CSC 212: Data Structures and Abstractions  
11: Merge Sort

Marco Alvarez  
Department of Computer Science and Statistics  
University of Rhode Island  
Spring 2017

Motivation

- sorting with insertion sort is $O(n^2)$
- we can divide the array into two halves and sort them separately
- each subproblem could be sorted in $O(n^2/4)$
- sorting both halves will require a total of $O(n^2/2)$
- we need an additional operation to combine both solutions

Divide and Conquer

- **Divide** the problem into smaller subproblems

  - **Conquer** recursively
    - ... each subproblem

  - **Combine** Solutions

  ![Divide and Conquer Diagram]

Merge Sort

- **Divide** the array into two halves
  - just need to calculate the mid point

- **Conquer** Recursively each half
  - call Merge Sort on each half (i.e. solve 2 smaller problems)

- **Merge** Solutions
  - after both calls are finished, proceed to merge the solutions
Merge Sort: pseudocode

```plaintext
if (hi <= lo) return;

int mid = lo + (hi - lo) / 2;
mergesort(A, lo, mid);
mergesort(A, mid+1, hi);
merge(A, lo, mid, hi);
```

Merging two sorted arrays

![Merging two sorted arrays](image)

A secondary array is necessary to guarantee a linear time operation

Merge

```plaintext
1 void merge(int *A, int *aux, int lo, int mid, int hi) {
2    // copy array
3    std::memcpy(aux+lo, A+lo, (hi-lo+1) * sizeof(int));
4    // merge
5    int i = lo, j = mid + 1;
6    for (int k = lo ; k <= hi ; k ++) {
7        if (i > mid) A[k] = aux[j++];
8        else if (j > hi) A[k] = aux[i++];
9        else if (aux[j] < aux[i]) A[k] = aux[j++];
10       else A[k] = aux[i++];
11    }
12 }
```

Merge Sort

```plaintext
1 void r_mergesort(int *A, int *aux, int lo, int hi) {
2    // base case (single element or empty list)
3    if (hi <= lo) return;
4    // divide
5    int mid = lo + (hi - lo) / 2;
6    // recursively sort halves
7    r_mergesort(A, aux, lo, mid);
8    r_mergesort(A, aux, mid+1, hi);
9    // merge results
10   merge(A, aux, lo, mid, hi);
11 }
```

```plaintext
1 void mergesort(int *A, int n) {
2    int *aux = new int[n];
3    r_mergesort(A, aux, 0, n-1);
4    delete [] aux;
5 }
```
### Analysis of Merge Sort

\[ T(1) = 1 \quad T(n) = 2T(n/2) + n \]

- **two recursive calls + one merge**

\[ = \Theta(n \log n) \]

### Recursion Tree

A recursion tree accounts for how much work each recursive call makes.

\[ O \left( \sum_{i=0}^{k} \frac{n}{2^i} \right) = O \left( \sum_{i=0}^{k} n \right) = O(k \cdot n) \]

\[ = O(n \cdot \lg n) \]
Comments on Merge Sort

• Major disadvantage
  • it is not in-place
  • in-place algorithm exists but it is complex and inefficient

• Improvements
  • use insertion sort for small arrays
    • avoid overhead on small instances (~10 elements)
  • stop if already sorted
    • avoids unnecessary merge
    • works well with partially sorted arrays

Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best-Case</th>
<th>Average-Case</th>
<th>Worst-Case</th>
<th>Stable?</th>
<th>In-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>