More Sorting

cs151

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 (Improved) — In Place

- Given an array toSort
  - for loc=1 upto toSort.length
    - p=loc-1
    - pVal = toSort[loc]
    - while toSort[p] better than pVal and p>=0
      - toSort[p+1] = toSort[p]
      - p = p-1
    - toSort[p+1]=pVal

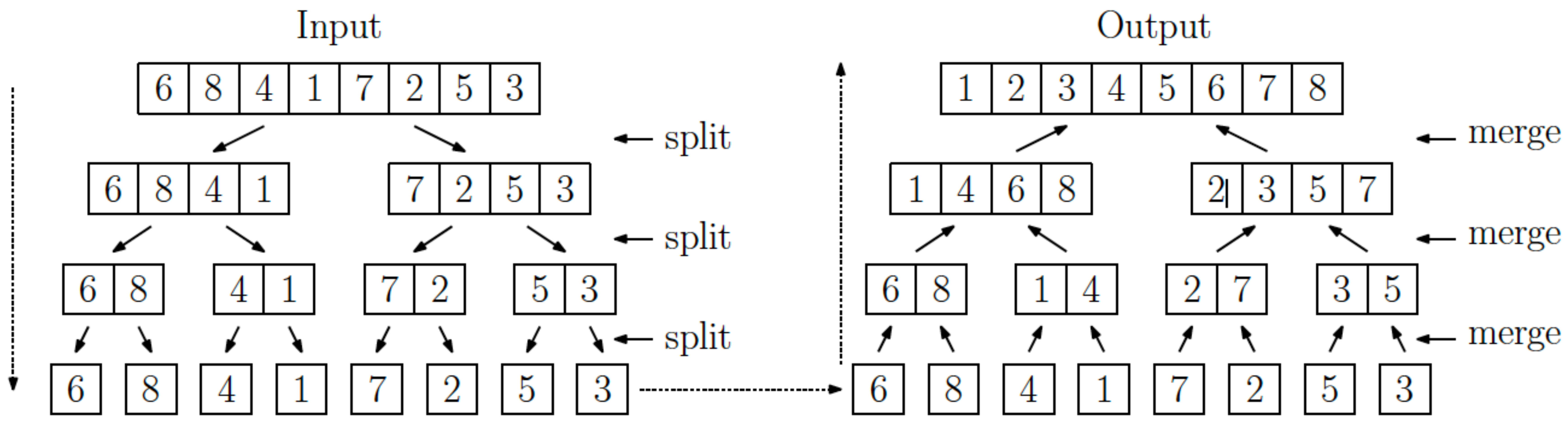
Divide-and-Conquer

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

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

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)
```

Merge Algorithm

- The key is the merging process
- How does one merge two sorted lists?
- Each element in $A \bigcup B$ is considered once
- $O(n)$

```
Merge two sorted lists
  let newlist - ()
  repeat until one list is empty
    compare first items in each
    remove lesser of first items from its list
          add removed item to newlist
copy remaining items from non-empty list into newlist


for example merge A=(1, 7 ) with B=(6,9) NL=()
  compare 1 with 6
    remove 1 from A and add it to NL
    result: A=(7), B=(6,9) NL=(1)
  compare 7 with 6
    remove 6 from B and add it to NL
    result: A=(7) B=(8), NL=(1,6)
  compare 7 and 9
    remove 7 from A and add it to NL
    result A=() B=(9) NL=(1,6,7)
  A is empty so stop
  Copy rest of B to NL
  final result NL=(1,6,7,9)
```

Merge (in Java)

```java
private int[] domerge(int[] list1, int[] list2) {
    int[] rtn = new int[list1.length + list2.length];
    int locr=0, loc1=0, loc2=0;
    while (loc1<list1.length && loc2<list2.length)  {
        if (list1[loc1] < list2[loc2])
        {
        rtn[locr++]=list1[loc1++];
        }
        else
        rtn[locr++]=list2[loc2++];
    }
    for (int i=loc1; i<list1.length; i++)
        rtn[locr++]=list1[i];
    for (int i=loc2; i<list2.length; i++)
        rtn[locr++]=list2[i];
    return rtn;
    }
```

MergeSort

```
public int[] mergesort(int[] list) {
    return doMergeSort(list, 0, list.length-1);
}

private int[] doMergeSort(int[] list, int strt, int eend)
{
if (eend==strt)
{
    int[] tmp = new int[1];
    tmp[0]=list[strt];
    return tmp;
}
if (eend<strt)
    return new int[0];
int mid = (strt+eend)/2;
return domerge(mergesort(list, strt, mid), mergesort(list, mid+1, eend));
}
```

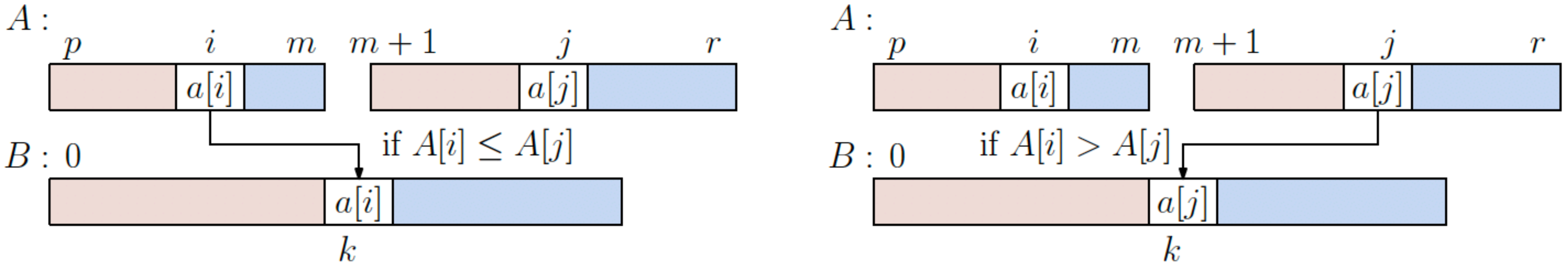
Timing

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



MergeSort using one temp array

```
private int[] array;
private int[] tempMergArr;
private int length;
public int[] mergesort3(int inputArr[]) {
    this.array = inputArr;
    this.length = inputArr.length;
    this.tempMergArr = new int[length];
    doMergeSort3(0, length - 1);
    return array;
}

private void doMergeSort3(int lowerIndex, int higherIndex) {

    if (lowerIndex < higherIndex) {
        int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
        // Below step sorts the left side of the array
        doMergeSort3(lowerIndex, middle);
        // Below step sorts the right side of the array
        doMergeSort3(middle + 1, higherIndex);
        // Now merge both sides
        mergeParts3(lowerIndex, middle, higherIndex);
    }
}
```

MergeSort with temp array

```
private void mergeParts3(int lowerIndex, int middle, int higherIndex) {

    for (int i = lowerIndex; i <= higherIndex; i++) {
        tempMergArr[i] = array[i];
    }
    int i = lowerIndex;
    int j = middle + 1;
    int k = lowerIndex;
    while (i <= middle && j <= higherIndex) {
        if (tempMergArr[i] <= tempMergArr[j]) {
            array[k] = tempMergArr[i];
            i++;
        } else {
            array[k] = tempMergArr[j];
            j++;
        }
        k++;
    }
    while (i <= middle) {
        array[k] = tempMergArr[i];
        k++;
        i++;
    }
}
```

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: sort L and G
  - combine: join L, E and G

Pseudo Code

```
quickSort(array, beginIndex, endIndex):
  if (endIndex<=beginIndex) return
  partIndex = partition(array, beginIndex, endIndex)
  quickSort(array, beginIndex, partIndex-1)
  quickSort(array, partIndex+1, endIndex)

partition(array, beginIndex, endIndex):
   elem=array[endIndex]
   loc = beginIndex
   for i from beginIndex To (endIndex-1):
     if (array[i] < elem):
        swap elements at i and loc
        loc=loc+1
   swap elements at endIndex and loc
   return loc
```

QS in action

Partition

(pink)

{7, 2, 9, 4, 3, 7, 1, 6}

Partition on 6

{2, 4, 3, 1}   6   {7, 7, 9}

Partition on 1   Partition on 9

{} 1 {2, 4, 3}   {7, 7} 9 {}

Partition on 3   Partition on 7

{2} 3 {4}   {7} 7 {}

Divide

Assemble

(aqua)

{2,3,4}   {7,7}

{1,2,3,4}   {7,7,9}

{1,2,3,4,6,7,7,9}

Conquer

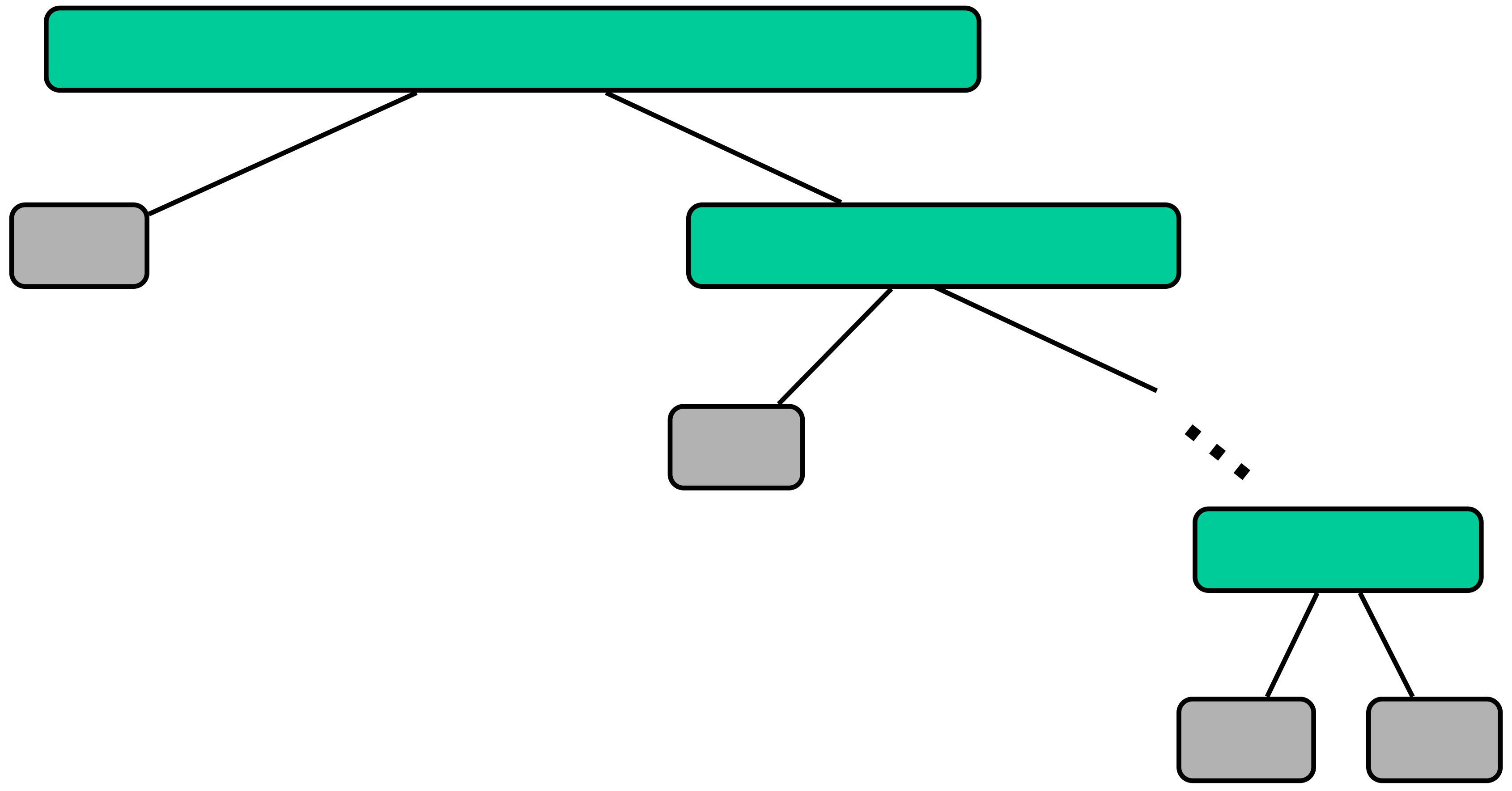
MergeSort, Quicksort, etc

- Quicksort does work on way down in recursion
  - Mergesort does work on way up
- Insertion sort does work on way down
  - Selection sort on way up
- Which one in faster Quick or Merge?

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	$\boldsymbol{n}$
1	$\boldsymbol{n} - 1$
…	…
$\boldsymbol{n} - 1$	1



In-place Quick Sort

- instead of three lists partition rearranges the input list
  - $L: [0, l-1]$
  - $E: [l,\ r]$
  - $G: [r+1,\ n-1]$
- Recursive calls on $[0,\ l-1]$ and $[r+1,\ n-1]$

QuickSort

```
private void quickSort(int arr[], int begin, int end)
{
    if ((end - begin) < 2) {
        int partitionIndex = partition(arr, begin, end);

        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
```

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

Quick and Merge

- Quicksort is reliably quicker than merge
- Quicksort does not need extra memory for auxiliary array

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 quicksort. For quicksort, always choose the leftmost element to partition

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection sort insertion sort		▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap sort		▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge sort quick sort		▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

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