Deletion — no more than $\log_2(n)$ steps per deletion
- priority queue is implemented with a heap

- Time:
  - Add:$O(n * \log_2(n))$ — doable in $O(n)$.
  - Remove: $O(n * \log_2(n))$
- Note: with a **lot of work** can do this without an additional array.

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,8,7)
(e)	5	(2,4,8,7,5)
(f)	3	(2,4,3,7,5,8)
(g)	9	(2,4,3,7,5,8,9)

Phase 2 — polling

(a)	(2)	(3,4,7,5,8,9)
(b)	(2,3)	(4,5,7,9,8)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Timing

size	selection	Insertion	Insertion (improved)	Heap
1000	16	15	11	2
2000	8	12	26	3
4000	24	23	20	5
8000	96	95	81	10
16000	370	378	315	17
32000	1585	1359	1218	36
64000	5771	5590	4605	77
128000	23087	21547	19849	161
256000				345
512000				1128
1024000				1973
2048000				3225
4096000				7577
8192000				18586

10000==1 second

anything below 1000
is very noisy

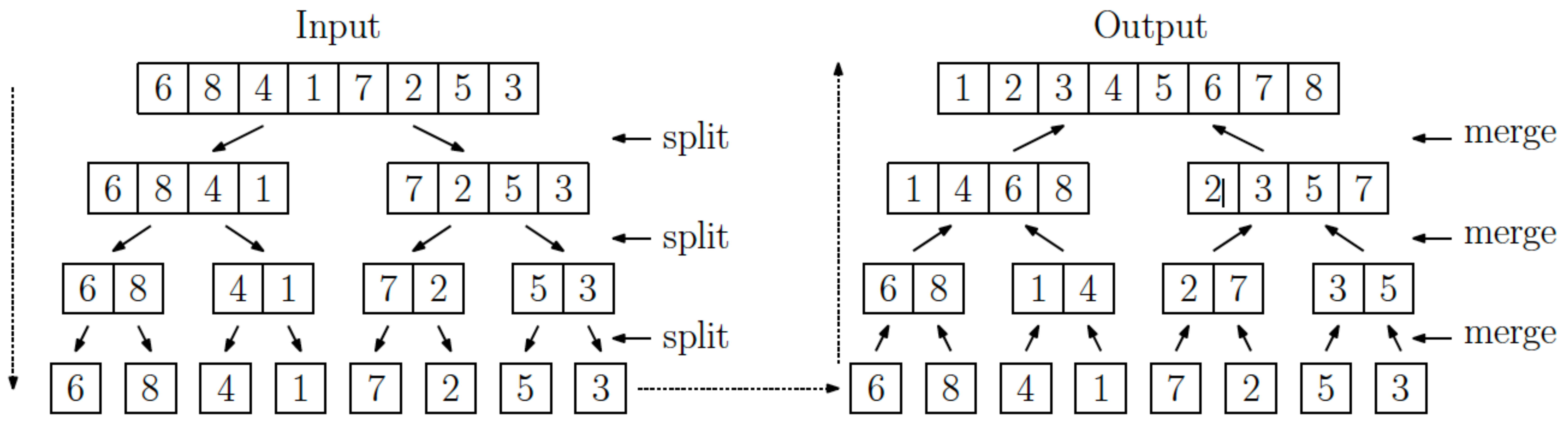
Divide-and-Conquer

- Divide – the problem (input) into smaller pieces
- Conquer – solve each piece individually
  - usually recursively
- Combine – the piecewise solutions into a global solution (if needed)

Merge Sort

- Sort a sequence of numbers A, $|A| = n$
- Base case: $|A| = 1$, then it's already sorted
- General
  - divide: split A into two halves, each of size $\frac{n}{2}$ ($\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$)
  - conquer: sort each half (by calling mergeSort recursively)
  - combine: merge the two sorted halves into a single sorted list

Example


Input
6 8 4 1 7 2 5 3
6 8 4 1
7 2 5 3
6 8
4 1
7 2
5 3
6
8
4
1
7
2
5
3
split
split
split
Output
1 2 3 4 5 6 7 8
1 4 6 8
2 3 5 7
6 8
1 4
2 7
3 5
6
8
4
1
7
2
5
3
merge
merge
merge

Algorithm

```
mergeSort(S):
  if S.size() <= 1 return
  else
    s1 = S[0,n/2]
    s2 = S[n/2+1,n-1]
    mergeSort(s1)
    mergeSort(s2)
    S = merge(s1, s2)
```

Selection Sort In-Place

- Given an array named toSort
- loc=0
- While loc < toSort.length-2
    - currentBestLoc = loc
    - for fnd=loc+1 upto toSort.length
        - if toSort[fnd] better than toSort[currentBestLoc]
            - currentBestLoc = fnd
    - if currentBestLoc != loc
        - swap items at currentBestLoc and Loc
    - loc = loc + 1

Insertion Sort — In Place

- Given an array toSort
  - for loc=1 upto toSort.length
    - p=0
    - while tosort[p] better than toSort[loc] and p<loc
      - p++
    - tmp=toSort[loc]
    - for mm=loc downto p+1
      - mm[loc]=mm[loc-1]
    - mm[p]=tmp