**MERGE SORT**

*Algorithm, Complexity*

---

### Divide and Conquer

- **Strategy:**
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine the solutions to get overall solution

- **Very important strategy in Computer Science**
  - **Why?**
    - Makes problem manageable
    - Often, makes solution faster
    - Allows use of parallel resources to solve even faster

- **Examples:**
  - The basis for Google’s Map-Reduce
    - Searching through the world’s largest database
  - Sorting
### Divide and Conquer: Sorting Examples

- **Mergesort**
  - Divide array into two halves
  - Recursively sort left and right halves
  - Merge the two halves

- **Quicksort**
  - Partition array into *small* and *large* items
  - Recursively sort the two sets

---

### Mergesort: Intro

- Divide input into two at mid-point
- Conquer each side in turn (by recursively sorting)
- Merge the two sorted halves together
**Mergesort: Example**

- Divide input into two at mid-point
- Conquer each side (recursively sort)
- Merge the two sorted halves together

```
26  1  24  13  2  38  15  27
```

<table>
<thead>
<tr>
<th>Divide</th>
<th>Divide</th>
<th>Divide</th>
<th>Merge</th>
<th>Merge</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 1 24 13 2 38 15 27</td>
<td>26 1 24 13 2 38 15 27</td>
<td>26 1 24 13 2 38 15 27</td>
<td>1 26</td>
<td>1 13 24 26</td>
<td>1 2 13 15 24 26 27 38</td>
</tr>
</tbody>
</table>

**Merging Needs an Output Array**

- Compare smallest not-yet-merged element of each side
- Copy the smaller one, and advance that side’s pointer
- Copy back to original location!

```
1  13  24  26  2  15  27  38
```

```
1  2  13  15  24  26  27  38
```
Mergesort: Recursive Code

- Sorting an integer array:
  ```java
  private static void mergeSort(int[] A, int[] B, int left, int right) {
      if (left == right)
          return;
      int mid = (left + right)/2;
      mergeSort(A, B, left, mid);
      mergeSort(A, B, mid+1, right);
      merge(A, B, left, mid+1, right);
  }
  public static void mergeSort(int[] A) {
      int[] tmpArr = new int[A.length];
      mergeSort(A, tmpArr, 0, A.length - 1);
  }
  ```

- Look up merge routine in textbook

Mergesort Example: Iterative Version

- Try writing iterative version of mergesort on your own
Mergesort: Run-time

- Let $T(k)$ – run-time to mergesort an array of $k$ elements
  - $T(1) \leq c_0$
  - $T(N) \leq c_1 + 2T(N/2) + c_2N$
    $\leq 2T(N/2) + c_3N$  // simplify

- Solving the recurrence:
  - $T(N) \leq 2T(N/2) + c_3N$
    $\leq 2(2T(N/4) + c_4N/2) + c_3N$
    $= 4T(N/4) + 2c_4N/2 + c_3N$
    $\ldots$
    $\leq c_3N + 2c_4N/2 + 4c_5N/4 + \ldots + 2^{\log N}c_{\log N+3}T(N/2^{\log N})$
    $\leq c_3N + c_4N + c_5N + \ldots + c_{\log N+3}c_0N$
    $\leq c_{\max}N + c_{\max}N + c_{\max}N + \ldots + c_{\max}N$
    $= (\log N+1)c_{\max}N$
    $= O(N \log N)$

Divide and Conquer: Run-time Speedup

- Assume: each step itself takes constant time

- If each step:
  - Reduces the problem size by at least a constant factor
  - Complexity – $O(\log N)$: binary search

- If each step:
  - Divides the problem into a constant number of parts, and conquers these
  - Then combines the solutions in linear time
  - Complexity – $O(N \log N)$: mergesort

- If each step:
  - Divides the problem into a constant number of parts, and conquers these
  - Then combines the solutions in constant time
  - Complexity – $O(N)$: findMin
Does Run-time Depend on Input?


- Claim: run-time is $\Theta(N \log N)$ for all input
  - Let $T(k)$ – run-time to mergesort an array of $k$ elements
  - Can we derive a lower bound of $\Omega(N \log N)$ on $T(k)$?
  - Try on Piazza.

Mergesort: Properties

- Run-time – $\Theta(N \log N)$ for all input
  - Merging step always takes $\Theta(N)$ time $\Rightarrow$ sorting is $\Theta(N \log N)$

- Space complexity:
  - Uses linear extra memory – $O(N)$

- Additional work of copying to the temporary array (and back) throughout the algorithm slows the sort considerably
  - This cost is language dependent
    - Java (generic sort)
      - Comparisons may be expensive
      - Moving elements is cheap
    - C++ (generic sort)
      - Copying elements can be expensive
      - Comparing can be made cheap by compiler

- Quicksort – uses fewer moves, but some more comparisons