Sorting

Selection Sort

- 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 Queue
      - insertion == O(N)
      - polling == $O(N^2)$
- Space: In Place

Selection Sort Example

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

- Given an array of N items:
  - 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 Time:
  - Time: $O(n^2)$
    - In terms of Priority Queue
      - Add:$O(n^2)$
      - Remove: O(n)
    - Generally faster than selection sort
  - Space: In-Place

Insertion Sort Example

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

- Time:
  - Add:O(n * $\log_2(n)$) — doable in O(n).
  - Remove: O(n * $\log_2(n)$)
- Space:
  - Most naturally requires an extra array
    - with a **lot of work** can be in place.

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)	()

Complexity Analysis

	Selection Sort	Insertion Sort	Heap Sort
Add N Items	O(N)	O(N^2)	O(N * lg_2N)
Remove N Items	O(N^2)	O(N)	O(N * lg_2N)
Overall	O(N^2)	O(N^2) in practice better than selection	O(N * lg_2N)

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

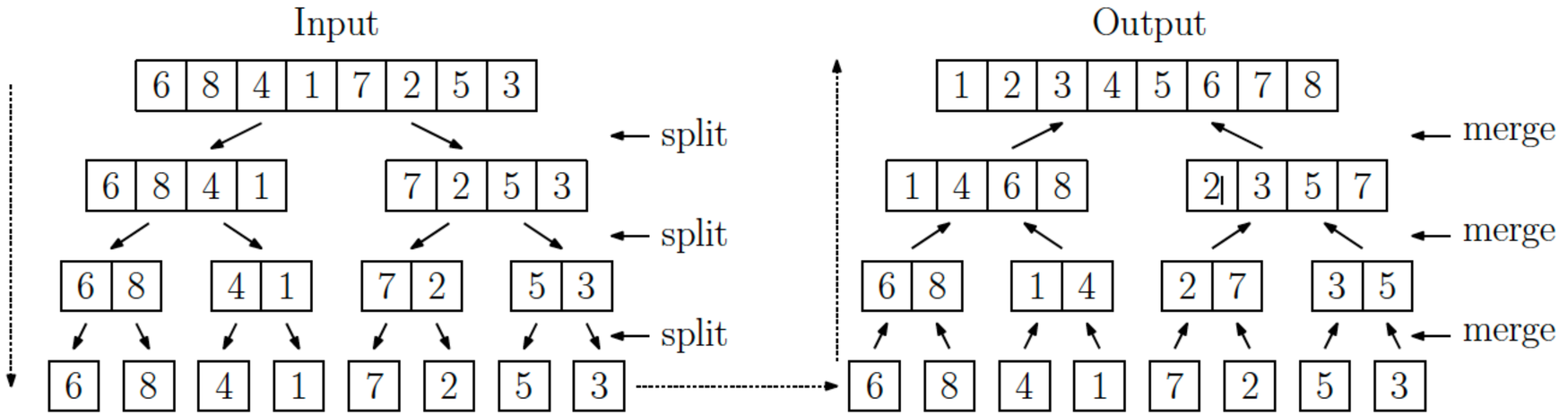
Divide-and-Conquer

- 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 of length N
- Base case: if N<=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
Output
6 8 4 1 7 2 5 3
split
split
split
merge
merge
merge
1 2 3 4 5 6 7 8
1 4 6 8
2 3 5 7
6 8
1 4
2 7
3 5

Algorithm

```
mergeSort(S):
   if S.size() <= 1 return
   else
      s1 = S[0,n/2]
      s2 = S[n/2+1,n-1]
      s1m = mergeSort(s1)
      s2m = mergeSort(s2)
      S = merge(s1m, s2m)
```

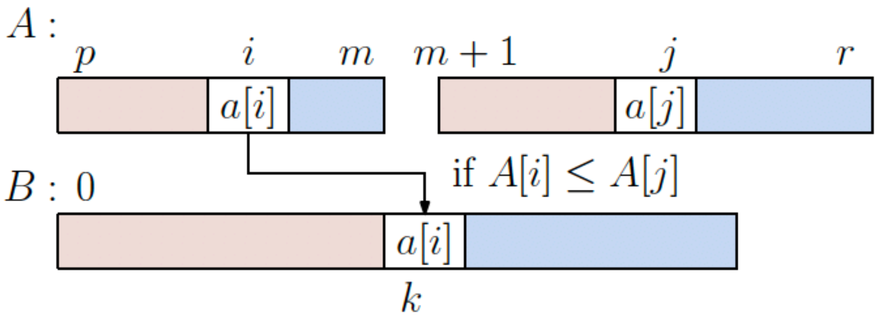
Timing

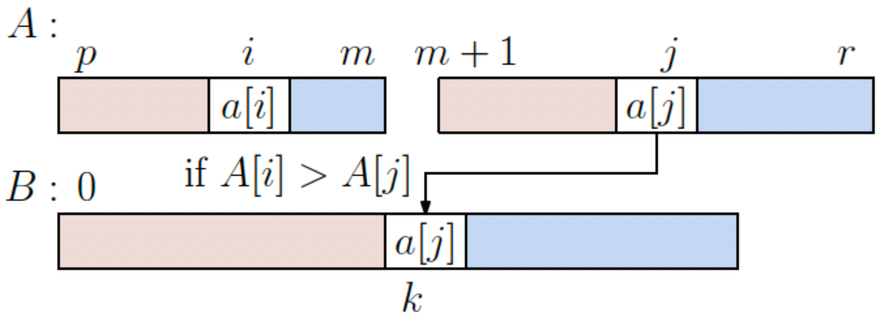
Table 1

size	selection	Insertion	Insertion	Heap	merge
1000	16	15	11	2	3
2000	8	12	26	3	3
4000	24	23	20	5	7
8000	96	95	81	10	13
16000	370	378	315	17	27
32000	1585	1359	1218	36	58
64000	5771	5590	4605	77	119
128000	23087	21547	19849	161	219
256000				345	372
512000				1128	776
1024000				1973	1631
2048000				3225	3822
4096000				7577	6772
8192000				18586	14159

In-place-ish Merge

- Making new lists is slow!
- How does one merge two sorted lists `A[p,…,m]` and `A[m+1,…,r]`?
- Use a temp array `B` and maintain two indices `i` and `j`, one for each subarray





Timing

Table 1

size	selection	Insertion	Insertion	Heap	merge	merge (improved)
1000	16	15	11	2	3	2
2000	8	12	26	3	3	3
4000	24	23	20	5	7	7
8000	96	95	81	10	13	9
16000	370	378	315	17	27	16
32000	1585	1359	1218	36	58	32
64000	5771	5590	4605	77	119	69
128000	23087	21547	19849	161	219	143
256000				345	372	294
512000				1128	776	563
1024000				1973	1631	1191
2048000				3225	3822	2412
4096000				7577	6772	5191
8192000				18586	14159	10282

Quick Sort

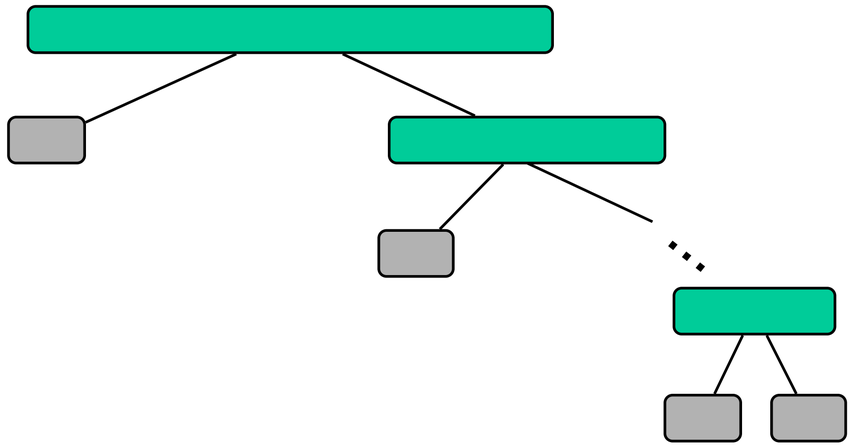
- Another divide-and conqueror sort
  - divide: pick a random element x (pivot) and partition into
    - L: $< x$
    - E: $= x$
    - G: $> x$
  - conquer: Quicksort L and G
  - combine: join L, E and G
    - When done in place, there is literally no work in the combine step.

Quicksort -- Example

Worst-case Running Time

- When the pivot is the min or max
  - one of L or G has size $n-1$
  - $T(n) = n + (n-1) + \ldots + 2 + 1 = O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



More Timing

Table 1

size	Insertion	Heap	merge (improved)	Quick
1000	11	2	2	1
2000	26	3	3	1
4000	20	5	7	2
8000	81	10	9	5
16000	315	17	16	13
32000	1218	36	32	30
64000	4605	77	69	59
128000	19849	161	143	108
256000		345	294	219
512000		1128	563	464
1024000		1973	1191	955
2048000		3225	2412	1989
4096000		7577	5191	4148
8192000		18586	10282	10101
16384000				17614
32768000				37291

Complexity Analysis

	Selection Sort	Insertion Sort	Heap Sort	Merge Sort	Quicksort
Add N Items	O(N)	O(N^2)	O(N * lg_2N)		
Remove N Items	O(N^2)	O(N)	O(N * lg_2N)		
Overall	O(N^2)	O(N^2)	O(N * lg_2N)	O(N * lg_2N)	O(N * lg_2N)

Sorting — Other Considerations

	Incremental Additions	Stability	Memory Usage
Selection Sort	No	Yes	N — in place
Insertion Sort	YES	YES	N
Heap Sort	NO	NO	N — with work
Merge Sort	NO	YES	2*N
Quicksort	NO	NO	N

Practice

```
    14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 16, 0
28, 10, 18, 37, 19, 35, 3, 31, 26, 22, 8, 26, 27, 11, 7, 36
```

For the data above, show all the steps of a merge sort and insertion sort.

Do each row of data independently.