CMPSC 465: LECTURE III

Sorting algorithms

Ke Chen

September 03, 2025

Sorting problem

Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$.

Output: A reordering (permutation) of the input sequence $a'_1, a'_2, \ldots, a'_n$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

Input:	8, 1, 9, 2, 8, 4, 6, 5
Output:	1, 2, 4, 5, 6, 8, 8, 9

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

Input:	8	1	9	2	8	4	6	5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

Input: 8 1 9 2 8 4 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

1st iteration: 1 8 9 2 8 4 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

2nd iteration: 1 8 9 2 8 4 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

3rd iteration: 1 8 9 2 8 4 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

3rd iteration: $1 \quad 8 \quad 2 \quad 9 \quad 8 \quad 4 \quad 6 \quad 5$

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

3rd iteration: $1 \quad 2 \quad 8 \quad 9 \quad 8 \quad 4 \quad 6 \quad 5$

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

4th iteration: $1 \quad 2 \quad 8 \quad 9 \quad 8 \quad 4 \quad 6 \quad 5$

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

4th iteration: 1 2 8 8 9 4 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

5th iteration: $1 \quad 2 \quad 8 \quad 8 \quad 9 \quad 4 \quad 6 \quad 5$

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

6th iteration: 1 2 4 8 8 9 6 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

7th iteration: 1 2 4 6 8 8 9 5

InsertionSort

Idea: repeatedly insert the next number to the partially sorted sequence.

7th iteration: 1 2 4 5 6 8 8 9

InsertionSort

InsertionSort($A[1..n]$)

for $i = 1$ **to** n **do**
- $key = A[i]$
- $j = i - 1$
- **while** $j > 0$ **and** $A[j] > key$ **do**
  - $A[j+1] = A[j]$
  - $A[j] = key$
  - $j = j - 1$

InsertionSort

InsertionSort($A[1..n]$)

```
for i = 1 to n do
    key = A[i]
    j = i − 1
    while j > 0 and A[j] > key do
        A[j + 1] = A[j]
        A[j] = key
        j = j − 1
```

Correctness?

Loop invariant (property that is true after each iteration):
after the k-th iteration, $A[1..k]$ is sorted.

InsertionSort

```
InsertionSort(A[1..n])
  for i = 1 to n do
    key = A[i]
    j = i − 1
    while j > 0 and A[j] > key do
      A[j + 1] = A[j]
      A[j] = key
      j = j − 1
```

Correctness?

Loop invariant (property that is true after each iteration):
after the k-th iteration, $A[1..k]$ is sorted.

Proof by induction: $A[1]$ is sorted (base case).
If $A[1..k-1]$ is sorted, after inserting $A[k]$, $A[1..k]$ is sorted.
Consequently, $A[1..n]$ is sorted when the algorithm finishes. □

InsertionSort

InsertionSort($A[1..n]$)

for $i = 1$ **to** n **do**
    $key = A[i]$
    $j = i - 1$
    while $j > 0$ **and** $A[j] > key$ **do**
        $A[j+1] = A[j]$
        $A[j] = key$
        $j = j - 1$

Time complexity?

while-loop: $O(n)$ time

for-loop: $O(n)$ rounds, $O(n)$ time each, in total $O(n^2)$.

InsertionSort

```
InsertionSort(A[1..n])
    for i = 1 to n do
        key = A[i]
        j = i − 1
        while j > 0 and A[j] > key do
            A[j + 1] = A[j]
            A[j] = key
            j = j − 1
```

Time complexity?

while-loop: $O(n)$ time

for-loop: $O(n)$ rounds, $O(n)$ time each, in total $O(n^2)$.

A more careful analysis: the i-th round takes $O(i)$ time.

In total: $O(1) + O(2) + \cdots + O(n)$

InsertionSort

```
InsertionSort(A[1..n])
  for i = 1 to n do
    key = A[i]
    j = i − 1
    while j > 0 and A[j] > key do
      A[j + 1] = A[j]
      A[j] = key
      j = j − 1
```

Time complexity?

while-loop: $O(n)$ time

for-loop: $O(n)$ rounds, $O(n)$ time each, in total $O(n^2)$.

A more careful analysis: the i-th round takes $O(i)$ time.

In total: $O(1) + O(2) + \cdots + O(n)$

$= O(1 + 2 + \cdots + n) = O(n(n+1)/2) = O(n^2)$.

InsertionSort

InsertionSort($A[1..n]$)

for $i = 1$ **to** n **do**
- $key = A[i]$
- $j = i - 1$
- **while** $j > 0$ **and** $A[j] > key$ **do**
  - $A[j+1] = A[j]$
  - $A[j] = key$
  - $j = j - 1$

Space complexity?

Only uses $O(1)$ additional space beyond the n cells for the input data.

InsertionSort

InsertionSort($A[1..n]$)

for $i = 1$ **to** n **do**
- $key = A[i]$
- $j = i - 1$
- **while** $j > 0$ **and** $A[j] > key$ **do**
  - $A[j+1] = A[j]$
  - $A[j] = key$
  - $j = j - 1$

Can we do better?

InsertionSort

InsertionSort($A[1..n]$)

for $i = \cancel{1}_{2}$ **to** n **do**
- $key = A[i]$
- $j = i - 1$
- **while** $j > 0$ **and** $A[j] > key$ **do**
  - $A[j+1] = A[j]$
  - $A[j] = key$
  - $j = j - 1$

Can we do better?

Yes? We can do some optimization ...

InsertionSort

InsertionSort($A[1..n]$)

for $i = \cancel{1}\ 2$ **to** n **do**

  $key = A[i]$

  $j = i - 1$

  while $j > 0$ **and** $A[j] > key$ **do**

    $A[j+1] = A[j]$

    $\cancel{A[j] = key}$

    $j = j - 1$

  $A[j+1] = key$

Can we do better?

Yes? We can do some optimization ...

InsertionSort

InsertionSort($A[1..n]$)

for $i = \cancel{1}\,2$ **to** n **do**
  $key = A[i]$
  $j = i - 1$
  while $j > 0$ **and** $A[j] > key$ **do**
    $A[j+1] = A[j]$
    $\cancel{A[j] = key}$
    $j = j - 1$
  $A[j+1] = key$

Can we do better?

Yes? We can do some optimization ...

No. If $A[1..n]$ is given in decreasing order, we need at least $1 + 2 + \cdots + n - 1 = n(n-1)/2 = \Omega(n^2)$ comparisons. So the worst-case running time of InsertionSort is $\Theta(n^2)$.

InsertionSort

InsertionSort($A[1..n]$)

for $i = \cancel{1}\ 2$ **to** n **do**
  $key = A[i]$
  $j = i - 1$
  while $j > 0$ **and** $A[j] > key$ **do**
    $A[j+1] = A[j]$
    $\cancel{A[j] = key}$
    $j = j - 1$
  $A[j+1] = key$

Can we do better?

Yes? We can do some optimization ...

No. If $A[1..n]$ is given in decreasing order, we need at least $1 + 2 + \cdots + n - 1 = n(n-1)/2 = \Omega(n^2)$ comparisons. So the worst-case running time of InsertionSort is $\Theta(n^2)$.

Maybe with a different sorting algorithm?

MergeSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

1. Split input in halves: 8,1,9,2 and 8,4,6,5
2. Sort each half: 1,2,8,9 and 4,5,6,8
3. Merge two sorted halves: 1,2,4,5,6,8,8,9

MergeSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

1. Split input in halves: 8,1,9,2 and 8,4,6,5
2. Sort each half: 1,2,8,9 and 4,5,6,8
3. Merge two sorted halves: 1,2,4,5,6,8,8,9

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- $B_L =$ MergeSort($A\,[1..\lceil n/2 \rceil]$)
- $B_R =$ MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

1. Split input in halves: 8,1,9,2 and 8,4,6,5
2. Sort each half: 1,2,8,9 and 4,5,6,8
3. Merge two sorted halves: 1,2,4,5,6,8,8,9

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
$B_L = $ MergeSort($A\,[1..\lceil n/2\rceil]$)
$B_R = $ MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
return Merge(B_L, B_R)

How to do Merge?

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$\begin{array}{lcccc} B_L: & 1 & 2 & 8 & 9 \\ & \uparrow & & & \\ B_R: & 4 & 5 & 6 & 8 \\ & \uparrow & & & \\ \text{Output}: & & & & \end{array}$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lcccc}
B_L: & 1 & \underset{\uparrow}{2} & 8 & 9 \\
B_R: & \underset{\uparrow}{4} & 5 & 6 & 8 \\
\text{Output}: & 1 & & &
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{rcccc}
B_L: & 1 & 2 & \underset{\uparrow}{8} & 9 \\
B_R: & \underset{\uparrow}{4} & 5 & 6 & 8 \\
\text{Output}: & 1 & 2 & &
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lcccc}
B_L: & 1 & 2 & \underset{\uparrow}{8} & 9 \\
B_R: & 4 & \underset{\uparrow}{5} & 6 & 8 \\
\text{Output}: & 1 & 2 & 4 &
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lcccc}
B_L: & 1 & 2 & 8 & 9 \\
 & & & \uparrow & \\
B_R: & 4 & 5 & 6 & 8 \\
 & & & \uparrow & \\
\text{Output}: & 1 & 2 & 4 & 5
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lccccc}
B_L: & 1 & 2 & \underset{\uparrow}{8} & 9 & \\
B_R: & 4 & 5 & 6 & \underset{\uparrow}{8} & \\
\text{Output}: & 1 & 2 & 4 & 5 & 6
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lcccccc}
B_L: & 1 & 2 & 8 & 9 & & \\
 & & & & \uparrow & & \\
B_R: & 4 & 5 & 6 & 8 & & \\
 & & & & \uparrow & & \\
\text{Output}: & 1 & 2 & 4 & 5 & 6 & 8
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

$$
\begin{array}{lccccccc}
B_L: & 1 & 2 & 8 & 9 & & & \\
 & & & & \uparrow & & & \\
B_R: & 4 & 5 & 6 & 8 & & & \\
 & & & & & \uparrow & & \\
\text{Output}: & 1 & 2 & 4 & 5 & 6 & 8 & 8
\end{array}
$$

Merge two sorted arrays

Idea: should take advantage of the fact that both B_L and B_R are already sorted.

B_L :	1	2	8	9				
B_R :	4	5	6	8				
Output :	1	2	4	5	6	8	8	9

Merge two sorted arrays

Merge($X[1..k]$, $Y[1..\ell]$)

if X *is empty* **then return** Y
if Y *is empty* **then return** X
if $X[1] \leq Y[1]$ **then**
    return $[X[1]$, Merge$(X[2..k], Y)]$
else
    return $[Y[1]$, Merge$(X, Y[2..\ell])]$

Merge two sorted arrays

Merge($X[1..k]$, $Y[1..\ell]$)

if X *is empty* **then return** Y
if Y *is empty* **then return** X
if $X[1] \leq Y[1]$ **then**
  return $[X[1]$, Merge$(X[2..k], Y)]$
else
  return $[Y[1]$, Merge$(X, Y[2..\ell])]$

Correctness?

- Since X and Y are both sorted, if $X[1] \leq Y[1]$, $X[1]$ is a smallest element.
- By induction, Merge$(X[2..k], Y)$ produces a sorted array.
- After prepending $X[1]$, the result remains sorted.
- Similar for $X[1] > Y[1]$.

Merge two sorted arrays

```
Merge(X[1..k], Y[1..ℓ])
    if X is empty then return Y
    if Y is empty then return X
    if X[1] ≤ Y[1] then
        return [X[1], Merge(X[2..k], Y)]
    else
        return [Y[1], Merge(X, Y[2..ℓ])]
```

Time complexity?

Each round adds a new element to the output in constant time: $O(k + \ell)$.

Merge two sorted arrays

Merge($X[1..k]$, $Y[1..\ell]$)

if X *is empty* **then return** Y
if Y *is empty* **then return** X
if $X[1] \leq Y[1]$ **then**
  return $[X[1]$, Merge$(X[2..k], Y)]$
else
  return $[Y[1]$, Merge$(X, Y[2..\ell])]$

Time complexity?

Each round adds a new element to the output in constant time: $O(k + \ell)$.

Exercise: Show that for all $k, \ell \geq 1$, there are two sorted arrays of sizes k and ℓ, respectively, such that merging them requires $k + \ell - 1$ comparisons. Hence the above upper bound is tight.

Merge two sorted arrays

Merge($X[1..k]$, $Y[1..\ell]$)

if X *is empty* **then return** Y
if Y *is empty* **then return** X
if $X[1] \leq Y[1]$ **then**
 return $[X[1]$, Merge$(X[2..k], Y)]$
else
 return $[Y[1]$, Merge$(X, Y[2..\ell])]$

Time complexity?

Each round adds a new element to the output in constant time: $O(k+\ell)$.

Exercise: Show that for all $k, \ell \geq 1$, there are two sorted arrays of sizes k and ℓ, respectively, such that merging them requires $k + \ell - 1$ comparisons. Hence the above upper bound is tight.

Space complexity?

Uses $O(k+\ell)$ extra space for the output.

Merge two sorted arrays

Merge($X[1..k]$, $Y[1..\ell]$)

if X *is empty* **then return** Y
if Y *is empty* **then return** X
if $X[1] \leq Y[1]$ **then**
  return $[X[1]$, Merge$(X[2..k], Y)]$
else
  return $[Y[1]$, Merge$(X, Y[2..\ell])]$

Can we do better?

Time bound is tight: $\Theta(k + \ell)$.

Space complexity can be improved to $O(\min\{k, \ell\})$. (Exercise)

MergeSort

<u>MergeSort($A[1..n]$)</u>

if $n == 1$ **then return** A
$B_L =$ MergeSort($A\,[1..\lceil n/2 \rceil]$)
$B_R =$ MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
return Merge(B_L, B_R)

MergeSort

MergeSort($A[1..n]$)
- **if** $n == 1$ **then return** A
- $B_L =$ MergeSort($A\,[1..\lceil n/2\rceil]$)
- $B_R =$ MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
- **return** Merge(B_L, B_R)

Correctness?

Proof.

By induction on the size of the input n. (What is the base case?)

Suppose MergeSort correctly sorts arrays of size up to $n - 1$.

Then the two recursive calls produce sorted arrays.

Since we have shown Merge is correct, the final result is sorted.

So MergeSort works correctly on arrays of size n.

By induction, MergeSort is correct for all finite input sizes. $\square$

MergeSort

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
return Merge(B_L, B_R)

MergeSort in action

Input:
$8, 1, 9, 2, 8, 4, 6, 5$

MergeSort

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
B_L = MergeSort($A\left[1..\lceil n/2\rceil\right]$)
B_R = MergeSort($A\left[\lceil n/2\rceil + 1..n\right]$)
return Merge(B_L, B_R)

MergeSort in action

Input:
$8, 1, 9, 2, 8, 4, 6, 5$

split

$8, 1, 9, 2$ $\qquad$ $8, 4, 6, 5$

MergeSort

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort in action

Input:

$8, 1, 9, 2, 8, 4, 6, 5$

split

$8, 1, 9, 2$ $\quad$ $8, 4, 6, 5$

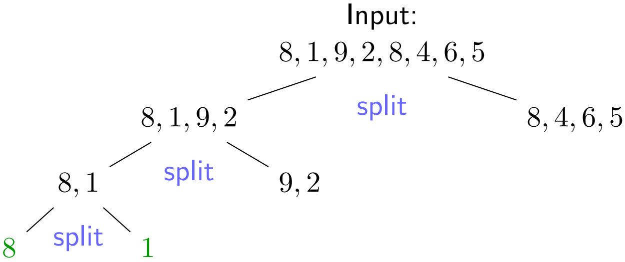
split

$8, 1$ $\quad$ $9, 2$

MergeSort

MergeSort($A[1..n]$)
- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort in action

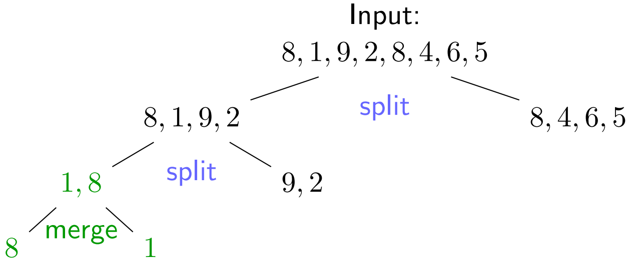


MergeSort

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort in action

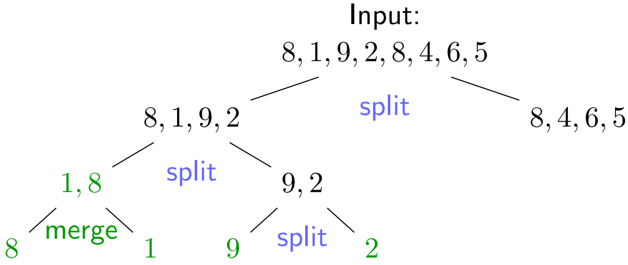


MergeSort

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

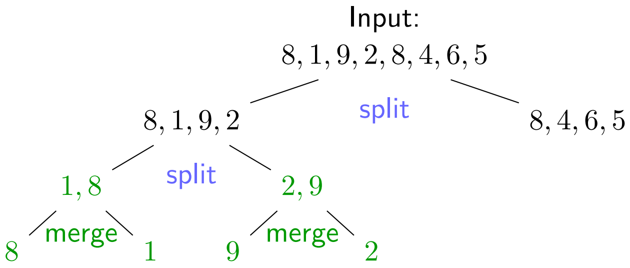
MergeSort in action



MergeSort

MergeSort($A[1..n]$)
- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort in action

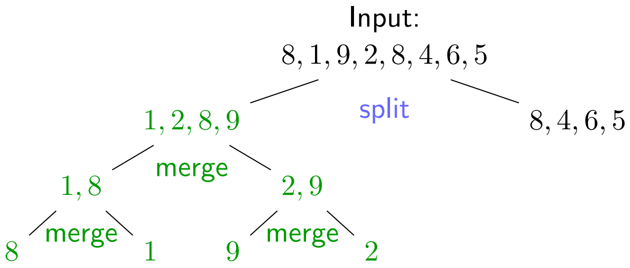


MergeSort

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
B_L = MergeSort($A\,[1..\lceil n/2\rceil]$)
B_R = MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
return Merge(B_L, B_R)

MergeSort in action

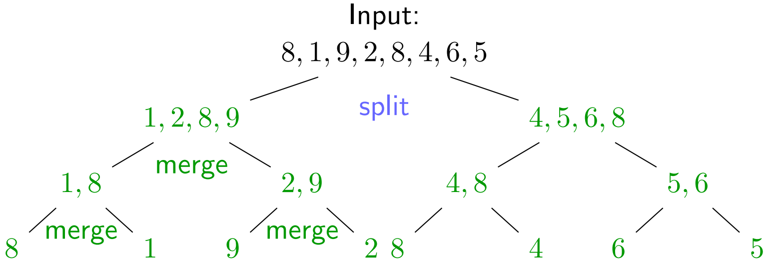


MergeSort

<u>MergeSort($A[1..n]$)</u>

> **if** $n == 1$ **then return** A
> $B_L =$ MergeSort($A\,[1..\lceil n/2\rceil]$)
> $B_R =$ MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
> **return** Merge(B_L, B_R)

MergeSort in action

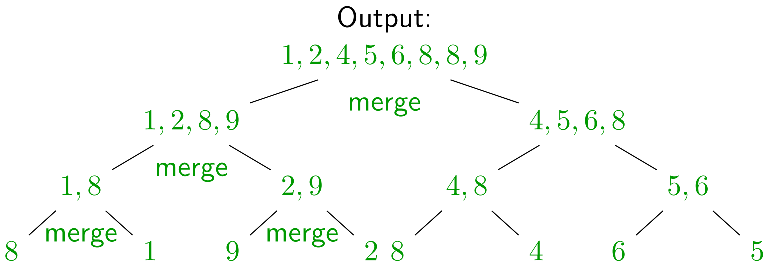


MergeSort

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R)

MergeSort in action



MergeSort

MergeSort($A[1..n]$)

- **if** $n == 1$ **then return** A
- $B_L =$ MergeSort($A\,[1..\lceil n/2\rceil]$)
- $B_R =$ MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
- **return** Merge(B_L, B_R)

Time complexity?

Let $T(n)$ be the running time of MergeSort on an input of size n.

We have:

$$T(n) =$$

MergeSort

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
$B_L = $ MergeSort($A\,[1..\lceil n/2\rceil]$) $\leftarrow$
$B_R = $ MergeSort($A\,[\lceil n/2\rceil + 1..n]$)
return Merge(B_L, B_R)

Time complexity?

Let $T(n)$ be the running time of MergeSort on an input of size n.
We have:

$$T(n) = T(\lceil n/2\rceil) +$$

MergeSort

MergeSort($A[1..n]$)

if $n == 1$ **then return** A
B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$) $\leftarrow$
return Merge(B_L, B_R)

Time complexity?

Let $T(n)$ be the running time of MergeSort on an input of size n.
We have:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) +$$

MergeSort

MergeSort($A[1..n]$)
- **if** $n == 1$ **then return** A
- B_L = MergeSort($A\,[1..\lceil n/2 \rceil]$)
- B_R = MergeSort($A\,[\lceil n/2 \rceil + 1..n]$)
- **return** Merge(B_L, B_R) $\leftarrow$

Time complexity?

Let $T(n)$ be the running time of MergeSort on an input of size n.

We have:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

MergeSort

MergeSort($A[1..n]$)

```
if n == 1 then return A
B_L = MergeSort(A[1..⌈n/2⌉])
B_R = MergeSort(A[⌈n/2⌉ + 1..n])
return Merge(B_L, B_R)
```

Time complexity?

Let $T(n)$ be the running time of MergeSort on an input of size n.

We have:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

How to solve for $T(n)$?

Solving recurrences

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

Simplification: assume n is a power of 2 so we can ignore floors and ceilings.

Solving recurrences

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$

Simplification: assume n is a power of 2 so we can ignore floors and ceilings.

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

Solving recurrences

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$.

Simplification: assume n is a power of 2 so we can ignore floors and ceilings.

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

Solve by substitution

- Make a guess, e.g., $T(n) = O(n \log n)$.
- Try to prove the guess by induction.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n \log n)$.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n \log n)$.

In order to prove our guess, we need to show that $T(n) \leq c \cdot n \log n$ for some constant c.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n \log n)$.

In order to prove our guess, we need to show that $T(n) \leq c \cdot n \log n$ for some constant c.

Assume this is true for all $m < n$, in particular, for $m = n/2$. Substituting into the recurrence gives

Solving recurrences by substitution

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

Let's guess $T(n) = O(n \log n)$.

In order to prove our guess, we need to show that
$T(n) \leq c \cdot n \log n$ for some constant c.

Assume this is true for all $m < n$, in particular, for $m = n/2$.
Substituting into the recurrence gives

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&\leq 2c \cdot (n/2) \log(n/2) + O(n) \\
&= 2c \cdot (n/2) \log n - 2c \cdot n/2 + O(n) \\
&\leq c \cdot n \log n - c \cdot n + c' \cdot n \\
&= c \cdot n \log n - (c - c') \cdot n \\
&\leq c \cdot n \log n.
\end{aligned}$$

The last step holds as long as we choose $c \geq c'$.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n)$.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n)$.

Need to show that $T(n) \leq c \cdot n$ for some constant c.

Solving recurrences by substitution

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

Let's guess $T(n) = O(n)$.

Need to show that $T(n) \leq c \cdot n$ for some constant c.

Assume this is true for all $m < n$, in particular, for $m = n/2$.
Substituting into the recurrence gives

Solving recurrences by substitution

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

Let's guess $T(n) = O(n)$.

Need to show that $T(n) \leq c \cdot n$ for some constant c.

Assume this is true for all $m < n$, in particular, for $m = n/2$. Substituting into the recurrence gives

$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&\leq 2c \cdot n/2 + O(n) \\
&\leq 2c \cdot n/2 + c' \cdot n \\
&= (c + c')n
\end{aligned}
$$

Solving recurrences by substitution

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

Let's guess $T(n) = O(n)$.

Need to show that $T(n) \leq c \cdot n$ for some constant c.

Assume this is true for all $m < n$, in particular, for $m = n/2$.
Substituting into the recurrence gives

$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&\leq 2c \cdot n/2 + O(n) \\
&\leq 2c \cdot n/2 + c' \cdot n \\
&= (c + c')n > c \cdot n
\end{aligned}
$$