## Sorting Classification

<table>
<thead>
<tr>
<th>In memory sorting</th>
<th>External sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison sorting ( \Omega(N \log N) )</td>
<td>Specialized Sorting</td>
</tr>
<tr>
<td>( O(N^2) )</td>
<td>( O(N \log N) )</td>
</tr>
<tr>
<td>( O(N) )</td>
<td># of disk accesses</td>
</tr>
</tbody>
</table>

- Bubble Sort
- Selection Sort
- Insertion Sort
- Shell Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Bucket Sort
- Radix Sort
- Simple External Merge Sort
- Variations
$O(n \log n)$ Comparison Sorting
Merge sort

- **merge sort**: orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining
  - Invented by John von Neumann in 1945

- Another "divide and conquer" algorithm
  - *divide* the list into two roughly equal parts
  - *conquer* by sorting the two parts
    - recursively divide each part in half, continuing until a part contains only one element (one element is sorted)
  - *combine* the two parts into one sorted list
Merge sort idea

- Divide the array into two halves.
- Recursively sort the two halves (using merge sort).
- Use merge to combine the two arrays.

mergeSort(0, n/2-1)
mergeSort(n/2, n-1)

merge(0, n/2, n-1)
<table>
<thead>
<tr>
<th></th>
<th>98</th>
<th>23</th>
<th>45</th>
<th>14</th>
<th>6</th>
<th>67</th>
<th>33</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td>6</td>
<td>67</td>
<td>33</td>
<td>42</td>
</tr>
</tbody>
</table>
Merge
98 23 45 14 6 67 33 42
98 23 45 14
98 23 45 14
98 23 45 14
23 98 14 45
14 23 45 98
Merge
Merge
Merging two sorted arrays

- **merge operation:**
  - Given two sorted arrays, *merge* operation produces a sorted array with all the elements of the two arrays

<table>
<thead>
<tr>
<th>A</th>
<th>6</th>
<th>13</th>
<th>18</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>18</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

Running time of *merge*: $O(n)$, where $n$ is the number of elements in the merged array.

When merging two sorted parts of the same array, we'll need a *temporary array* to store the merged whole.
public static void mergeSort(int[] a) {
    int[] temp = new int[a.length];
    mergeSort(a, temp, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] temp, int left, int right) {
    if (left >= right) {  // base case
        return;
    }
    int mid = (left + right) / 2;
    mergeSort(a, temp, left, mid);
    mergeSort(a, temp, mid + 1, right);
    // merge the sorted halves into a sorted whole
    merge(a, temp, left, right);
}
private static void merge(int[] a, int[] temp, int left, int right) {
    int mid = (left + right) / 2;
    int count = right - left + 1;

    int l = left;                  // counter indexes for L, R
    int r = mid + 1;

    // main loop to copy the halves into the temp array
    for (int i = 0; i < count; i++) {
        if (r > right) {           // finished right; use left
            temp[i] = a[l++];
        } else if (l > mid) {      // finished left; use right
            temp[i] = a[r++];
        } else if (a[l] < a[r]) {  // left is smaller (better)
            temp[i] = a[l++];
        } else {                   // right is smaller (better)
            temp[i] = a[r++];
        }
    }

    // copy sorted temp array back into main array
    for (int i = 0; i < count; i++) {
        a[left + i] = temp[i];
    }
}
Merge sort example 2

13  6  21  18  9  4  8  20
0  7

13  6  21  18  9  4  8  20
0  3

13  6  21  18  9  4  8  20
0  1  3

13  6  21  18  9  4  8  20
0  1  2  3

13  6  21  18  9  4  8  20
0  1  2  3

13  6  21  18  9  4  8  20
0  1  2  3

13  6  21  18  9  4  8  20
0  1  2  3

4  6  8  9  13  18  20  21
0  7

50
Merge sort runtime

- Let $T(n)$ be runtime of merge sort on $n$ items
  - $T(0) = 1$
  - $T(1) = 2*T(0) + 1$
  - $T(2) = 2*T(1) + 2$
  - $T(4) = 2*T(2) + 4$
  - $T(8) = 2*T(4) + 8$
  - $T(n/2) = 2*T(n/4) + n/2$
  - $T(n) = 2*T(n/2) + n$

- Substitute to solve for $T(n)$
Repeated Substitution Method

\[ T(n) = 2^k T(n/2^k) + kn \]

What is \( k \)? How many times can you cut \( n \) in half?

Setting \( k = \log_2 n \).

\[ T(n) = 2^\log n T(n/2^{\log n}) + (\log n) n \]
\[ T(n) = n \cdot T(n/n) + n \log n \]
\[ T(n) = n \cdot T(1) + n \log n \]
\[ T(n) = n \cdot 1 + n \log n \]
\[ T(n) = n + n \log n \]
\[ T(n) = O(n \log n) \]
Sorting practice problem

- Consider the following array of int values.

\[22, 11, 34, -5, 3, 40, 9, 16, 6\]

Write the contents of the array after all the recursive calls of merge sort have finished (before merging).
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

- another divide and conquer algorithm:
  - choose one element in the list to be the pivot (partition element)
  - divide the elements so that all elements less than the pivot are to its left and all greater are to its right
  - conquer by applying the quick sort algorithm (recursively) to both partitions
Quick sort, continued

- For correctness, it's okay to choose any pivot.

- For efficiency, one of the following is best case, the other worst case:
  - pivot partitions the list roughly in half
  - pivot is greatest or least element in list

- Which case above is best?

- We will divide the work into two methods:
  - `quickSort` – performs the recursive algorithm
  - `partition` – rearranges the elements into two partitions
Quick sort pseudo-code

Let \( S \) be the input set.

1. If number of elements in \( S \) is 0 or 1, then return.

2. Pick an element \( v \) in \( S \). Call \( v \) the pivot.

3. Partition remaining elements of \( S \) into two groups:
   - \( S_1 = \{ \text{elements } \leq v \} \)
   - \( S_2 = \{ \text{elements } \geq v \} \)

4. Return \( \{ \text{quicksort}(S_1), v, \text{quicksort}(S_2) \} \)
Quick sort illustrated

pick a pivot

partition

quicksort

combine
How to choose a pivot

- first element
  - bad if input is sorted or in reverse sorted order
  - bad if input is nearly sorted
  - variation: particular element (e.g. middle element)

- random element
  - even a malicious agent cannot arrange a bad input

- median of three elements
  - choose the median of the left, right, and center elements
The basic idea:

1. Move the pivot to the rightmost position.
2. Starting from the left, find an element ≥ pivot. Call the position i.
3. Starting from the right, find an element ≤ pivot. Call the position j.
4. Swap S[i] and S[j].

<table>
<thead>
<tr>
<th>8</th>
<th>1</th>
<th>4</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>
"Median of three" pivot

<table>
<thead>
<tr>
<th></th>
<th>9</th>
<th>17</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>21</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

pick pivot

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>17</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>21</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>j</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>17</th>
<th>21</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>j</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>8</th>
<th>12</th>
<th>17</th>
<th>21</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>j</td>
<td></td>
<td></td>
<td>7</td>
<td>i</td>
</tr>
</tbody>
</table>

swap S[i] with S[right]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>8</th>
<th>9</th>
<th>17</th>
<th>21</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>j</td>
</tr>
</tbody>
</table>
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);

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

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
Quick sort code, cont'd.

// 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) {
    i--;  j++;  // kludge because the loops pre-increment
    while (true) {
        // move index markers i,j toward center
        // until we find a pair of mis-partitioned elements
        do { i++; } while (i < j && a[i] < pivot);
        do { j--; } while (i < j && a[j] > pivot);

        if (i >= j) {
            break;
        } else {
            swap(a, i, j);
        }
    }

    return i;
}
Quick sort runtime

- Worst case: pivot is the smallest (or largest) element all the time (recurrence solution technique: telescoping)
  \[ T(n) = T(n-1) + cn \]
  \[ T(n-1) = T(n-2) + c(n-1) \]
  \[ T(n-2) = T(n-3) + c(n-2) \]
  ...
  \[ T(2) = T(1) + 2c \]

  \[ T(N) = T(1) + c \sum_{i=2}^{N} i = O(N^2) \]

- Best case: pivot is the median (recurrence solution technique: Master's Theorem)
  \[ T(n) = 2T(n/2) + cn \]
  \[ T(n) = cn \log n + n = O(n \log n) \]
## Runtime summary

<table>
<thead>
<tr>
<th></th>
<th>comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
| quick | average: $O(n \log n)$  
          worst: $O(n^2)$  |
Consider the following array of int values.

\[ [22, 11, 34, -5, 3, 40, 9, 16, 6] \]

Write the contents of the array after all the partitioning of quick sort has finished (before any recursive calls).

Assume that the median of three elements (first, middle, and last) is chosen as the pivot.
Consider the following array:

\[ 7, 17, 22, -1, 9, 6, 11, 35, -3 \]

Each of the following is a view of a sort-in-progress on the elements. Which sort is which?

(If the algorithm is a multiple-loop algorithm, the array is shown after a few of these loops have completed. If the algorithm is recursive, the array is shown after the recursive calls have finished on each sub-part of the array.)

Assume that the quick sort algorithm chooses the first element as its pivot at each pass.

(a) \([-3, -1, 6, 17, 9, 22, 11, 35, 7]\)
(b) \([-1, 7, 17, 22, -3, 6, 9, 11, 35]\)
(c) \([-1, 7, 6, 9, 11, -3, 17, 22, 35]\)
(d) \([-3, 6, -1, 7, 9, 17, 11, 35, 22]\)
(e) \([-1, 7, 17, 22, 9, 6, 11, 35, -3]\)
For the following questions, indicate which of the five sorting algorithms will successfully sort the elements in the least amount of time.

- The algorithm chosen should be the one that completes fastest, without crashing.
- Assume that the quick sort algorithm chooses the first element as its pivot at each pass.
- Assume stack overflow occurs on 5000+ stacked method calls.

(a) array size 2000, random order
(b) array size 500000, ascending order
(c) array size 100000, descending order
  - special constraint: no extra memory may be allocated! (O(1) storage)
(d) array size 1000000, random order
(e) array size 10000, ascending order
  - special constraint: no extra memory may be allocated! (O(1) storage)
External Sorting
Simple External Merge Sort

- Divide and conquer: divide the file into smaller, sorted subfiles (called runs) and merge runs

- Initialize:
  - Load chunk of data from file into RAM
  - Sort internally
  - Write sorted data (run) back to disk (in separate files)

- While we still have runs to sort:
  - Merge runs from previous pass into runs of twice the size (think merge() method from mergesort)
  - Repeat until you only have one run