Divide-and-Conquer
MergeSort
Lecture 7

Divide-and-conquer

• Many recursive algorithms fit the following framework:
  1. **Divide** the problem into subproblems
  2. **Conquer** the subproblems by solving them recursively
  3. **Combine** the solution of each subproblem into the solution of the original problem

MergeSort

• Problem:
  − Sort the elements of an array of n numbers
• Algorithm:
  1. **Divide** the array in left and right halves
  2. **Conquer** each half by recursively sorting them
  3. **Combine** the sorted left and right halves into a full sorted array

Example - MergeSort

• Array to be sorted
  3 1 4 1 5 9 2 6 5 3 5 8 9

• Divide array into two halves
  3 1 4 1 5 9 2 6 5 3 5 8 9

• Conquer: Recursively sort each half
  1 1 3 4 5 9 2 3 5 5 6 8 9

• Merge each half into fully sorted array
  1 2 3 3 4 5 5 5 6 8 9 9

Merging halves

1 3 4 5 9 2 3 5 5 6 8 9

• Create temporary array of same size as original:
  tmp =

• Do the "two indices walk", filling tmp
• Copy tmp back into original array

Algorithm merge(A, left, mid, right)
  Input: An array A and indices left, mid, and right, where A[left...mid] is sorted and A[mid+1...right] is sorted
  Output: A[left...right] is sorted

  indexLeft ← left
  /* Index for left half of A */
  indexRight ← mid + 1
  /* Index for right half of A */
  tmp ← Array of same type and size as A
  tmpIndex ← left
  /* Index for tmp */
  while (tmpIndex ≤ right) do {
    if (indexRight > right or (indexLeft = mid and A[indexLeft] = A[indexRight])) {
      tmp[tmpIndex] ← A[indexLeft]
      /* Select left element */
      indexLeft ← indexLeft + 1
    } else {
      tmp[tmpIndex] ← A[indexRight]
      /* Select right element */
      indexRight ← indexRight + 1
    }
    tmpIndex ← tmpIndex + 1
  }
  for k = left to right do A[k] ← tmp[k]
  /* Copy tmp back into A */
mergeSort pseudocode

Algorithm mergeSort(A, left, right)
Input: An array A of numbers, the bounds left and right for the elements to be sorted
Output: A[left...right] is sorted
if (left < right) {
    /* We have at least two elements to sort */
    mid ← ⌊(left + right)/2⌋
    mergeSort(A, left, mid)
    /* Now A[left...mid] is sorted */
    mergeSort(A, mid + 1, right)
    /* Now A[mid+1...right] is sorted */
    merge(A, left, mid, right)
}

Example of execution
mergeSort([3 1 5 4 2], 0, 4)
mergeSort([3 1 5 4 2], 0, 2)
mergeSort([3 1 5 4 2], 0, 1)
mergeSort([3 1 5 4 2], 0, 0) // nothing to do
mergeSort([1 3 5 4 2], 1, 1) // nothing to do
mergeSort([1 3 5 4 2], 2, 2) // nothing to do
mergeSort([1 3 5 4 2], 0, 1, 2) // array becomes [1 3 5 4 2]
mergeSort([1 3 5 4 2], 1, 2, 2) // array stays [1 3 5 4 2]
mergeSort([1 3 5 4 2], 3, 4)
mergeSort([1 3 5 4 2], 3, 3) // nothing to do
mergeSort([1 3 5 4 2], 3, 4) // nothing to do
mergeSort([1 3 5 4 2], 3, 3, 4) // array becomes [1 3 5 2 4]
merge([1 3 5 2 4], 0, 2, 4) // array becomes [1 2 3 4 5]

mergeSort running time

- How does the running time of mergeSort depends on the size n of the array?
- Let T(n) = time to sort an array of size n = right - left + 1
- Assume n is a power of 2 (for simplicity)

Algorithm mergeSort(A, l, r)
if (l<r) {
    mid ← ⌊(l+r)/2⌋
    mergeSort(A, l, mid)
    mergeSort(A, mid+1, r)
    merge(A, l, mid, r)
}

Running time, with n = 1 - r + 1
C₁ (independent of n)
C₂ (independent of n)
T(n/2) + C₃
C₄ ⋅ n + C₅
Total: T(n) = C₁ + C₂ + C₃ + C₄ ⋅ n + T(n/2) + C₅
      = C₆ + T(n/2) + C₅n
T(0) = T(1) = C₁

Example
Suppose C₁ = C₀ = C₄ = 1 (for simplicity of example)
We have
T(0) = T(1) = 1
T(n) = 1 + 2 T(n/2) + n
Thus,

<table>
<thead>
<tr>
<th>n</th>
<th>size</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1 + 2</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>1 + 2 + 4</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>128</td>
<td>512</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>512</td>
<td>1024</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>1024</td>
<td>2048</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>2048</td>
<td>4096</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>4096</td>
<td>8192</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>8192</td>
<td>16384</td>
<td>1 + 2 + 4 + 8</td>
</tr>
<tr>
<td>16384</td>
<td>32768</td>
<td>1 + 2 + 4 + 8</td>
</tr>
</tbody>
</table>

Running time of Merge Sort

Graph showing running time of merge sort algoirthm