CSE 373

Sorting 3: Merge Sort, Quick Sort

reading: Weiss Ch. 7

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:

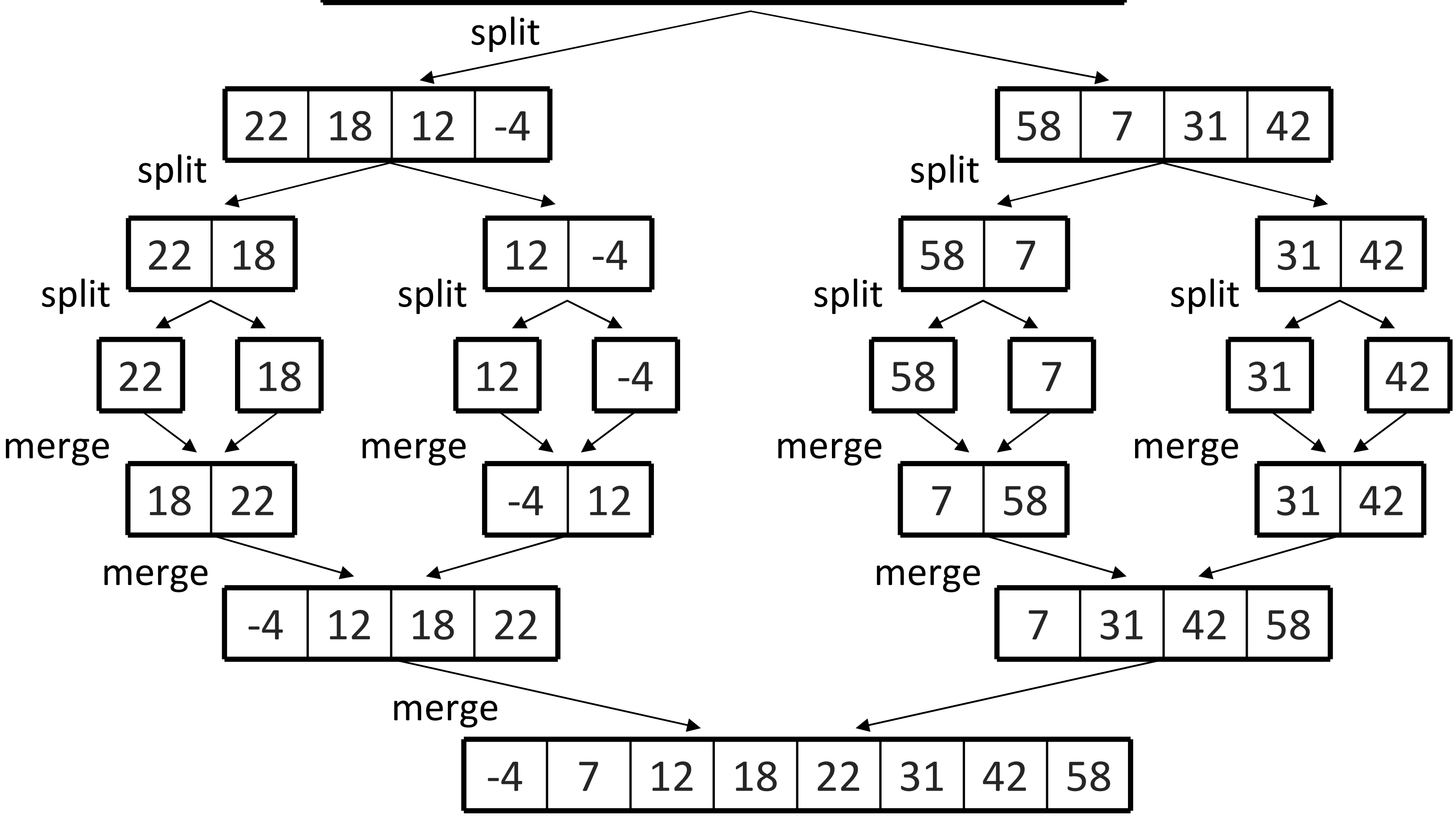
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945

- Runtime: O(*N* log *N*). Somewhat faster for asc/descending input.

Merge sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Merging sorted halves

Subarrays								Next include	Merged array							
0	1	2	3	0	1	2	3		0	1	2	3	4	5	6	7
14	32	67	76	**23**	41	58	85	14 from left	14							
i1				i2					i							
14	**32**	67	76	**23**	41	58	85	23 from right	14	23						
	i1			i2						i						
14	**32**	67	76	23	**41**	58	85	32 from left	14	23	32					
	i1				i2						i					
14	32	**67**	76	23	**41**	58	85	41 from right	14	23	32	41				
		i1			i2							i				
14	32	**67**	76	23	41	**58**	85	58 from right	14	23	32	41	58			
		i1				i2							i			
14	32	**67**	76	23	41	58	**85**	67 from left	14	23	32	41	58	67		
		i1					i2							i		
14	32	67	**76**	23	41	58	**85**	76 from left	14	23	32	41	58	67	76	
			i1				i2								i	
14	32	67	76	23	41	58	**85**	85 from right	14	23	32	41	58	67	76	85
							i2									i

Merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                                       int[] right) {
    int i1 = 0;   // index into left array
    int i2 = 0;   // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
           (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];   // take from right
            i2++;
        }
    }
}
```

Merge sort code

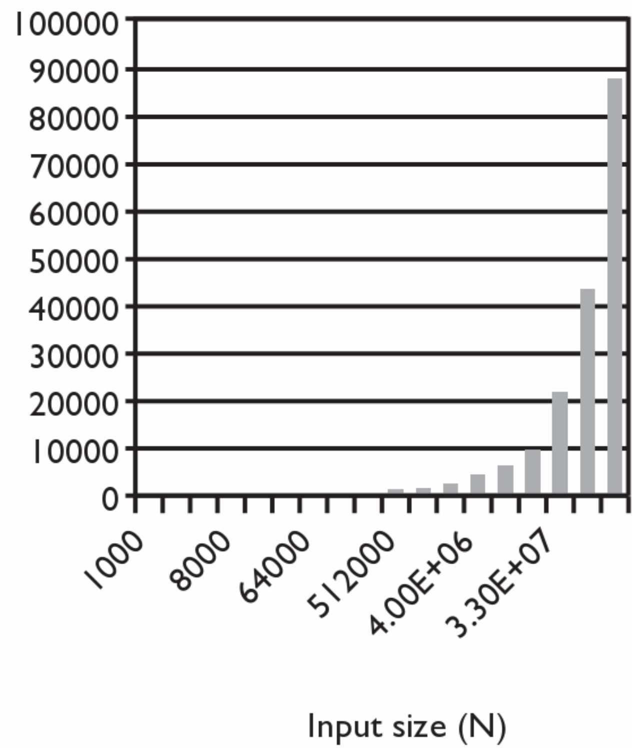
```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2,
                                          a.length);
        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

Merge sort runtime

- What is the complexity class (Big-Oh) of merge sort?

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



Recursive code and runtime

- It is difficult to look at a recursive method and estimate its runtime.
  - Let $T(N)$ = Runtime for merge sort to process an array of size N.

```
mergeSort(a, length=N):
   if N ≥ 2:
      left  = copyOfRange(0, N/2)        // approx. N/2 time
      right = copyOfRange(N/2, N)        // approx. N/2 time
      mergeSort(left)                    // T(N/2)
      mergeSort(right)                   // T(N/2)
      merge(a, left, right)              // approx. N time
```

- $T(N) = N/2 + N/2 + T(N/2) + T(N/2) + N$
- $T(N) = 2T(N/2) + 2N$, when $N \geq 2$
- $T(1) = 1$, when $N = 1$

- **recurrence relation:** An equation that recursively defines a sequence, specifically the runtime of a recursive algorithm.

Recurrence relations

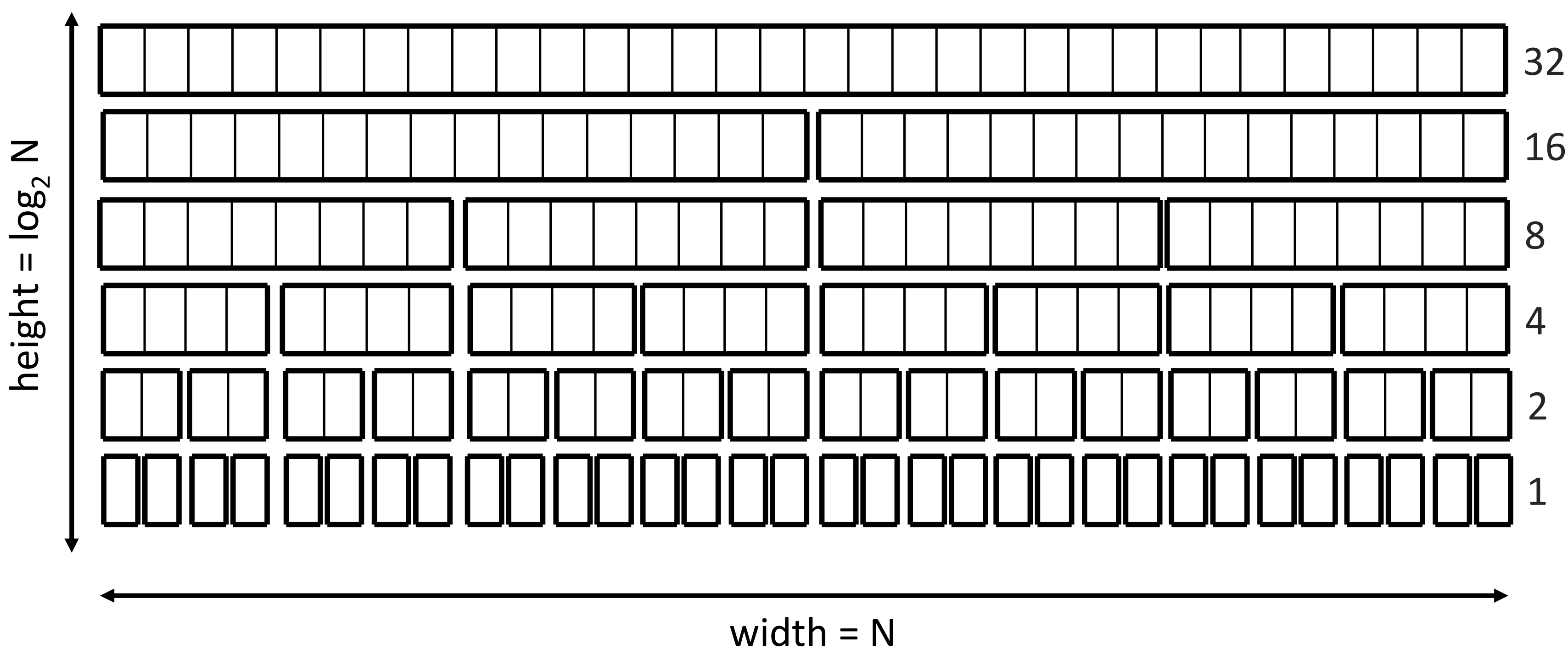
- Intuition about recurrence relations: Use *repeated substitution*.

- $T(N) = 2\, \mathbf{T(N/2)} + 2N$
  - $T(N/2) = 2\, T(N/4) + 2(N/2)$
- $T(N) = 2\, (\, 2\, T(N/4) + 2(N/2)\,) + 2N$
  $T(N) = 4\, \mathbf{T(N/4)} + 4N$
  - $T(N/4) = 2\, T(N/8) + 2(N/4)$
- $T(N) = 4\, (\, 2\, T(N/8) + 2(N/4)\,) + 4N$
  $T(N) = 8\, T(N/8) + 6N$
- $T(N) = 16\, T(N/16) + 8N$
- $T(N) = 32\, T(N/32) + 10N$
- …
- $T(N) = 2^k\, T(N/2^k) + 2kN$
  - At what value of k will we hit $T(1)$?

- Let $k = \log_2 N$.
- $T(N) = 2^{\log_2 N}\, T(N/2^{\log_2 N}) + 2\, (\log_2 N)\, N$
- $T(N) = N\, T(N/N) + 2\, N \log_2 N$
- $T(N) = N\, T(1) + 2\, N \log_2 N$
- $T(N) = 2\, N \log_2 N + N$
- $T(N) = O(N \log N)$

More runtime intuition

- Merge sort performs O(N) operations on each level. *(width)*
  - Each level splits the array in 2, so there are $\log_2 N$ levels. *(height)*
  - Product of these = $N * \log_2 N$ = O($N \log N$). *(area)*
  - Example: N = 32. Performs ~ $\log_2 32$ = 5 levels of N operations each:



Quick sort

- **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960

- Quick sort is another divide and conquer algorithm:
  - Choose one element in the list to be the pivot.
  - *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - *Conquer* by applying quick sort (recursively) to both partitions.

- Runtime: O(N log N) average, O(N^2) worst case.
  - Generally somewhat faster than merge sort.

Choosing a "pivot"

- The algorithm will work correctly no matter which element you choose as the pivot.
  - A simple implementation can just use the first element.

- But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  - What kind of value would be a good pivot? A bad one?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

Partitioning an array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until *i*,*j* meet):
  - Starting from *i* = 0, find an element a[*i*] ≥ pivot.
  - Starting from *j* = *N*-1, find an element a[*j*] ≤ pivot.
  - These elements are out of order, so swap a[*i*] and a[*j*].
- Swap the pivot back to index *i* to place it between the partitions.

index	0	1	2	3	4	5	6	7	8	9
value	6	1	4	9	0	3	5	2	7	8
	8 i							←	j	6
	2	i	→	→			j	8		
				5	i	→	9			
							6			9
	2	1	4	5	0	3	6	8	7	9

Quick sort example

index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65

32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
32	23	16	43	57	39	92	81	75	65	
32	23	16	43	57	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	43	57	39	pivot=32
39	23	16	43	57	**32**	swap to end
16	23	**39**	43	57	32	swap 39, 16
16	23	**32**	43	57	**39**	swap 32 back in

...

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

Quick sort code

```
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}
private static void quickSort(int[] a, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = a[min];
    swap(a, min, max);      // move pivot to end

    // partition the two sides of the array
    int middle = partition(a, min, max - 1, pivot);

    swap(a, middle, max);   // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
```

Partition code

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center
        // until we find a pair of out-of-order elements
        while (i <= j && a[i] < pivot) { i++; }
        while (i <= j && a[j] > pivot) { j--; }

        if (i <= j) {
            swap(a, i, j);
            i++;
            j--;
        }
    }

    return i;
}
```

Quick sort runtime

- Best-case analysis: If partition divides the array fairly evenly.
  - Let $T(N)$ = Runtime for quick sort to process an array of size N.

```
quickSort(a, length=N):
    swap pivot to end.              // O(1)
    mid = partition(a, pivot).      // approx. N time
    swap pivot to mid.              // O(1)
    quickSort(min, mid-1).          // T(N/2)  if left size ≈ N/2
    quickSort(mid+1, max).          // T(N/2)  if right size ≈ N/2
```

  - $T(N) = 2T(N/2) + k_1N + k_2(1)$ for some constants k_1, k_2
  - $T(N) = O(N \log N)$

- Worst-case: What if the pivot is chosen poorly?
  What is the runtime?
  - $T(N) = T(N\text{-}1) + T(1) + k_1N + k_2(1) = O(N^2)$

Choosing a better pivot

- Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  - does not partition the array into roughly-equal size chunks

- Alternative methods of picking a pivot:
  - *random:* Pick a random index from [*min* .. *max*]
  - *median-of-3:* look at left/middle/right elements and pick the one with the medium value of the three:
    - `a[min]`, `a[(max+min)/2]`, and `a[max]`
    - better performance than picking random numbers every time
    - provides near-optimal runtime for almost all input orderings

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**

Stable sorting

- **stable sort**: One that maintains relative order of "equal" elements.
  - important for secondary sorting, e.g.
    - sort by name, then sort again by age, then by salary, ...

- All of the N^2 sorts shown are stable, as is shell sort.
  - bubble, selection, insertion, shell
- Merge sort is stable.
- Quick sort is *not* stable.
  - The partitioning algorithm can reverse the order of "equal" elements.
  - For this reason, Java's Arrays/Collections.sort() use merge sort.

Unstable sort example

- Suppose you want to sort these points by Y first, then by X:
  - `[(4, 2), (5, 7), (3, 7), (3, 1)]`
- A stable sort like merge sort would do it this way:
  - `[(3, 1), (4, 2), (5, 7), (3, 7)]  sort by y`
  - `[(3, 1), (3, 7), (4, 2), (5, 7)]  sort by x`
  - Note that the relative order of (3, 1) and (3, 7) is maintained.
- Quick sort might leave them in the following state:
  - `[(3, 1), (4, 2), (5, 7), (3, 7)]  sort by y`
  - `[(3, 7), (3, 1), (4, 2), (5, 7)]  sort by x`
  - Note that the relative order of (3, 1) and (3, 7) has reversed.