CSE 221: Algorithms

Divide and Conquer

Mumit Khan

Computer Science and Engineering
BRAC University

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
2. Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms*. MIT OpenCourseWare, Fall 2005. Available from: `ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm`

Last modified: June 11, 2009

Divide-and-Conquer design strategy

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving these recursively.
3. *Combine* the solutions to the subproblems.

Divide-and-Conquer design strategy

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving these recursively.
3. *Combine* the solutions to the subproblems.

Divide-and-Conquer design strategy

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving these recursively.
3. *Combine* the solutions to the subproblems.

Divide-and-Conquer design strategy

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving these recursively.
3. *Combine* the solutions to the subproblems.

Divide-and-Conquer design strategy

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving these recursively.
3. *Combine* the solutions to the subproblems.

Example (of D&C strategy)

1. Binary search – divide the problem into half, and recursively search the appropriate 1 subproblem.
2. Mergesort – divide the problem into half, and recursively sort 2 subproblems, and then merge the results into a complete sorted sequence.
3. Computing x^n, computing fibonacci numbers, multiplying matrices (using *Strassen's algorithm*), etc.

Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

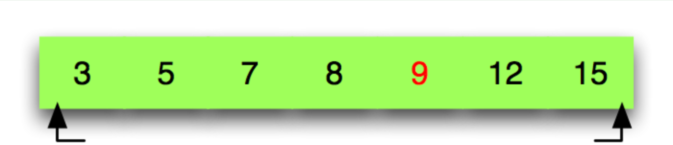
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



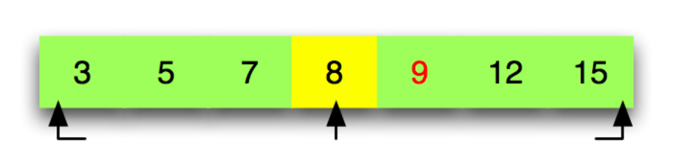
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



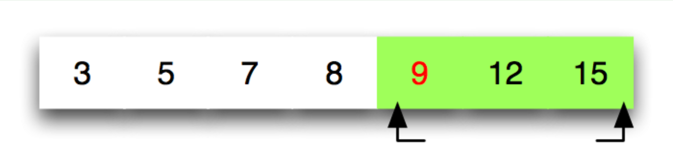
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



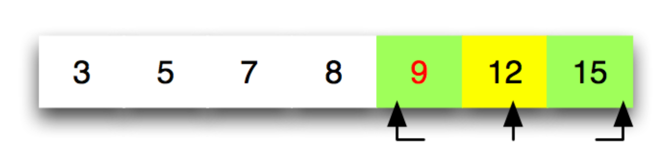
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



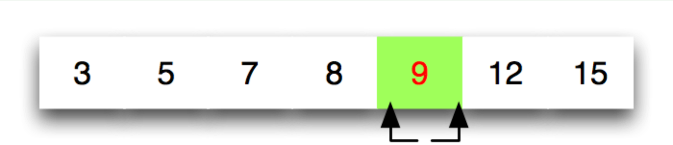
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



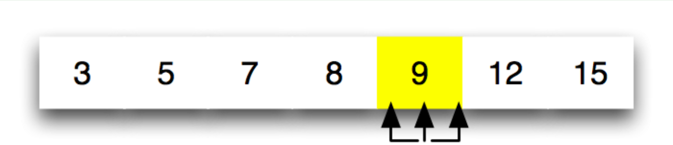
Binary search

The problem

Find an element in a sorted array:

1. *Divide:* Check the middle element.
2. *Conquer:* Recursively search 1 subarray.
3. *Combine:* Trivial.

Example (Searching for 9)



Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

Recurrence for binary search

$$T(n) = \underbrace{1}_{\text{\# subproblems}} T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(1)}_{\text{work dividing and combining}}$$

Recurrence for binary search

$$T(n) = \underbrace{1}_{\text{\# subproblems}}\, T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(1)}_{\text{work dividing and combining}}$$

Analysis

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow$ CASE 2 $(k = 0)$

$$\Rightarrow \boxed{T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(\lg n)}$$

Merge sort

The problem

Find an element in a sorted array:

1. *Divide:* Trivial.
2. *Conquer:* Recursively sort 2 subarrays.
3. *Combine:* Merge the sorted subarrays in $\Theta(n)$ time.

Merge sort

The problem

Find an element in a sorted array:

1. *Divide:* Trivial.
2. *Conquer:* Recursively sort 2 subarrays.
3. *Combine:* Merge the sorted subarrays in $\Theta(n)$ time.

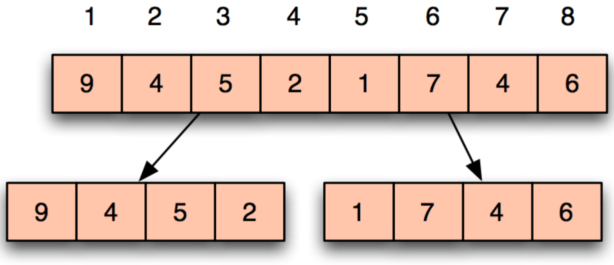
Key subroutine

MERGE – to merge two sorted arrays in linear-time.

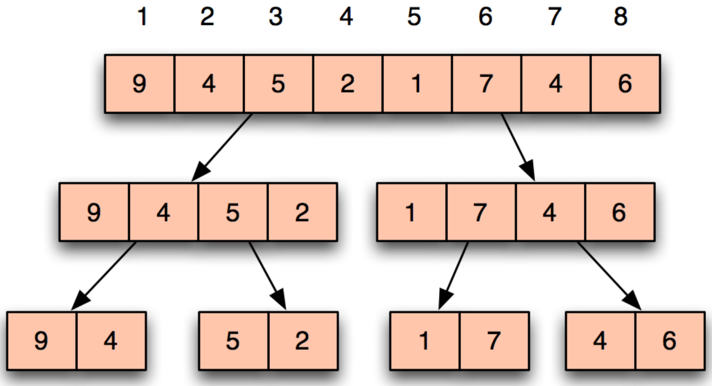
Merge sort in action

1	2	3	4	5	6	7	8
9	4	5	2	1	7	4	6

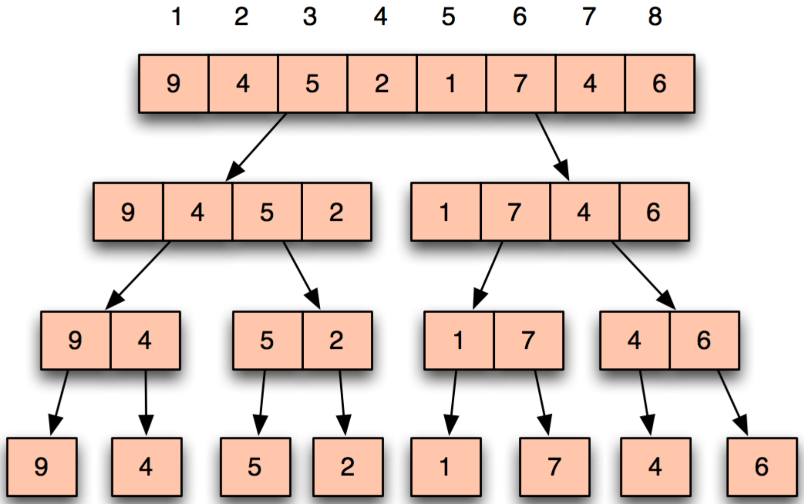
Merge sort in action



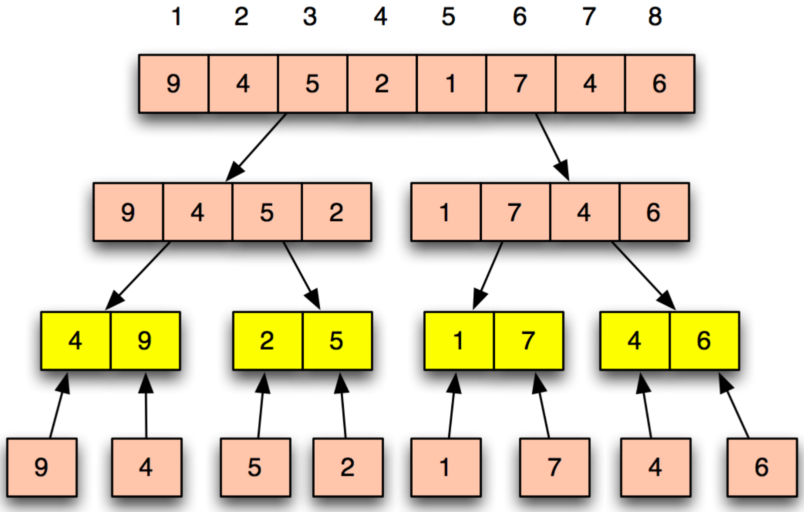
Merge sort in action



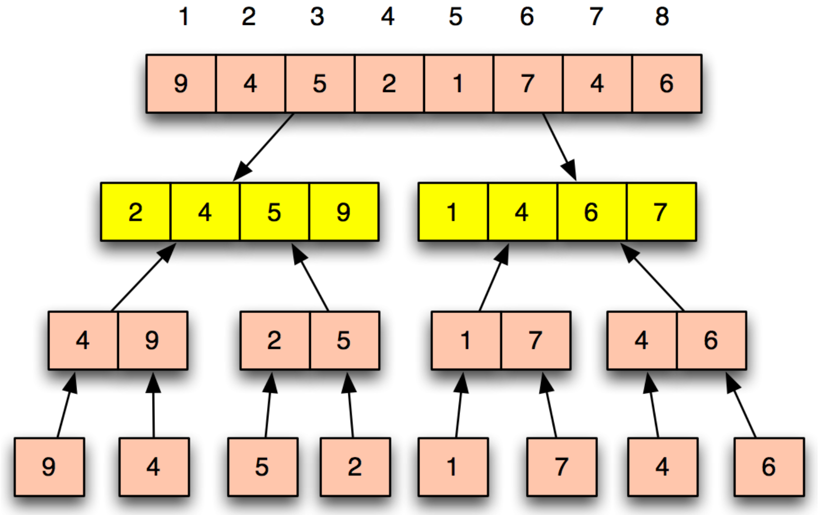
Merge sort in action



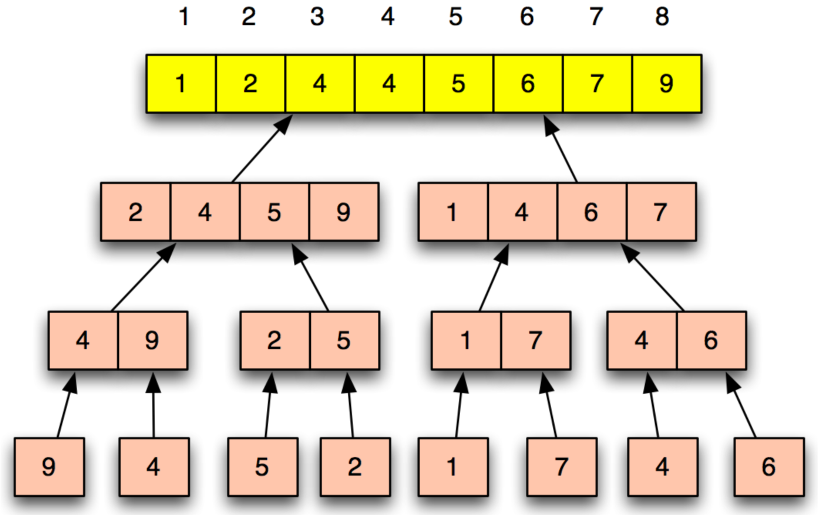
Merge sort in action



Merge sort in action



Merge sort in action

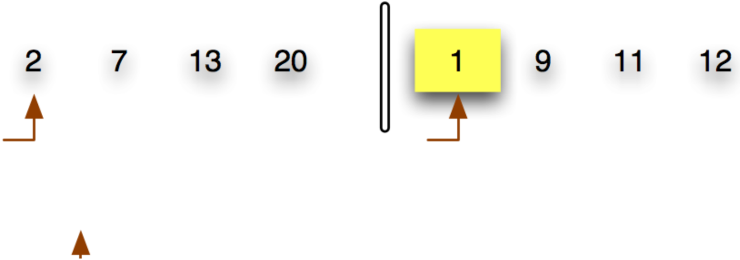


Merging in $\Theta(n)$ time

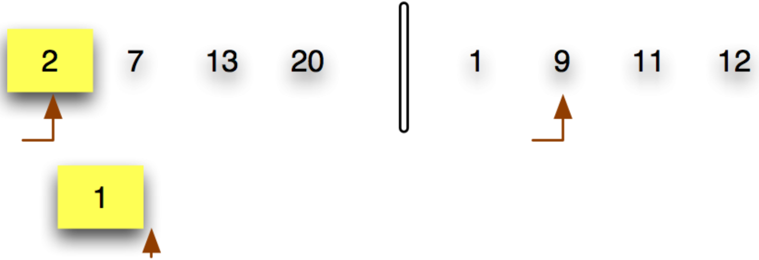
2 7 13 20 | 1 9 11 12

1 2 7 9 11 12 13 20

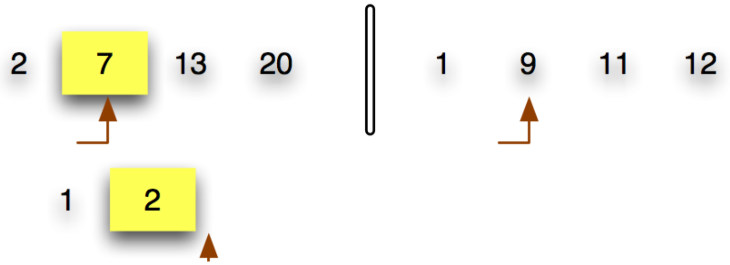
Merging in $\Theta(n)$ time



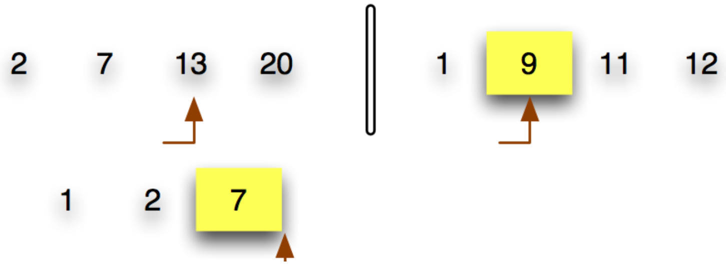
Merging in $\Theta(n)$ time



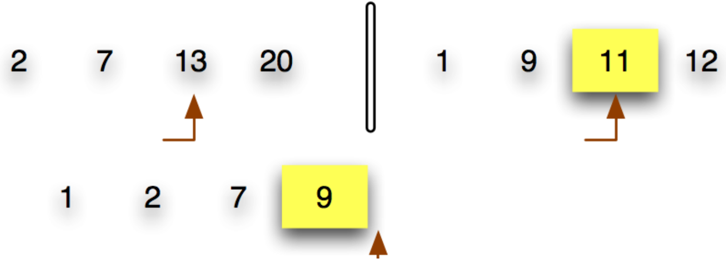
Merging in $\Theta(n)$ time



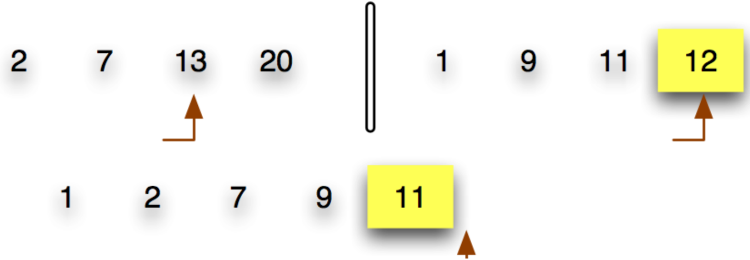
Merging in $\Theta(n)$ time



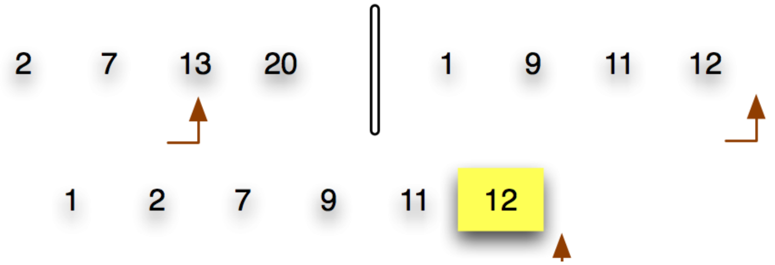
Merging in $\Theta(n)$ time



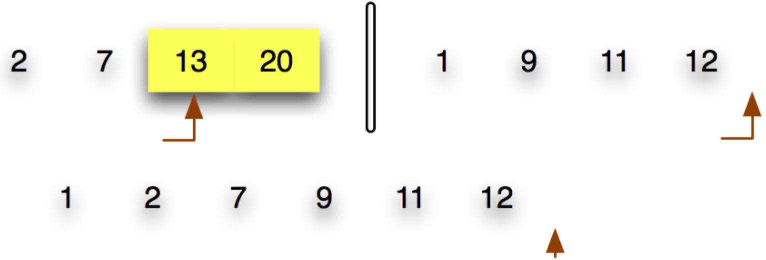
Merging in $\Theta(n)$ time



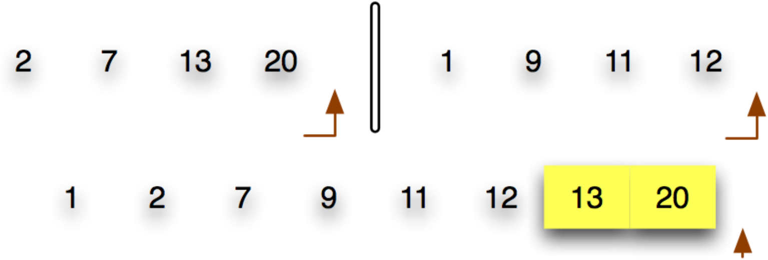
Merging in $\Theta(n)$ time



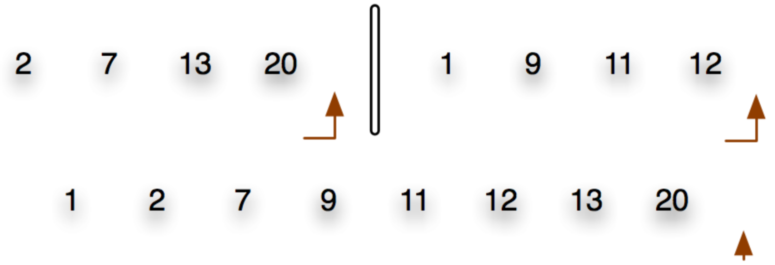
Merging in $\Theta(n)$ time



Merging in $\Theta(n)$ time



Merging in $\Theta(n)$ time



A $\Theta(n)$ time merge algorithm

```
MERGE(A, B)
    INPUT: Two sorted arrays A and B
    OUTPUT: Returns C as the merged array
    ▷ n1 = length[A], n2 = length[B], n = n1 + n2
    Create C[1 . . n]
    Initialize two indices to point to A and B
    while A and B are not empty
          do Select the smaller of two and add to end of C
             Advance the index that points to the smaller one
    if A or B is not empty
        then Copy the rest of the non-empty array to the end of C
    return C
```

A $\Theta(n)$ time merge algorithm

MERGE(A, B)

INPUT: Two sorted arrays A and B
OUTPUT: Returns C as the merged array
$\triangleright\ n_1 = length[A],\ n_2 = length[B],\ n = n_1 + n_2$
Create $C[1 \mathinner{.\,.} n]$
Initialize two indices to point to A and B
while A and B are not empty
    do Select the smaller of two and add to end of C
        Advance the index that points to the smaller one
if A or B is not empty
    then Copy the rest of the non-empty array to the end of C
return C

$\boxed{T(n) = \Theta(n)}$

A $\Theta(n)$ time merge algorithm

```
MERGE(A, B)
    INPUT: Two sorted arrays A and B
    OUTPUT: Returns C as the merged array
    ▷ n1 = length[A], n2 = length[B], n = n1 + n2
    Create C[1 . . n]
    Initialize two indices to point to A and B
    while A and B are not empty
          do Select the smaller of two and add to end of C
             Advance the index that points to the smaller one
    if A or B is not empty
       then Copy the rest of the non-empty array to the end of C
    return C
```

$\boxed{T(n) = \Theta(n)}$

Issue: Out-of-place algorithm.

A $\Theta(n)$ time merge algorithm

```
MERGE(A, B)
    INPUT: Two sorted arrays A and B
    OUTPUT: Returns C as the merged array
    ▷ n1 = length[A], n2 = length[B], n = n1 + n2
    Create C[1 . . n]
    Initialize two indices to point to A and B
    while A and B are not empty
        do Select the smaller of two and add to end of C
           Advance the index that points to the smaller one
    if A or B is not empty
      then Copy the rest of the non-empty array to the end of C
    return C
```

$$\boxed{T(n) = \Theta(n)}$$

Issue: Out-of-place algorithm. Can it be made in-place?

Merge sort algorithm

$\text{MERGE-SORT}(A) \triangleright A[1..n]$

```
if n = 1
    then return
    else                                    ▷ recursively sort the two subarrays
        A1 = MERGE-SORT(A[1 .. ⌈n/2⌉])
        A2 = MERGE-SORT(A[⌈n/2⌉ + 1 .. n])
        A = MERGE(A1, A2)                   ▷ merge the sorted arrays
```

Merge sort algorithm

$\textsc{Merge-Sort}(A)$ ▷ $A[1\,..\,n]$

if $n = 1$
  then return
  else ▷ recursively sort the two subarrays
    $A_1 = \textsc{Merge-Sort}(A[1\,..\,\lceil n/2 \rceil])$
    $A_2 = \textsc{Merge-Sort}(A[\lceil n/2 \rceil + 1\,..\,n])$
    $A = \textsc{Merge}(A_1, A_2)$ ▷ merge the sorted arrays

Few notes on the algorithm

1. Dividing is trivial, but it's the merging that requires most time.
2. The merging algorithm presented here is an out-of-place algorithm, which will increase space complexity.

Merge sort algorithm

$\text{MERGE-SORT}(A) \triangleright A[1\,..\,n]$

if $n = 1$
 then return
 else $\triangleright$ recursively sort the two subarrays
  $A_1 = \text{MERGE-SORT}(A[1\,..\,\lceil n/2 \rceil])$
  $A_2 = \text{MERGE-SORT}(A[\lceil n/2 \rceil + 1\,..\,n])$
  $A = \text{MERGE}(A_1, A_2)$ $\triangleright$ merge the sorted arrays

Few notes on the algorithm

1. Dividing is trivial, but it's the merging that requires most time.
2. The merging algorithm presented here is an out-of-place algorithm, which will increase space complexity. Can it be made in-place?

Recurrence for merge sort

$$T(n) = 2\,T(n/2) + \Theta(n)$$

Recurrence for merge sort

$$T(n) = \underbrace{2}_{\text{\# subproblems}} T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(n)}_{\text{work dividing and combining}}$$

Recurrence for merge sort

$$T(n) = \underbrace{2}_{\text{\# subproblems}} T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(n)}_{\text{work dividing and combining}}$$

Analysis

$n^{\log_b a} = n^{\log_2 2} = n^1 = n \Rightarrow \text{CASE 2}\ (k = 0)$

$$\Rightarrow \boxed{T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(n \lg n)}$$

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.
Naive algorithm: $\Theta(n)$.

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.
Naive algorithm: $\Theta(n)$.
Divide-and-conquer algorithm:

$$a_n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.
Naive algorithm: $\Theta(n)$.
Divide-and-conquer algorithm:

$$a_n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

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

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.
Naive algorithm: $\Theta(n)$.
Divide-and-conquer algorithm:

$$a_n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow \boxed{T(n) = \Theta(\lg n)}.$$

Conclusion

- Divide and Conquer is just one of several algorithm design strategies.
- Used by many of the commonly used algorithms
  - Binary search
  - Merge sort
  - Fast Fourier Transform (FFT)
  - Finding closest pair of points
  - Matrix multiplication (Strassen's algorithm)
  - Matrix inversion
  - Quicksort and (k^{th}) selection
  - ...
- Can be easily analyzed using recurrences
- Often leads to efficient algorithms