



Divide-and-Conquer MergeSort

Lecture 7

Divide-and-conquer

- Many recursive algorithms fit the following framework:
 - Divide** the problem into subproblems
 - Conquer** the subproblems by solving them recursively
 - 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:
 - Divide** the array in left and right halves
 - Conquer** each half by recursively sorting them
 - 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

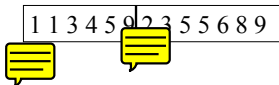
3	1	4	1	5	9	2	6	5	3	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---
- Conquer: Recursively sort each half

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

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



Merging halves



- 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([3 1 5 4 2], 1, 1) // nothing to do
| | | merge([3 1 5 4 2], 0, 0, 1) //array becomes [1 3 5 4 2]
| | | mergeSort([1 3 5 4 2], 2, 2) // nothing to do
| | | merge([1 3 5 4 2], 0, 1, 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], 4, 4) // nothing to do
| | | merge([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 $n = l - r + 1$
 C_1 (independent of n)
 C_2 (independent of n)
 $T(n/2)$
 $T(n/2) + C_3$
 $C_4 \cdot n + C_5$
Total: $T(n) = C_1 + C_2 + C_3 + C_5 + 2 T(n/2) + C_4 n$
 $= C_6 + 2 T(n/2) + C_4 n$
 $T(0) = T(1) = C_1$

Example

Suppose $C_1 = C_6 = C_4 = 1$ (for simplicity of example)

We have

$$T(0) = T(1) = 1$$

$$T(n) = 1 + 2 T(n/2) + n$$

Thus,

1	2	4	8	16	32	64	...	n
1	5	15	39	95	223	511		?

Running time of Merge Sort

