# Divide and Conquer — Merge Sort

Pontus Ekberg

Uppsala University

(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

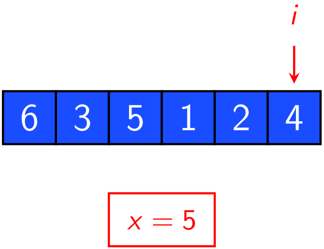
| | | | | | |
|---|---|---|---|---|---|
| 6 | 3 | 5 | 1 | 2 | 4 |

$\boxed{x = 5}$

- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else  return FALSE
```

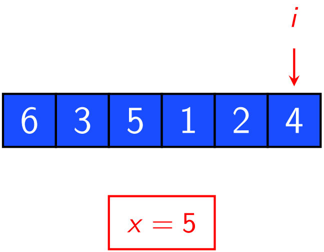



- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

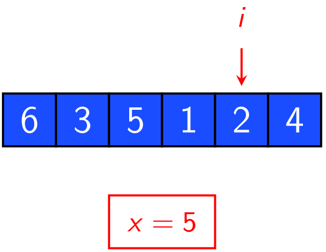



- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

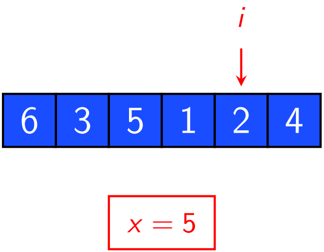



- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

SEARCH$(A, x)$
```
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else  return FALSE
```




- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

| | | | $i \downarrow$ | | |
|---|---|---|---|---|---|
| 6 | 3 | 5 | 1 | 2 | 4 |

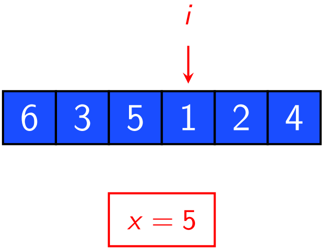
$\boxed{x = 5}$

- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- **Problem:** Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

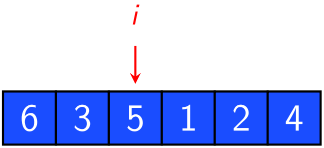



- **Incremental approach:** Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```



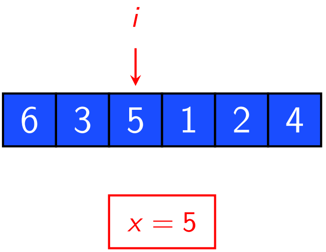

$\boxed{x = 5}$

- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```

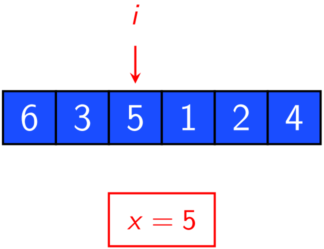



- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Sequential Search

- Problem: Check whether the value of $x$ appears in an array $A$

```
SEARCH(A, x)
i ← A.length
while A[i] ≠ x and i > 0
      do i ← i − 1
if i > 0
   then return TRUE
   else return FALSE
```



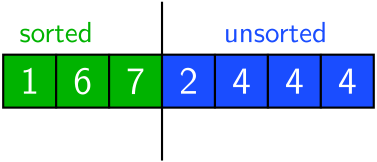

- Incremental approach: Having checked the subarray $A[i+1..n]$, we check if $A[i]$ is equal to $x$, yielding the checked subarray $A[i..n]$.

# Insertion Sort

- Problem: Sort an array $A$ of $n$ elements in non-decreasing order

INSERTION-SORT($A$)

```
for j ← 2 to A.length
    do key ← A[j]
       ▷ Insert A[j] into A[1 . . j − 1]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```




- Incremental approach: Having sorted the subarray $A[1..j-1]$, we inserted $A[j]$ into its proper place, yielding the sorted array $A[1..j]$.

# Incremental approach

- Advantages:
  - Simple and applicable to many problems
  - A good starting point to find better algorithms
- Disadvantages:
  - Will often produce less efficient algorithms
  - Less elegant and creative approach

# Another Approach


Divide et impera

# Another Approach


Divide et impera

Divide and rule / Divide and conquer

# Another Approach: The Divide-and-Conquer Methodology

- Divide the problem into a number of smaller subproblems
- Conquer the subproblems individually by solving them respectively
- Combine the solutions to the subproblems into a solution of the main problem
- Base Cases: If the size of the problem does not exceed some given threshold $n_0$, then a solution can be provided in a straightforward manner

# Example: Divide-and-Conquer Search Algorithm

- Problem: Check whether the value of $x$ appears in an array $A$
- Divide: Divide the input array into two halves of length $n/2$ each (or as close as possible).
- Conquer: Search recursively in each of the two subarrays.
- Combine: Check if the value of $x$ appears in any of the sub-arrays
- Base Cases: The size of $A$ is one. Checking whether the value of $x$ appears in $A$ is trivial.

# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

```
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```

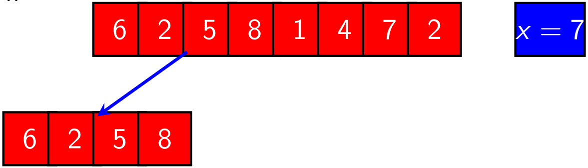
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 2 | 5 | 8 | 1 | 4 | 7 | 2 |

$x = 7$

# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

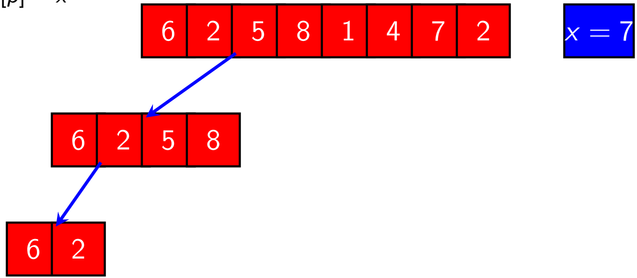
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

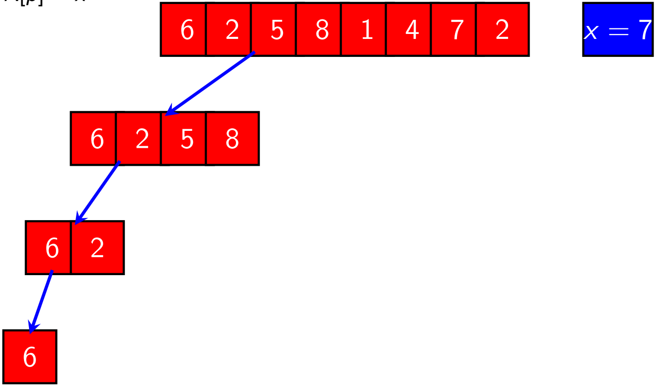
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

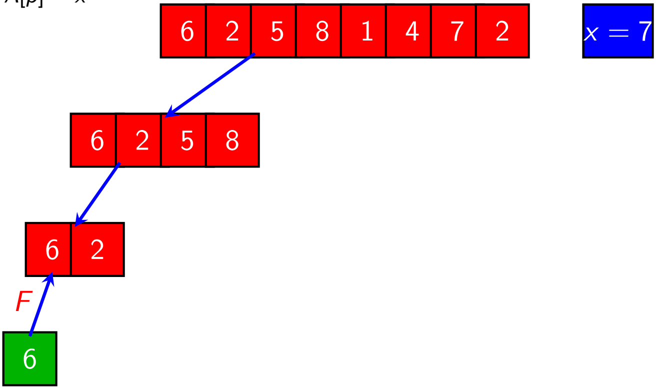
```
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

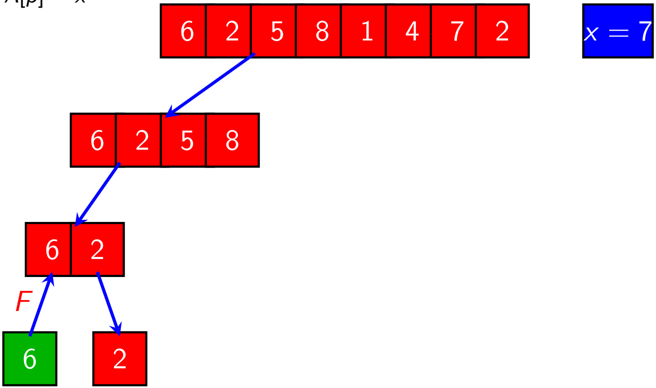
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

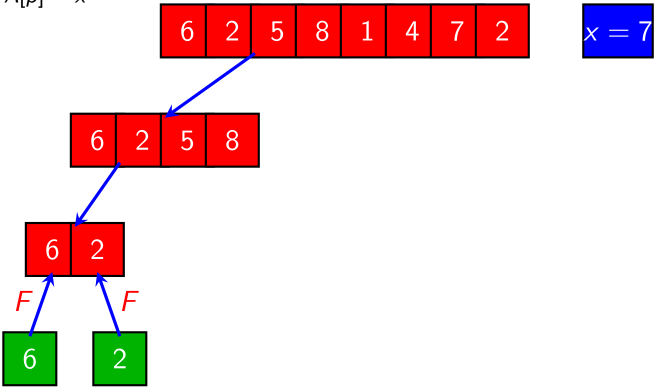
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** $\textsc{Search}$(A,p,q,x)
    *or* $\textsc{Search}$(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

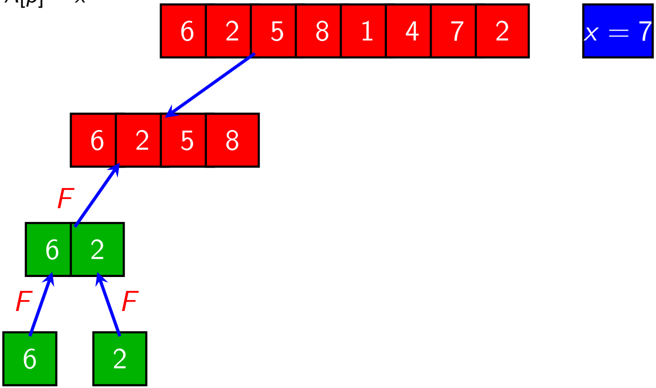
```
if p < r
   then q ← ⌊(p+r)/2⌋
        return Search(A,p,q,x)
        or Search(A,q+1,r,x)
   else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

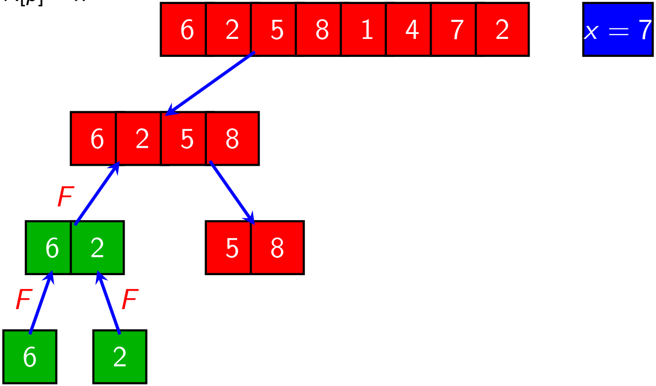
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

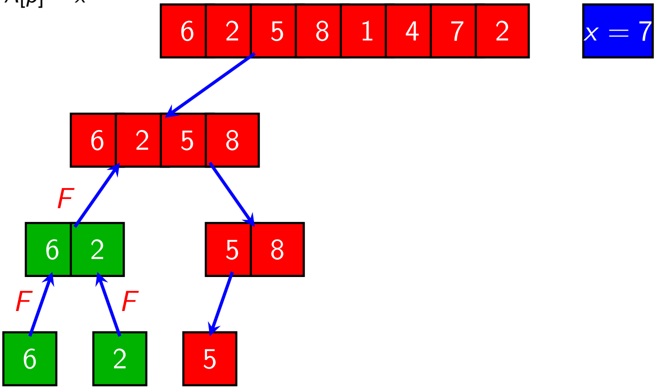
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

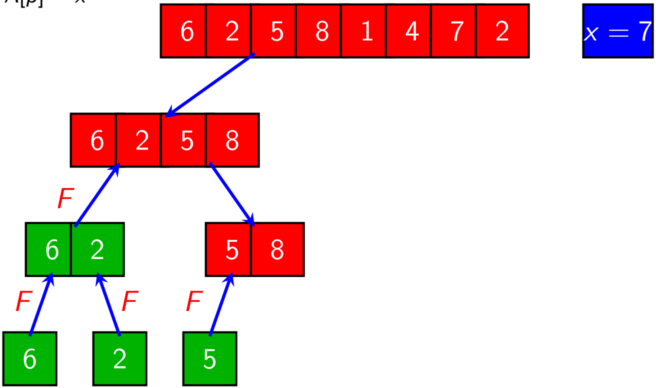
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

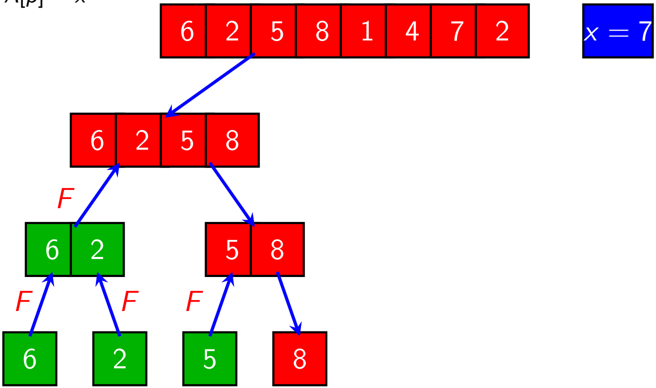
```
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

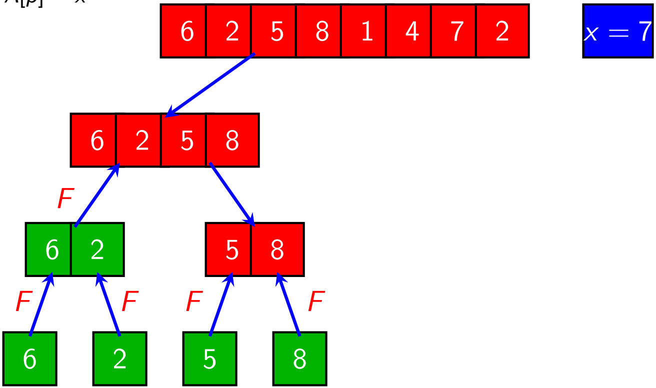
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

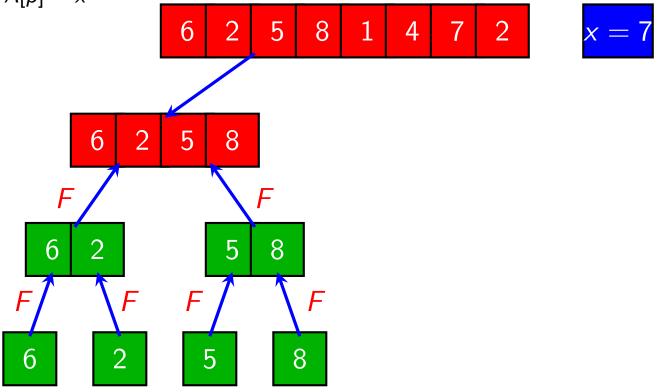
```
if p < r
    then q ← ⌊(p+r)/2⌋
        return SEARCH(A,p,q,x)
        or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

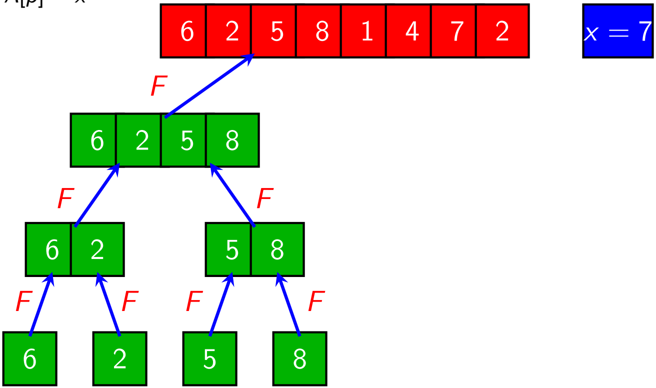
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

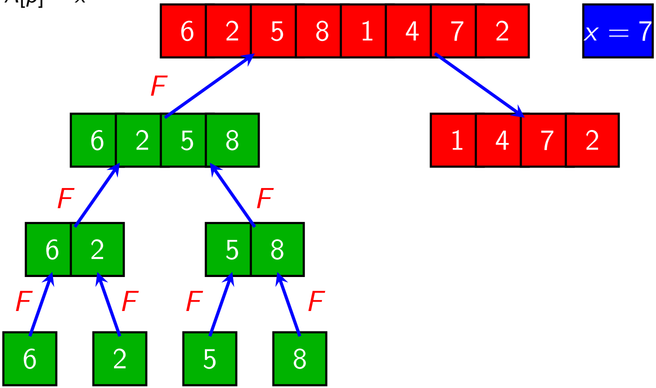
```
if p < r
    then q ← ⌊(p+r)/2⌋
        return SEARCH(A,p,q,x)
        or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

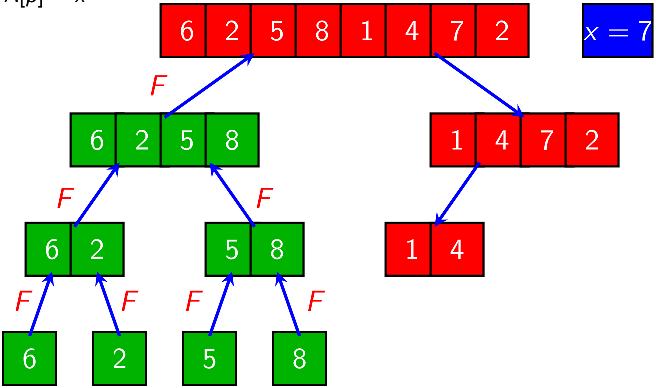
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

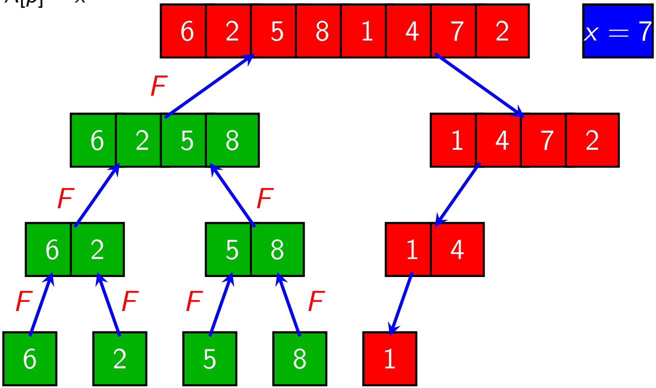
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

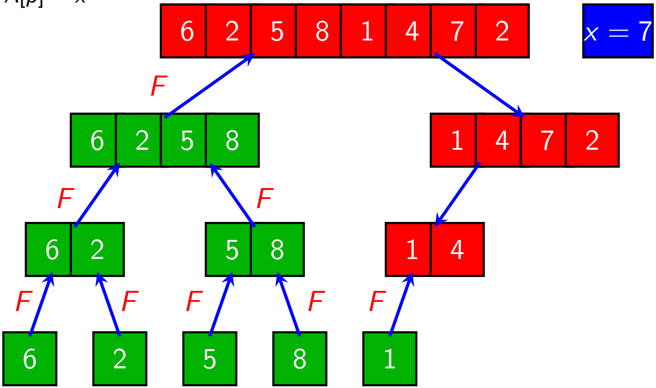
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

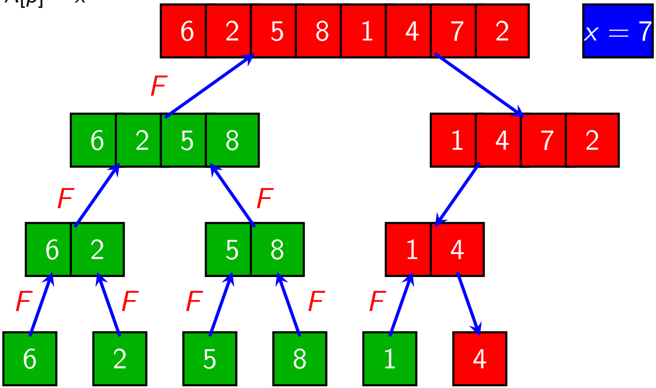
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** $\textsc{Search}$(A,p,q,x)
    *or* $\textsc{Search}$(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

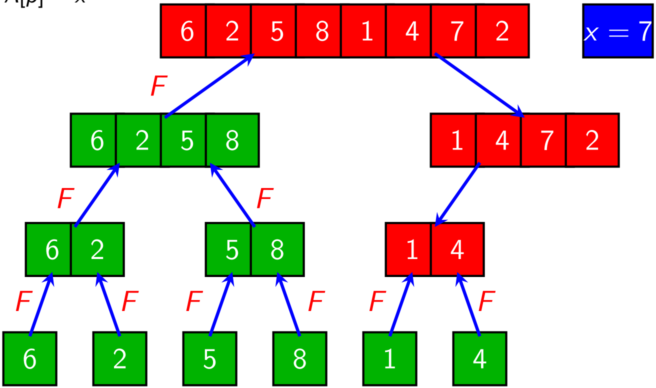
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** $\textsc{Search}$(A,p,q,x)
    *or* $\textsc{Search}$(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

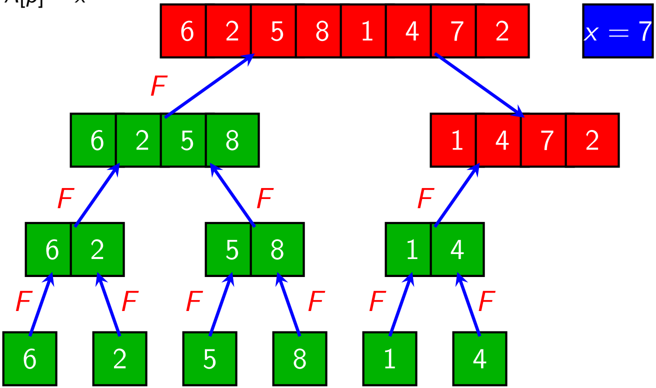
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

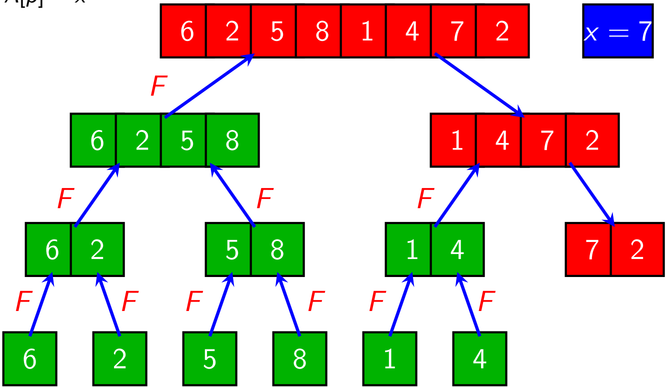
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH($A, p, r, x$)

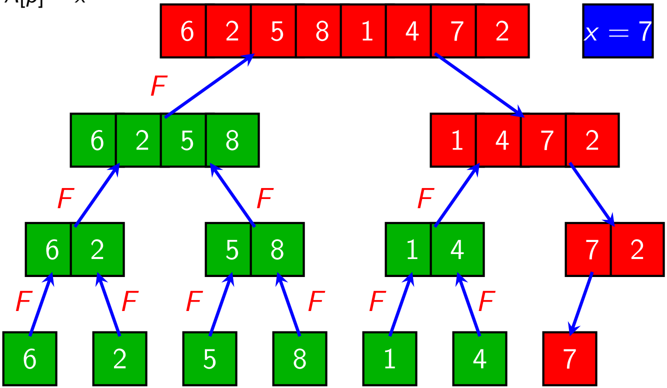
```
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

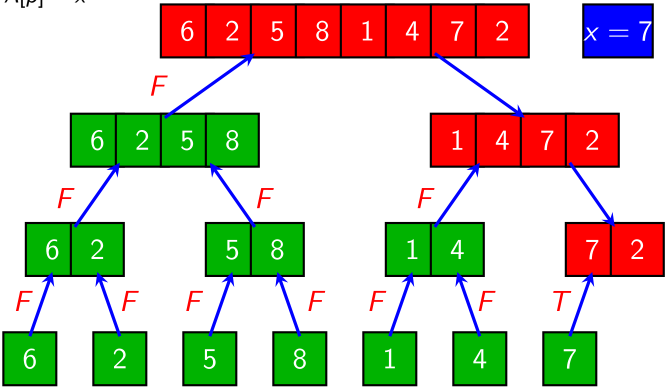
```
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```



# Divide-and-Conquer Search Algorithm

$\textsc{Search}(A, p, r, x)$

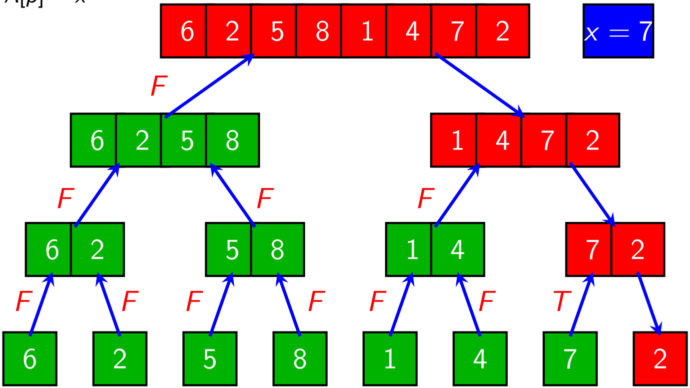
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** $\textsc{Search}$(A,p,q,x)
    *or* $\textsc{Search}$(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

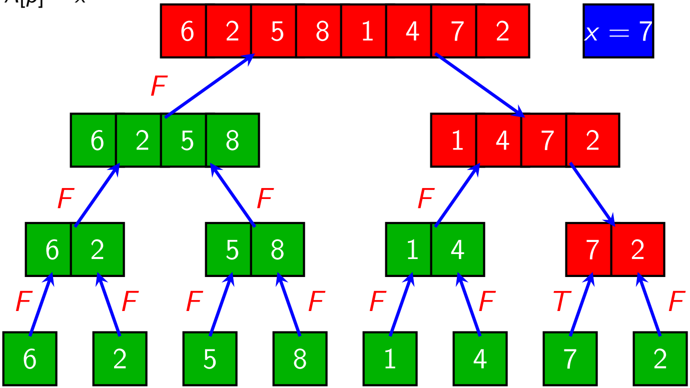
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

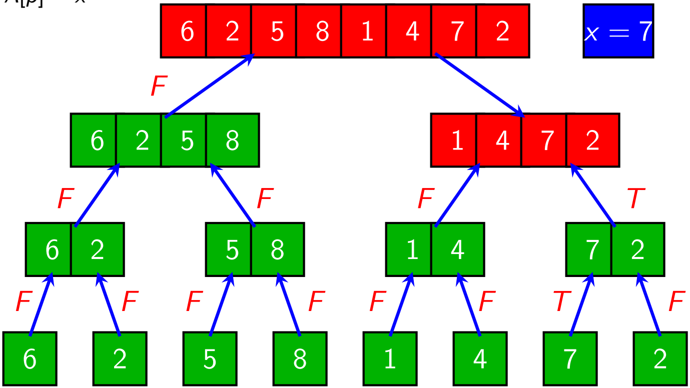
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

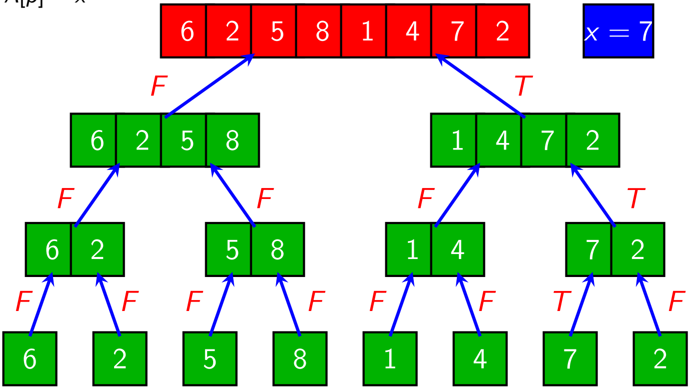
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** SEARCH(A,p,q,x)
    *or* SEARCH(A,q+1,r,x)
  **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

$\text{SEARCH}(A, p, r, x)$

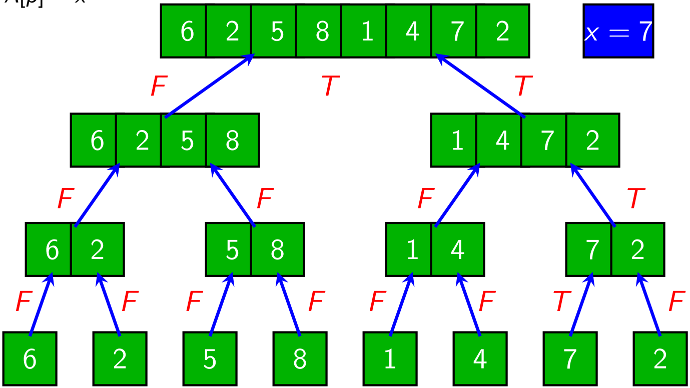
**if** $p < r$
    **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
        **return** $\text{SEARCH}$(A,p,q,x)
        *or* $\text{SEARCH}$(A,q+1,r,x)
    **else** **return** $A[p] = x$



# Divide-and-Conquer Search Algorithm

SEARCH$(A, p, r, x)$

if $p < r$
    then $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
        **return** SEARCH(A,p,q,x)
        *or* SEARCH(A,q+1,r,x)
    **else** **return** $A[p] = x$



# Merge Sort

- Problem: Sort an array $A$ of $n$ elements in non-decreasing order
- Divide: Divide the input array into two halves of length $n/2$ each (or as close as possible).
- Conquer: Sort each of the two subarrays recursively using merge sort.
- Combine: Merge the two sorted subarrays into a single array which is a sorted permutation of the original array.
- Base Cases: The size of $A$ is one. Then, $A$ is trivially sorted.

# Merge Sort

- Problem: Sort an array $A$ of $n$ elements in non-decreasing order
- Divide: Divide the input array into two halves of length $n/2$ each (or as close as possible).
- Conquer: Sort each of the two subarrays recursively using merge sort.
- Combine: Merge the two sorted subarrays into a single array which is a sorted permutation of the original array.
- Base Cases: The size of $A$ is one. Then, $A$ is trivially sorted.

# The Merge Procedure

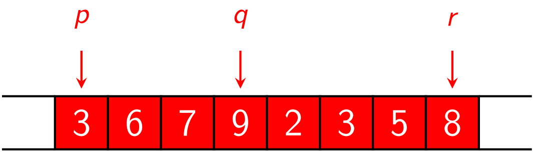
# The Merge Procedure

- Input:
  - $A[p..r]$: subarray.
  - $p \leq q < r$.
  - The two subarrays $A[p..q]$ and $A[q+1..r]$ are individually sorted.
- Task:
  - Merge the two subarrays into a single sorted array, which replaces the input subarray $A[p..q]$.

# Merge Sort

MERGE$(A, p, q, r)$

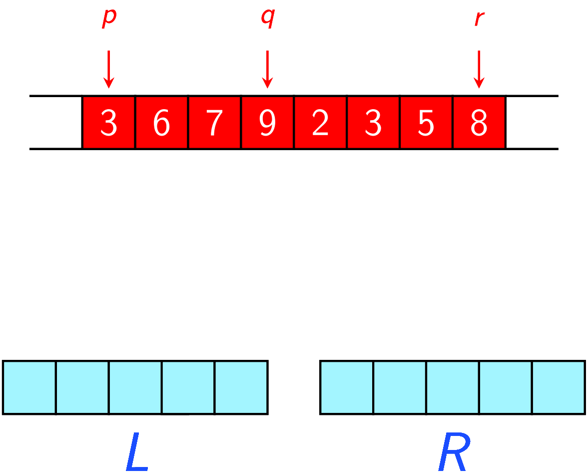
```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 . . n1 + 1]
and R[1 . . n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i − 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
                then A[k] ← L[i]
                        i ← i + 1
                else  A[k] ← R[j]
                        j ← j + 1
```



# Merge Sort

MERGE($A, p, q, r$)

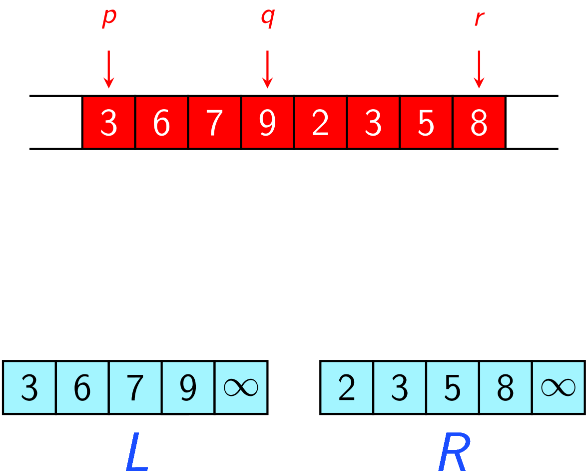
```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 . . n1 + 1]
  and R[1 . . n2 + 1]
for i ← 1 to n1
     do L[i] ← A[p + i − 1]
for j ← 1 to n2
     do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```



# Merge Sort

MERGE$(A, p, q, r)$

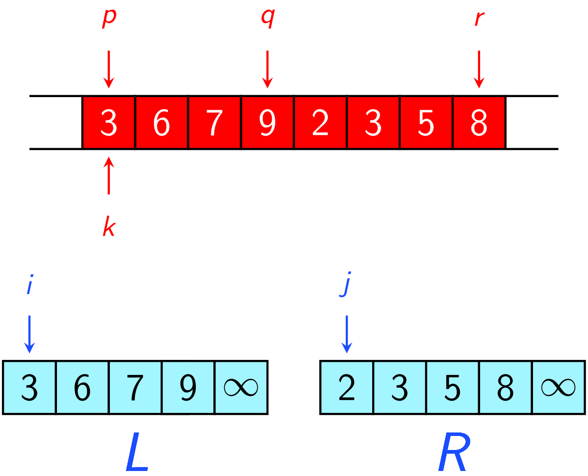
```
n_1 ← q − p + 1
n_2 ← r − q
create arrays L[1 . . n_1 + 1]
  and R[1 . . n_2 + 1]
for i ← 1 to n_1
    do L[i] ← A[p + i − 1]
for j ← 1 to n_2
    do R[j] ← A[q + j]
L[n_1 + 1] ← ∞
R[n_2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```



# Merge Sort

MERGE$(A, p, q, r)$

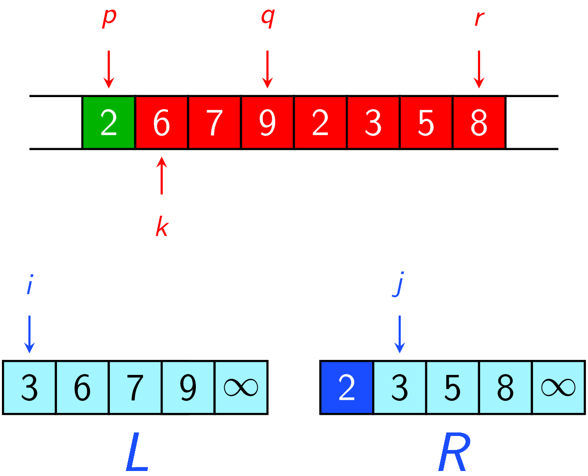
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$
and $R[1 \,..\, n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
    **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
    **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
    **do if** $L[i] \leq R[j]$
        **then** $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        **else** $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$



# Merge Sort

MERGE$(A, p, q, r)$

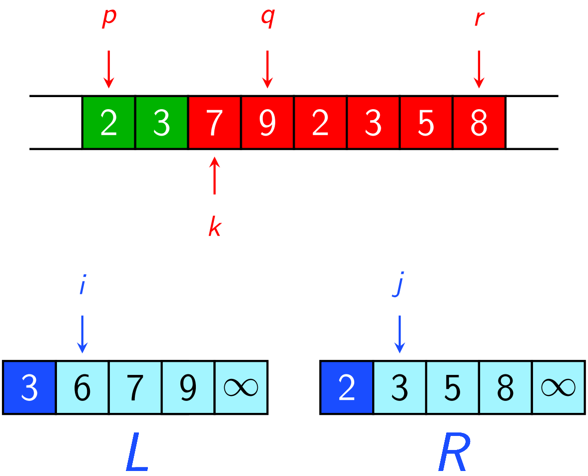
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \mathinner{..} n_1 + 1]$ and $R[1 \mathinner{..} n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
  **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
  **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
  **do if** $L[i] \leq R[j]$
    **then** $A[k] \leftarrow L[i]$
      $i \leftarrow i + 1$
    **else** $A[k] \leftarrow R[j]$
      $j \leftarrow j + 1$



# Merge Sort

MERGE$(A, p, q, r)$

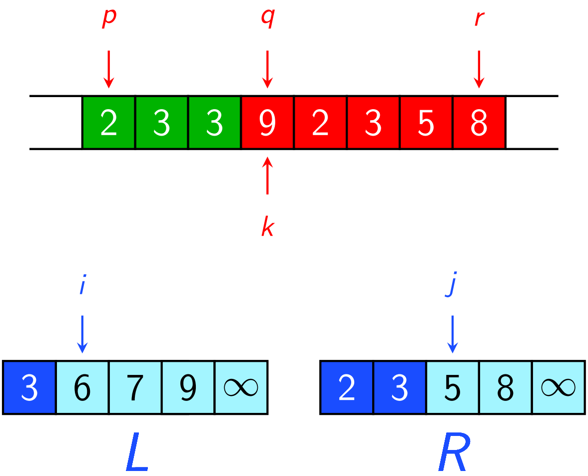
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
    **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
    **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
    **do if** $L[i] \leq R[j]$
        **then** $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        **else** $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$



# Merge Sort

MERGE($A, p, q, r$)

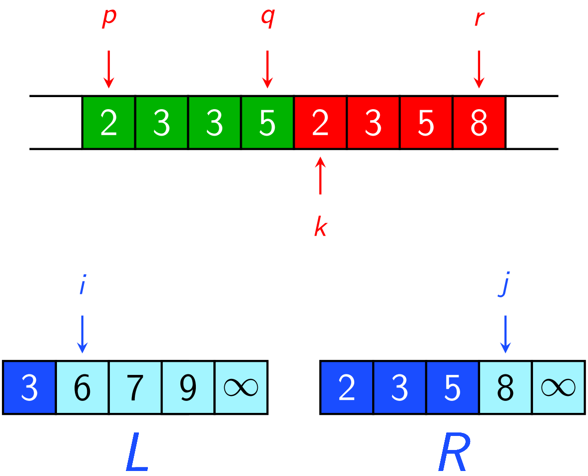
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1\,..\,n_1 + 1]$
and $R[1\,..\,n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
    **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
    **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
    **do if** $L[i] \leq R[j]$
        **then** $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        **else** $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$



# Merge Sort

MERGE($A, p, q, r$)

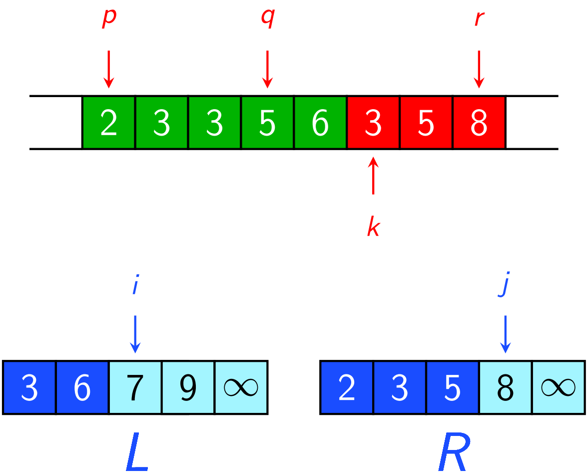
```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 . . n1 + 1]
  and R[1 . . n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
      do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```



# Merge Sort

MERGE$(A, p, q, r)$

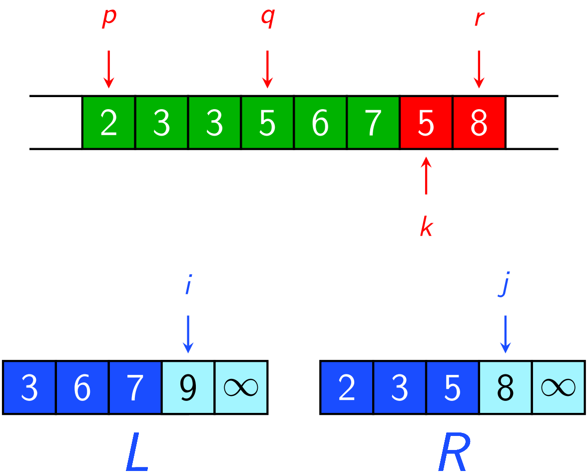
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1\,.\,.\,n_1+1]$ and $R[1\,.\,.\,n_2+1]$
**for** $i \leftarrow 1$ **to** $n_1$
    **do** $L[i] \leftarrow A[p+i-1]$
**for** $j \leftarrow 1$ **to** $n_2$
    **do** $R[j] \leftarrow A[q+j]$
$L[n_1+1] \leftarrow \infty$
$R[n_2+1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
    **do if** $L[i] \leq R[j]$
        **then** $A[k] \leftarrow L[i]$
            $i \leftarrow i+1$
        **else** $A[k] \leftarrow R[j]$
            $j \leftarrow j+1$



# Merge Sort

MERGE$(A, p, q, r)$

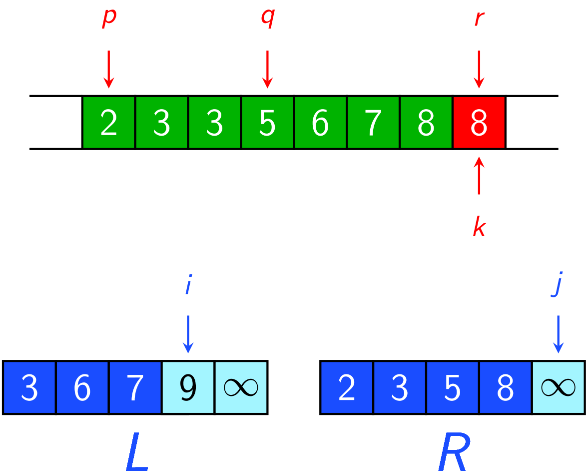
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$
and $R[1 \,..\, n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
    **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
    **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
    **do if** $L[i] \leq R[j]$
        **then** $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        **else** $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$



# Merge Sort

MERGE$(A, p, q, r)$

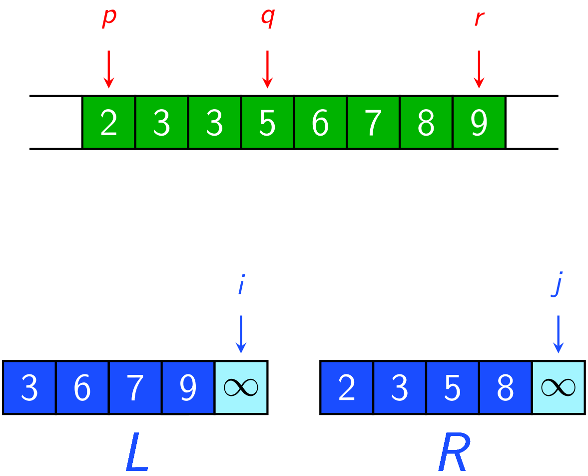
```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1]
  and R[1 .. n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
      do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```



# Merge Sort

MERGE$(A, p, q, r)$

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$
**for** $i \leftarrow 1$ **to** $n_1$
  **do** $L[i] \leftarrow A[p + i - 1]$
**for** $j \leftarrow 1$ **to** $n_2$
  **do** $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**for** $k \leftarrow p$ **to** $r$
  **do if** $L[i] \leq R[j]$
    **then** $A[k] \leftarrow L[i]$
      $i \leftarrow i + 1$
    **else** $A[k] \leftarrow R[j]$
      $j \leftarrow j + 1$



# Worst-Case Complexity Analysis of the Merge Procedure

- Let $n = r - p + 1$

MERGE$(A, p, q, r)$

$n_1 \leftarrow q - p + 1$<br>
$n_2 \leftarrow r - q$<br>
create arrays $L[1 \,..\, n_1 + 1]$<br>
and $R[1 \,..\, n_2 + 1]$<br>
**for** $i \leftarrow 1$ **to** $n_1$<br>
    **do** $L[i] \leftarrow A[p + i - 1]$<br>
**for** $j \leftarrow 1$ **to** $n_2$<br>
    **do** $R[j] \leftarrow A[q + j]$<br>
$L[n_1 + 1] \leftarrow \infty$<br>
$R[n_2 + 1] \leftarrow \infty$<br>
$i \leftarrow 1$<br>
$j \leftarrow 1$<br>
**for** $k \leftarrow p$ **to** $r$<br>
    **do if** $L[i] \leq R[j]$<br>
        **then** $A[k] \leftarrow L[i]$<br>
            $i \leftarrow i + 1$<br>
        **else** $A[k] \leftarrow R[j]$<br>
            $j \leftarrow j + 1$

# Worst-Case Complexity Analysis of the Merge Procedure

- Let $n = r - p + 1$
- Each of lines 1-3 and 8-11 takes constant time.

MERGE($A, p, q, r$)

1 $n_1 \leftarrow q - p + 1$

2 $n_2 \leftarrow r - q$

3 create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$

4 **for** $i \leftarrow 1$ **to** $n_1$

5     **do** $L[i] \leftarrow A[p + i - 1]$

6 **for** $j \leftarrow 1$ **to** $n_2$

7     **do** $R[j] \leftarrow A[q + j]$

8 $L[n_1 + 1] \leftarrow \infty$

9 $R[n_2 + 1] \leftarrow \infty$

10 $i \leftarrow 1$

11 $j \leftarrow 1$

12 **for** $k \leftarrow p$ **to** $r$

13     **do if** $L[i] \leq R[j]$

14         **then** $A[k] \leftarrow L[i]$

15             $i \leftarrow i + 1$

16         **else** $A[k] \leftarrow R[j]$

17             $j \leftarrow j + 1$

# Worst-Case Complexity Analysis of the Merge Procedure

- Let $n = r - p + 1$
- Each of lines 1-3 and 8-11 takes constant time.
- The **for** loops of lines 4-7 take $\Theta(n_1 + n_2) = \Theta(n)$

MERGE$(A, p, q, r)$

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 create arrays $L[1 \mathinner{..} n_1 + 1]$ and $R[1 \mathinner{..} n_2 + 1]$
4 **for** $i \leftarrow 1$ **to** $n_1$
5     **do** $L[i] \leftarrow A[p + i - 1]$
6 **for** $j \leftarrow 1$ **to** $n_2$
7     **do** $R[j] \leftarrow A[q + j]$
8 $L[n_1 + 1] \leftarrow \infty$
9 $R[n_2 + 1] \leftarrow \infty$
10 $i \leftarrow 1$
11 $j \leftarrow 1$
12 **for** $k \leftarrow p$ **to** $r$
13     **do if** $L[i] \leq R[j]$
14         **then** $A[k] \leftarrow L[i]$
15             $i \leftarrow i + 1$
16         **else** $A[k] \leftarrow R[j]$
17             $j \leftarrow j + 1$

# Worst-Case Complexity Analysis of the Merge Procedure

- Let $n = r - p + 1$
- Each of lines 1-3 and 8-11 takes constant time.
- The **for** loops of lines 4-7 take $\Theta(n_1 + n_2) = \Theta(n)$
- The **for** loop of lines 12-17 takes $n$-iterations, each of which takes constant time.

MERGE$(A, p, q, r)$

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
4 **for** $i \leftarrow 1$ **to** $n_1$
5     **do** $L[i] \leftarrow A[p + i - 1]$
6 **for** $j \leftarrow 1$ **to** $n_2$
7     **do** $R[j] \leftarrow A[q + j]$
8 $L[n_1 + 1] \leftarrow \infty$
9 $R[n_2 + 1] \leftarrow \infty$
10 $i \leftarrow 1$
11 $j \leftarrow 1$
12 **for** $k \leftarrow p$ **to** $r$
13     **do if** $L[i] \leq R[j]$
14         **then** $A[k] \leftarrow L[i]$
15             $i \leftarrow i + 1$
16         **else** $A[k] \leftarrow R[j]$
17             $j \leftarrow j + 1$

# Worst-Case Complexity Analysis of the Merge Procedure

- Let $n = r - p + 1$
- Each of lines 1-3 and 8-11 takes constant time.
- The **for** loops of lines 4-7 take $\Theta(n_1 + n_2) = \Theta(n)$
- The **for** loop of lines 12-17 takes $n$-iterations, each of which takes constant time.

The Merge Procedure runs in $\Theta(n)$

MERGE$(A, p, q, r)$

1 $n_1 \leftarrow q - p + 1$<br>
2 $n_2 \leftarrow r - q$<br>
3 create arrays $L[1 \mathinner{.\,.} n_1 + 1]$<br>
    and $R[1 \mathinner{.\,.} n_2 + 1]$<br>
4 **for** $i \leftarrow 1$ **to** $n_1$<br>
5     **do** $L[i] \leftarrow A[p + i - 1]$<br>
6 **for** $j \leftarrow 1$ **to** $n_2$<br>
7<br>
8 $L[n_1 + 1] \leftarrow \infty$<br>
9 $R[n_2 + 1] \leftarrow \infty$<br>
10 $i \leftarrow 1$<br>
11 $j \leftarrow 1$<br>
12 **for** $k \leftarrow p$ **to** $r$<br>
13     **do if** $L[i] \leq R[j]$<br>
14    **then** $A[k] \leftarrow L[i]$<br>
15         $i \leftarrow i + 1$<br>
16    **else** $A[k] \leftarrow R[j]$<br>
17         $j \leftarrow j + 1$

# Merge Sort

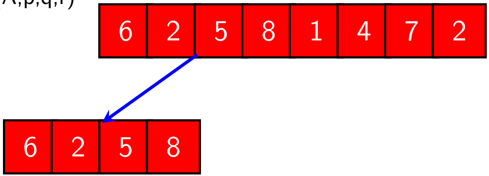
```
MERGE-SORT(A, p, r)
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 2 | 5 | 8 | 1 | 4 | 7 | 2 |

# Merge Sort

MERGE-SORT($A, p, r$)

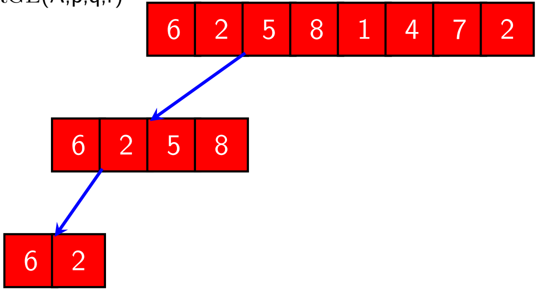
```
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

**if** $p < r$
    **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT$(A, p, r)$

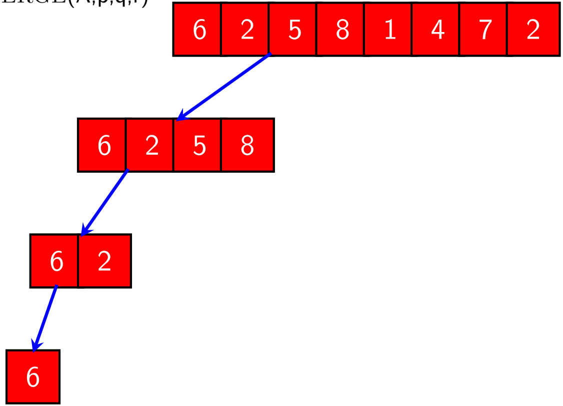
**if** $p < r$

  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$

    MERGE-SORT(A,p,q)

    MERGE-SORT(A,q+1,r)

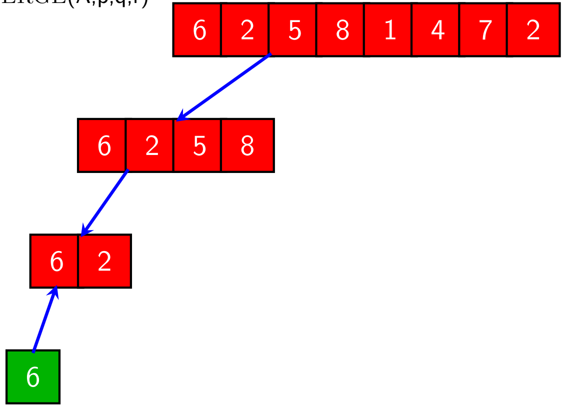
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT$(A, p, r)$

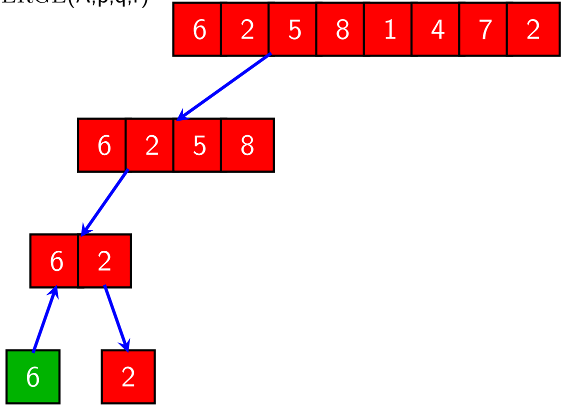
```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

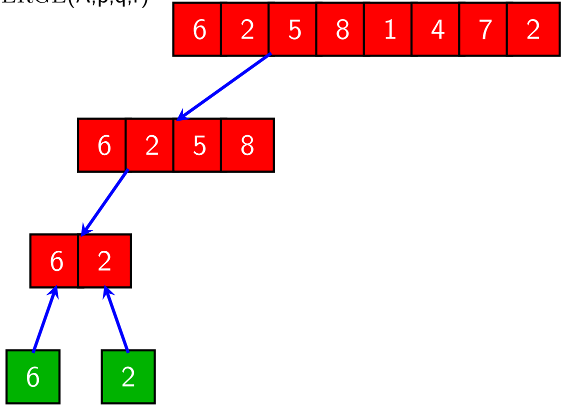
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

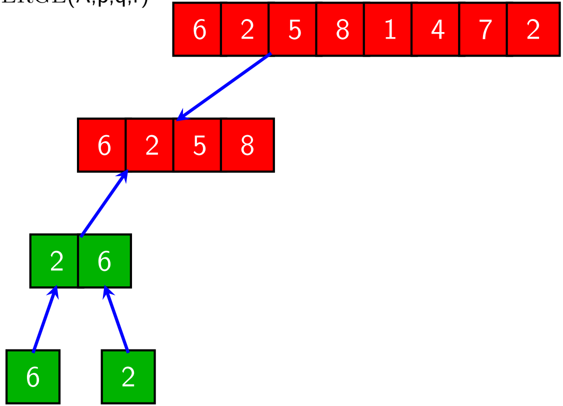
```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

$\text{MERGE-SORT}(A, p, r)$

```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

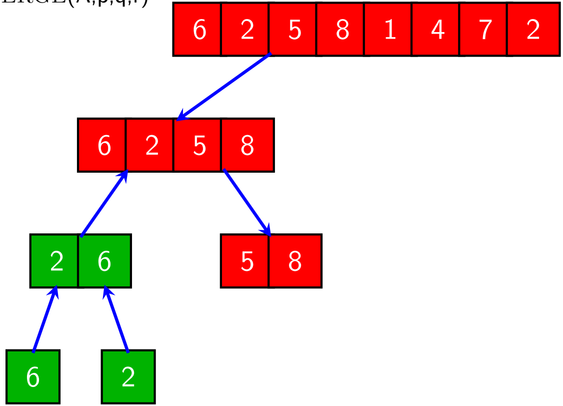
**if** $p < r$

  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$

    MERGE-SORT(A,p,q)

    MERGE-SORT(A,q+1,r)

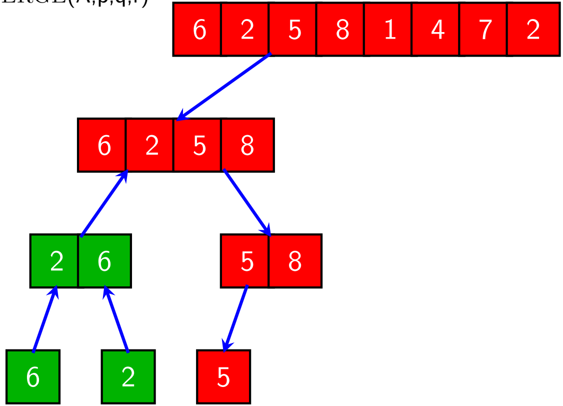
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

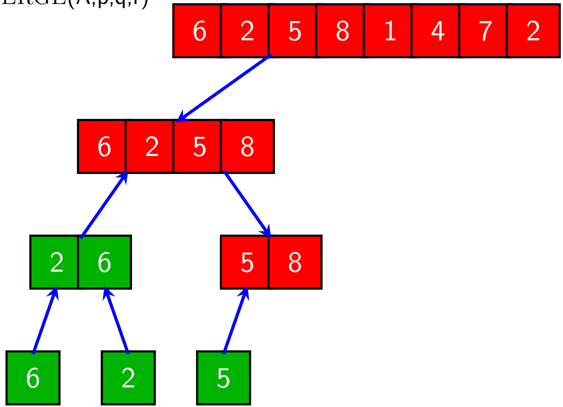
```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

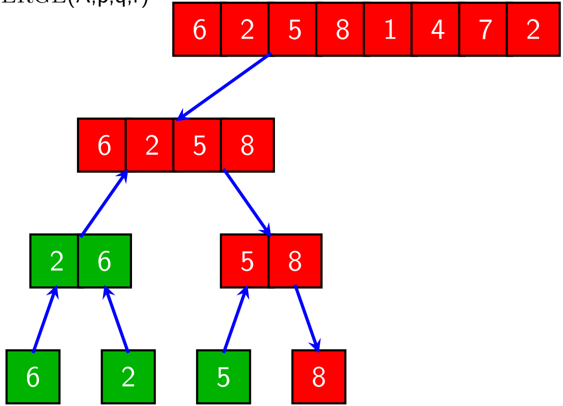
```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

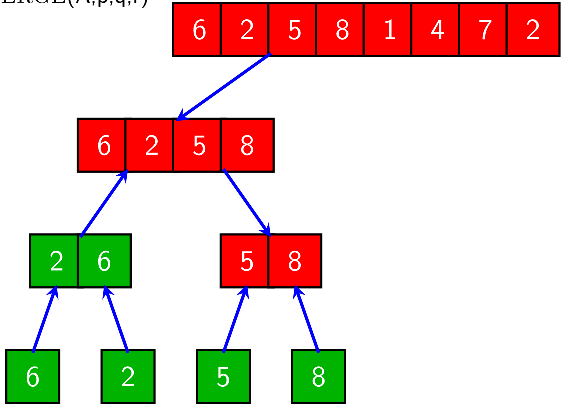
```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

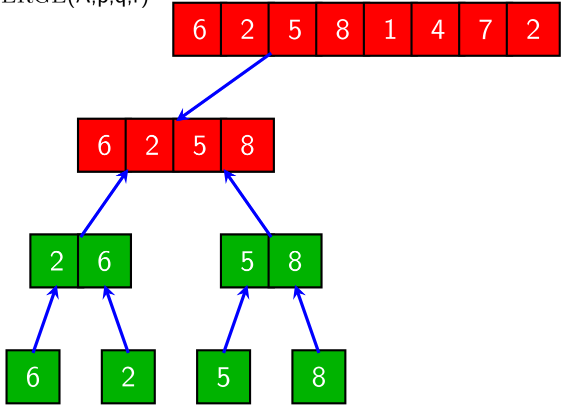
**if** $p < r$

  **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$

    MERGE-SORT(A,p,q)

    MERGE-SORT(A,q+1,r)

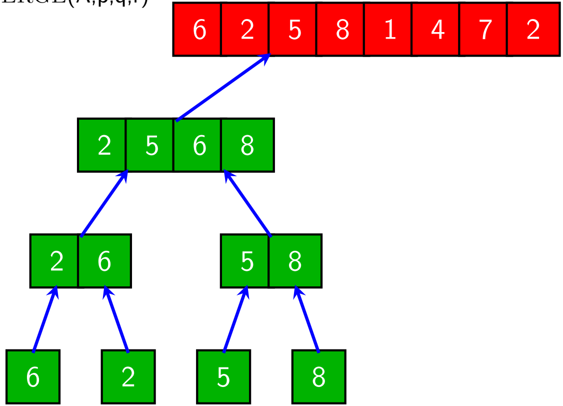
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT$(A, p, r)$

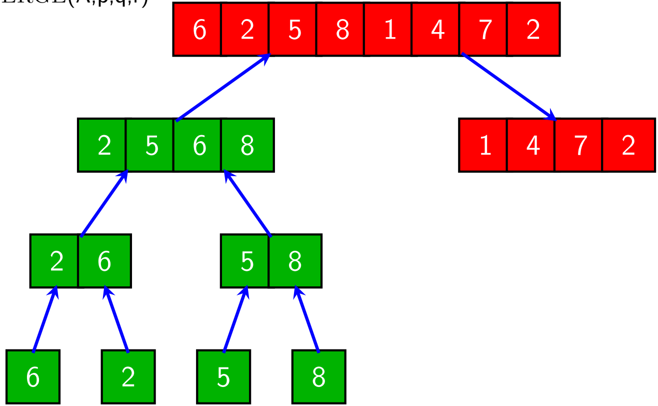
```
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

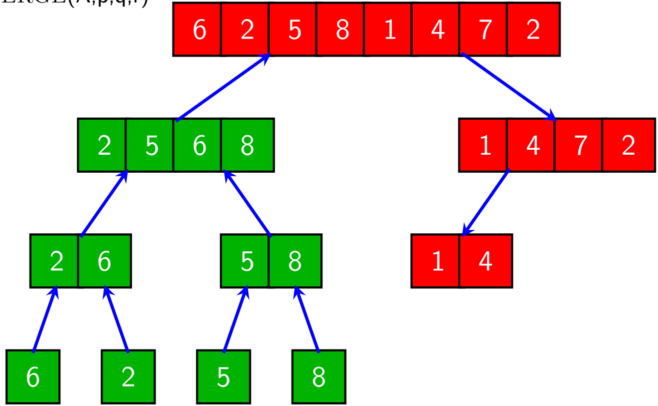
```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

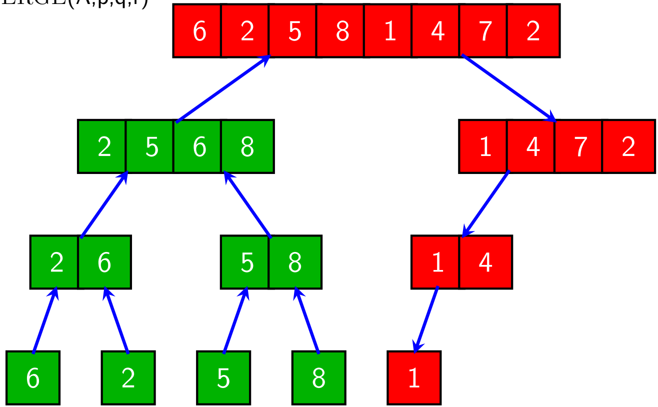
**if** $p < r$
    **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

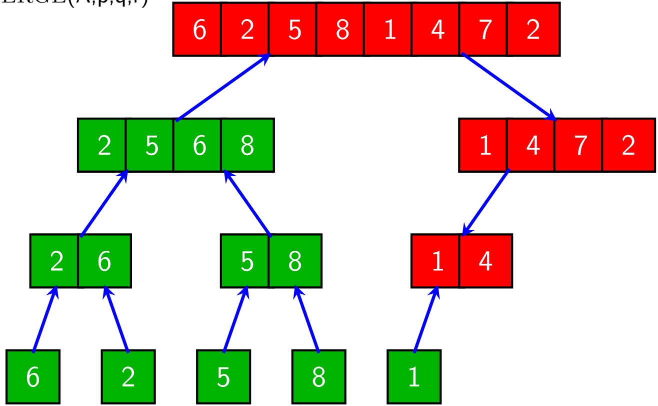
**if** $p < r$
    **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

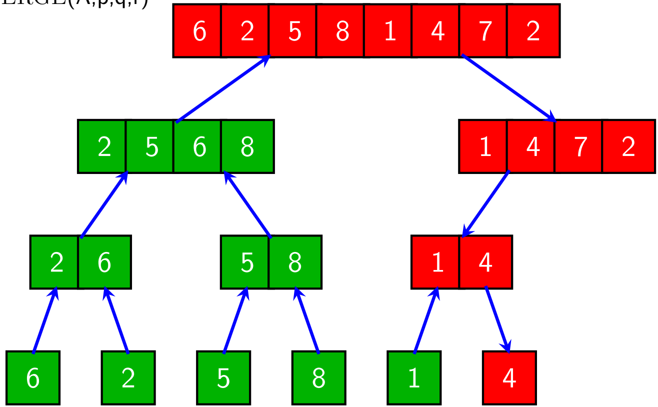
**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

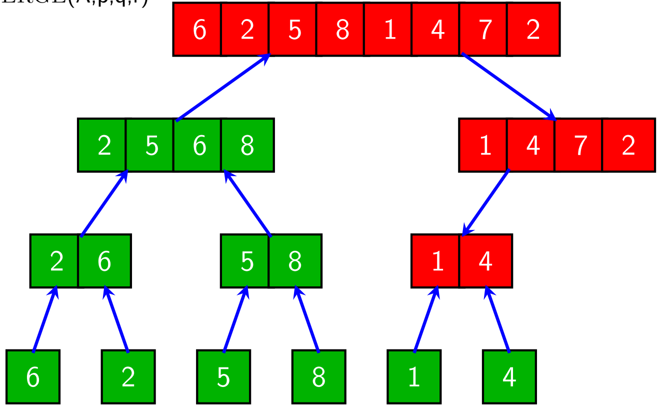
```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

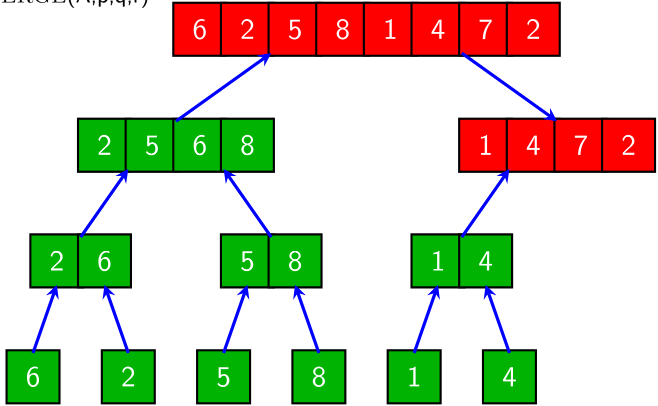
**if** $p < r$

  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$

    MERGE-SORT(A,p,q)

    MERGE-SORT(A,q+1,r)

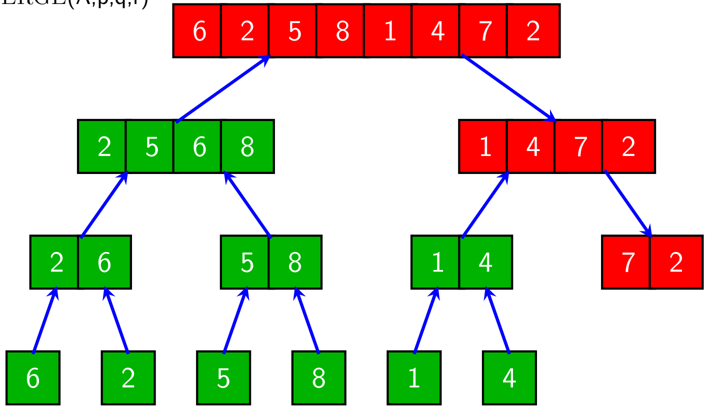
    MERGE(A,p,q,r)



# Merge Sort

MERGE-SORT($A, p, r$)

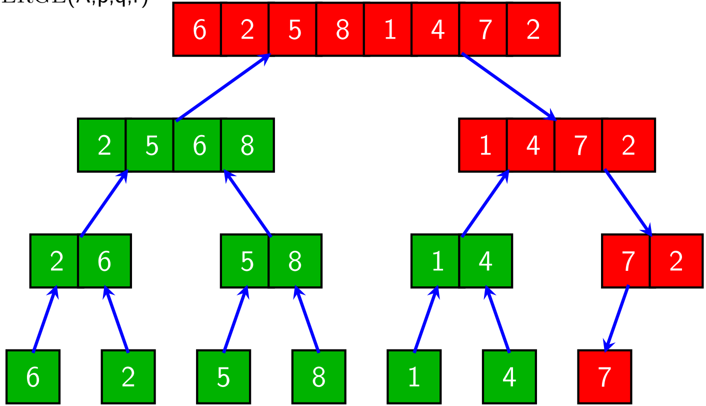
**if** $p < r$
  **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)



# Merge Sort

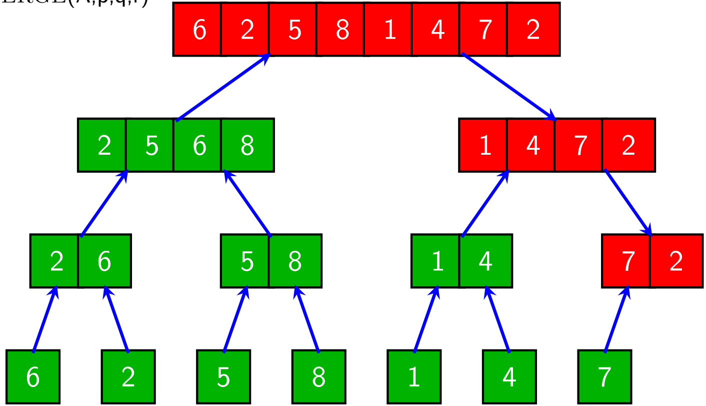
MERGE-SORT($A$, $p$, $r$)

```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



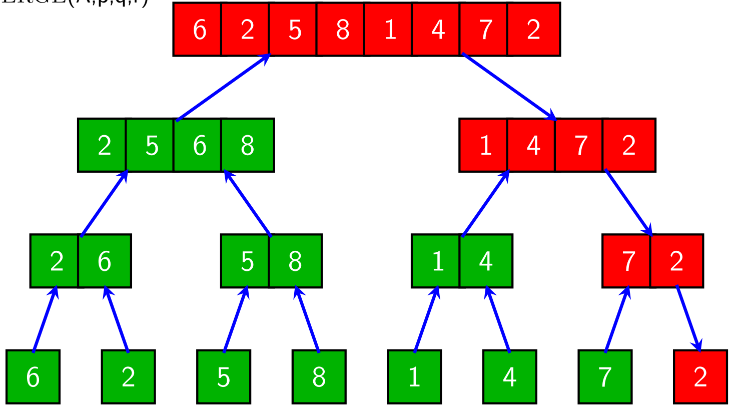
# Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



MERGE-SORT($A, p, r$)

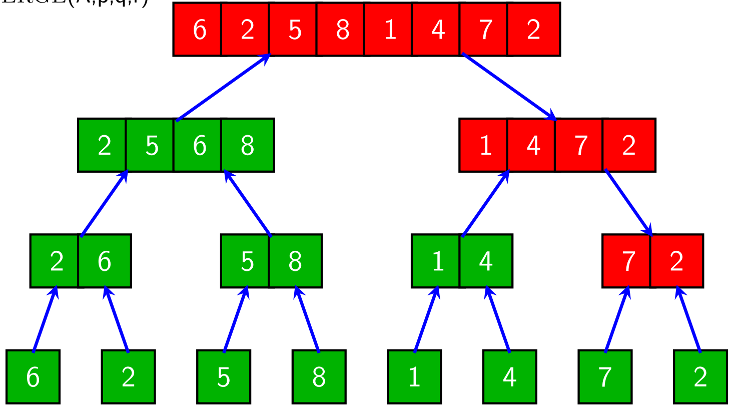
```
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

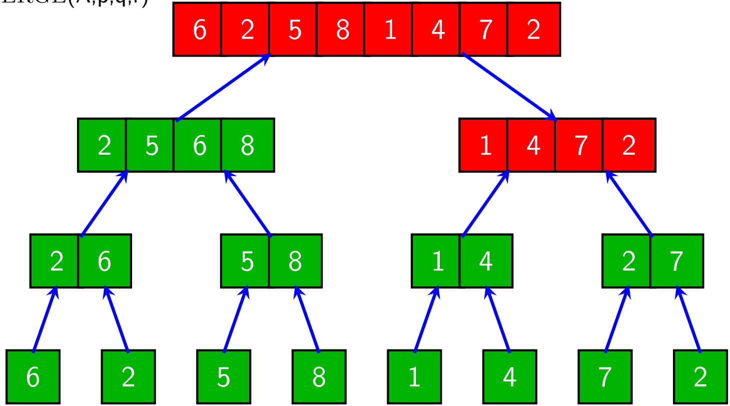
```
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

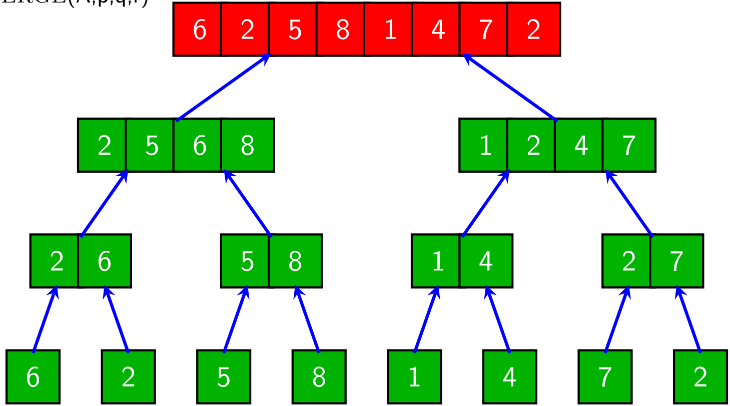
```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

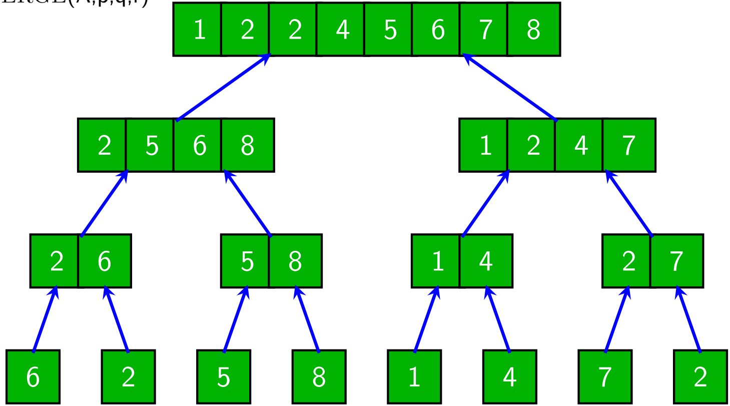
```
if p < r
   then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



# Merge Sort

MERGE-SORT($A, p, r$)

**if** $p < r$
  **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)



# Complexity Analysis of Divide-and-Conquer Algorithms

- Divide the problem into a number of smaller subproblems
- Conquer the subproblems individually by solving them respectively
- Combine the solutions to the subproblems into a solution of the main problem
- Base Cases: If the size of the problem does not exceed some given threshold $n_0$, then a solution can be provided in a straightforward manner

# Complexity Analysis of Divide-and-Conquer Algorithms

- $T(n)$: running time on a problem instance of size $n$.
- $a$: number of subproblems.
- $\frac{n}{b}$: size of each subproblem.
- $D(n)$: cost of dividing the problem into subproblems.
- $C(n)$: cost of combining the solutions to the subproblems into the solution of the original problem.
- The cost of each base case $(n \leq n_0)$ is constant.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + C(n) + D(n) & \text{if } n > n_0 \end{cases}$

  This is a recurrence equation.

# Example: Divide-and-Conquer Search Algorithm

- $a = 2$: number of subproblems.
- $\frac{n}{2}$ $(b = 2)$ : size of each subproblem.
- $D(n) = \Theta(1)$: cost of dividing the problem into subproblems.
- $C(n) = \Theta(1)$: cost of combining the sub-solutions
- Base case $n_0 = 1$
- $T(n) =$
$$\begin{cases} \Theta(1) & \text{if } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + C(n) + D(n) & \text{if } n > n_0 \end{cases}$$

$\text{Search}(A, p, r, x)$

**if** $p < r$
  **then** $q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
    **return** Search(A,p,q,x)
    *or* Search(A,q+1,r,x)
  **else** **return** $A[p] = x$

# Example: Divide-and-Conquer Search Algorithm

- $a = 2$: number of subproblems.
- $\frac{n}{2}$ $(b = 2)$ : size of each subproblem.
- $D(n) = \Theta(1)$: cost of dividing the problem into subproblems.
- $C(n) = \Theta(1)$: cost of combining the sub-solutions
- Base case $n_0 = 1$
- $T(n) =$
$$\begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + C(n) + D(n) & \text{if } n > 1 \end{cases}$$

```
SEARCH(A, p, r, x)
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```

# Example: Divide-and-Conquer Search Algorithm

- $a = 2$: number of subproblems.
- $\frac{n}{2}$ $(b = 2)$ : size of each subproblem.
- $D(n) = \Theta(1)$: cost of dividing the problem into subproblems.
- $C(n) = \Theta(1)$: cost of combining the sub-solutions
- Base case $n_0 = 1$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$

```
SEARCH(A, p, r, x)
if p < r
    then q ← ⌊(p+r)/2⌋
         return SEARCH(A,p,q,x)
         or SEARCH(A,q+1,r,x)
    else return A[p] = x
```

# Solving More Recurrences

Consider the runtime of Divide-and-Conquer Search Algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

# Solving More Recurrences

Consider the runtime of Divide-and-Conquer Search Algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

How can we construct an tight/upper/lower bound closed form of $T(n)$?

# Solving More Recurrences

Consider the runtime of Divide-and-Conquer Search Algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

How can we construct an tight/upper/lower bound closed form of $T(n)$?

We consider three different methods:

1. The substitution method
2. The recursion-tree method
3. The master method

# Method 1: Substitution

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

The substitution method consists in:

- Guess a solution
- Verify the correctness of the solution using mathematical induction

# Method 1: Substitution

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

We rewrite $T(n)$ as follows:

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

# Method 1: Substitution

We guess that $T(n) = O(n)$, and try to show that for all $n \geq n_0 = 1$, $T(n) \leq an - c$ with $a = 2c$.

- Base Case: $(n = n_0 = 1)$:
  $T(n) = T(1) \leq c = a \cdot 1 - c$

- Induction Step: Let $n > n_0$.

  Assume: $T(i) \leq ai - c$ for all $n_0 \leq i < n$

  Show $T(n) \leq an - c$

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + c \\ &\leq 2(a\tfrac{n}{2} - c) + c \qquad \text{(by the I.H.)} \\ &= an - c \end{aligned}$$

# Method 2: Recursive Trees

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

This method consists in visualizing the recursion as a tree where:

- Each node represents the costs one sub-problems.
- The sum of all node costs within a level gives the total cost of that level.
- The sum of all per-level costs gives the total cost.

# Method 2: Recursive Trees

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

# Method 2: Recursive Trees

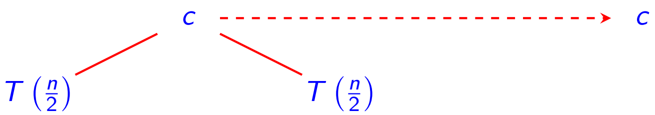
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

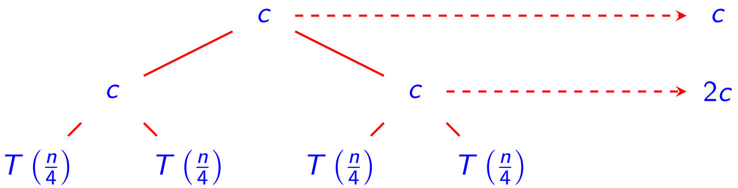
# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$



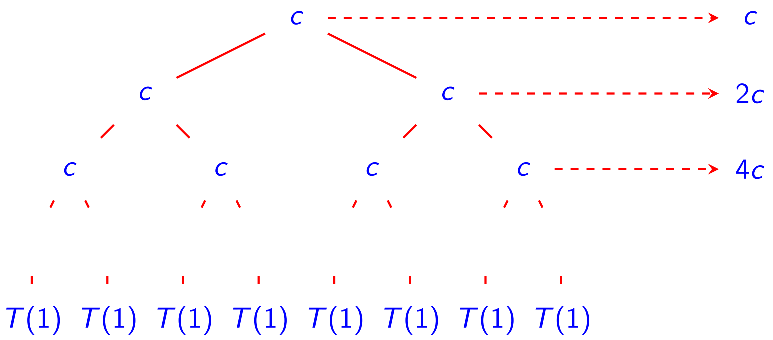
# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$



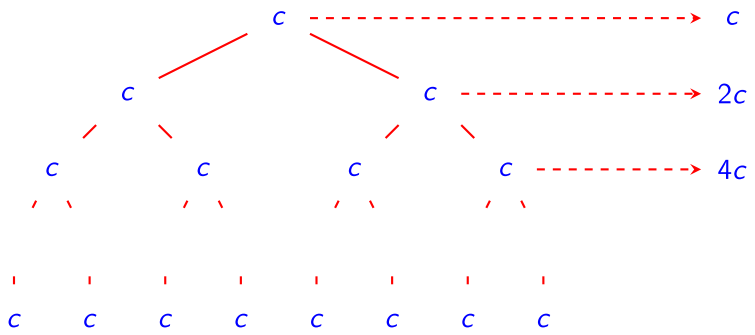
# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$



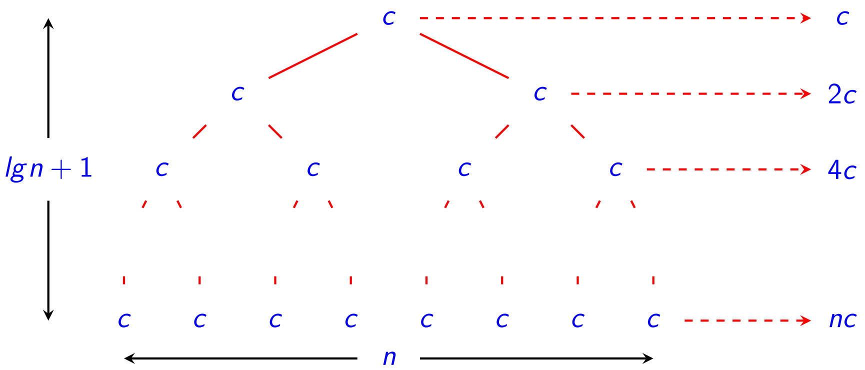
# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$



# Method 2: Recursive Trees

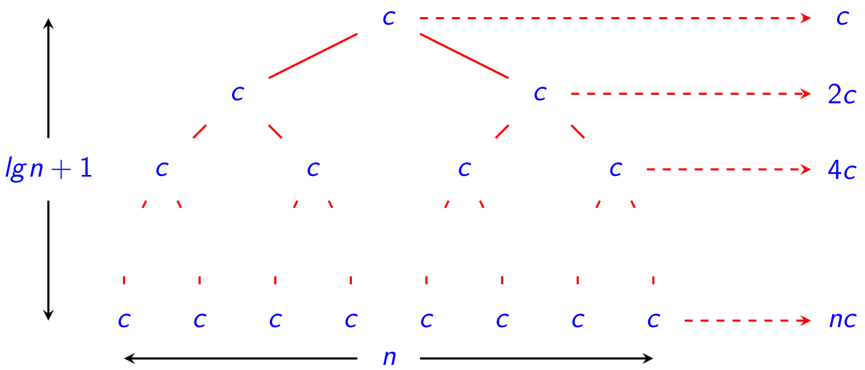
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$



# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

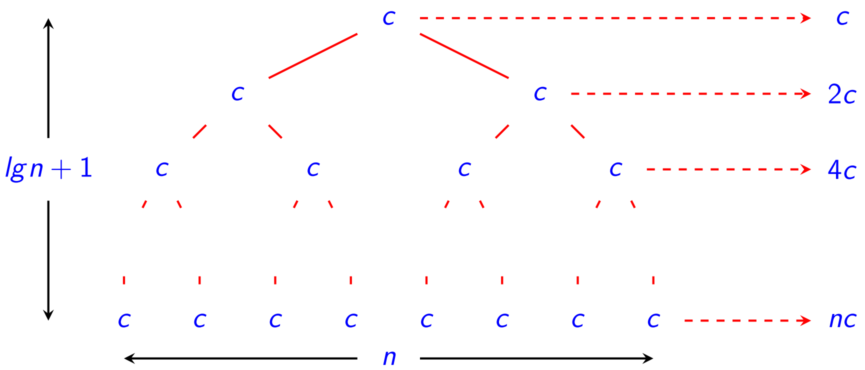
$$T(n) = (2n - 1)c$$



# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$$T(n) = \Theta(n)$$

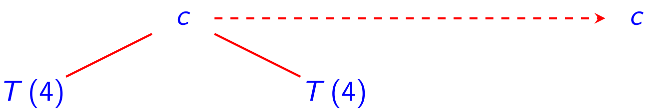


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n-1)c$

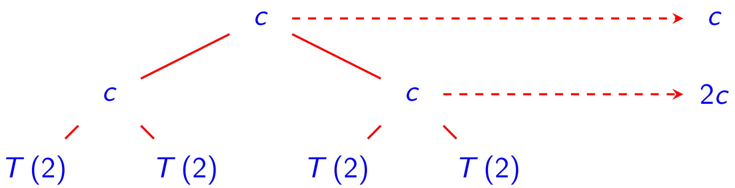


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n - 1)c$

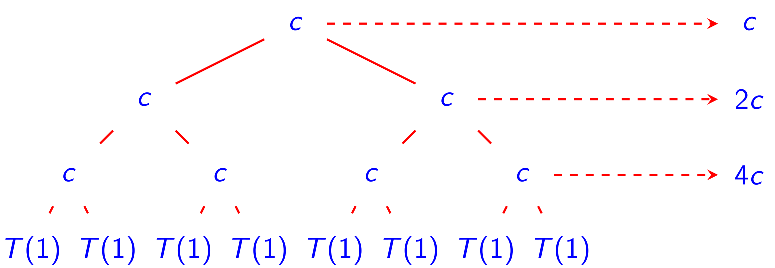


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n - 1)c$

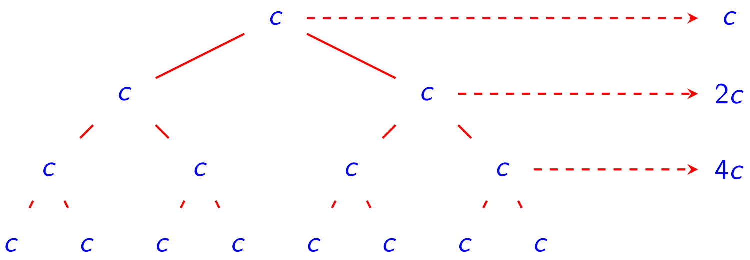


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n - 1)c$

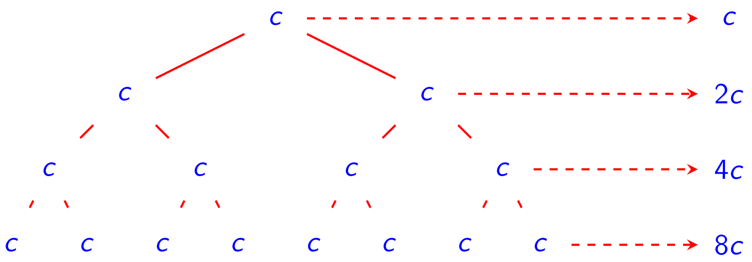


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n-1)c$

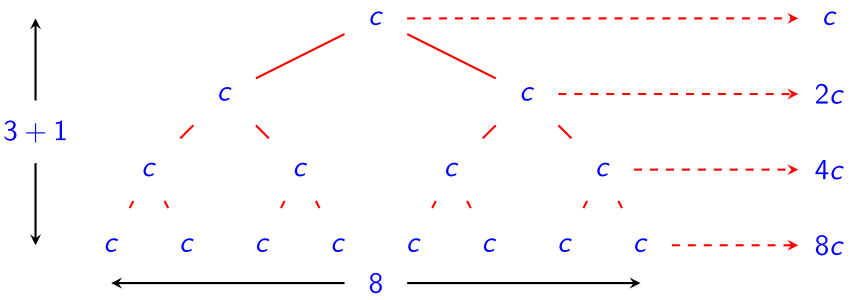


# Method 2: Recursive Trees

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = (2n - 1)c$



# The Master Method and Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Then, $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$
- If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log(n))$
- If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$, and if $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some constant $c < 1$ and sufficiently large $n$, then $T(n) = \Theta(f(n))$.

# Method 3: Master Theorem

Consider the recurrence equation of the search algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Then, we have $a = 2$, $b = 2$, $f(n) = \Theta(1)$. Case 1 of the Master Theorem applies since $f(n) = O(n^{\log_2(2)-\epsilon}) = O(1)$ where $\epsilon = 1$. Thus we have:

$$T(n) = \Theta(n^{\log_a(b)}) = \Theta(n)$$

.

Now let's use the three methods to find the complexity of Merge Sort, starting with the recursion tree method.

# Complexity Analysis of Divide-and-Conquer Algorithms

- $T(n)$: running time on a problem instance of size $n$.
- $a$: number of subproblems.
- $\frac{n}{b}$: size of each subproblem.
- $D(n)$: cost of dividing the problem into subproblems.
- $C(n)$: cost of combining the solutions to the subproblems into the solution of the original problem.
- The cost of each base case ($n \leq n_0$) is constant.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + C(n) + D(n) & \text{if } n > n_0 \end{cases}$

# Example: Merge Sort

- $a = 2$: number of subproblems.
- $\frac{n}{2}$ $(b = 2)$ : size of each subproblem.
- $D(n) = \Theta(1)$: cost of dividing the problem into subproblems.
- $C(n) = \Theta(n)$: cost of the procedure Merge
- Base case $n_0 = 1$
- $T(n) =$
$$\begin{cases} \Theta(1) & \text{if } n \leq n_0 \\ a \cdot T\left(\frac{n}{b}\right) + C(n) + D(n) & \text{if } n > n_0 \end{cases}$$

MERGE-SORT$(A, p, r)$

```
if p < r
    then q ← ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

# Example: Merge Sort

- $a = 2$: number of subproblems.
- $\frac{n}{2}$ $(b = 2)$ : size of each subproblem.
- $D(n) = \Theta(1)$: cost of dividing the problem into subproblems.
- $C(n) = \Theta(n)$: cost of the procedure Merge
- Base case $n_0 = 1$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$

MERGE-SORT$(A, p, r)$

```
if p < r
    then q ← ⌊(p+r)/2⌋
         MERGE-SORT(A,p,q)
         MERGE-SORT(A,q+1,r)
         MERGE(A,p,q,r)
```

# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

# The Recursion Tree method for Merge Sort

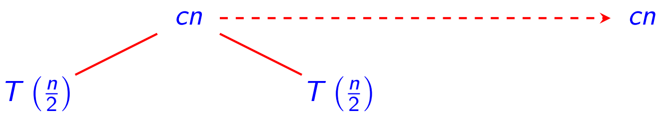
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

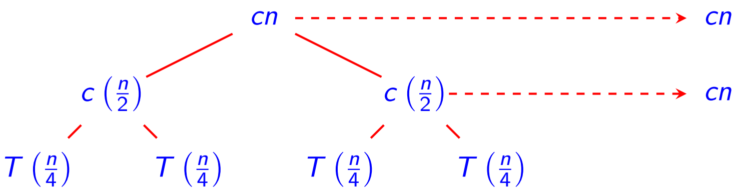
# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$



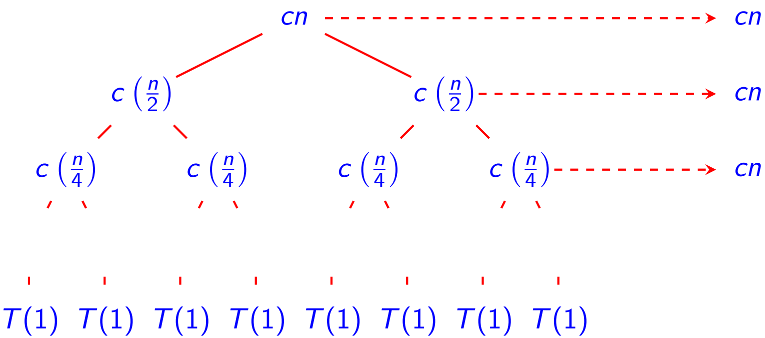
# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$



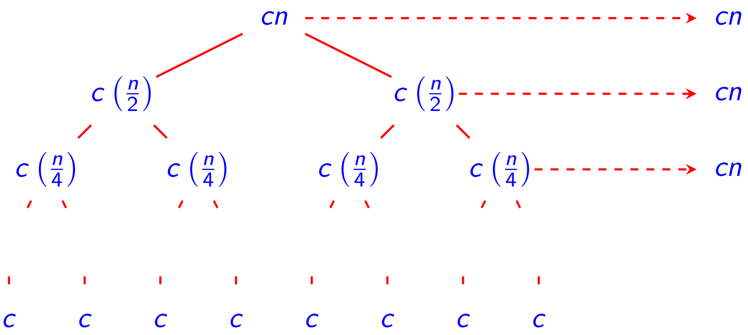
# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$



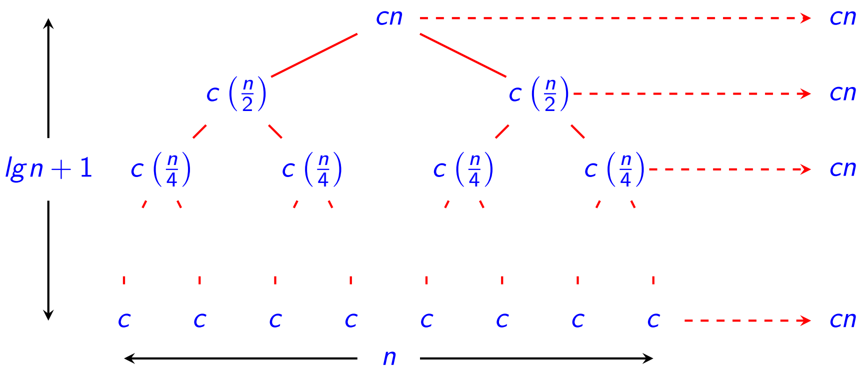
# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$



# The Recursion Tree method for Merge Sort

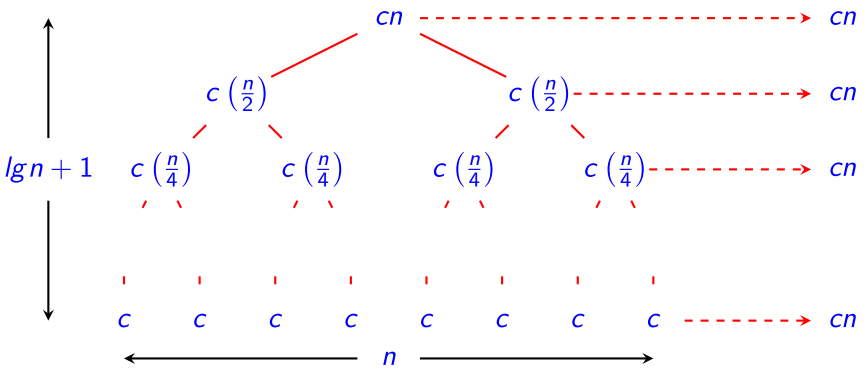
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$



# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

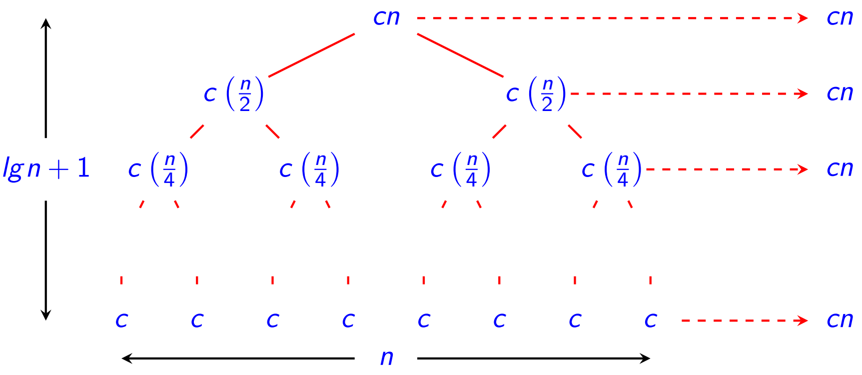
$$T(n) = cn\lg n + cn$$



# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$$T(n) = \Theta(n lg\, n)$$

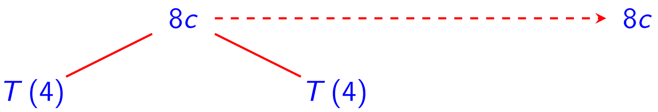


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn \lg n + cn$

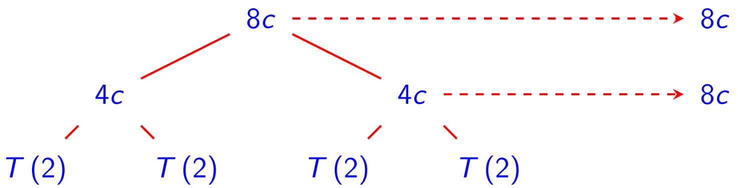


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn \lg n + cn$

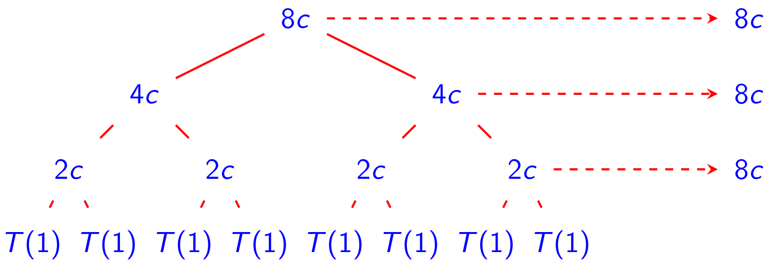


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn\lg n + cn$

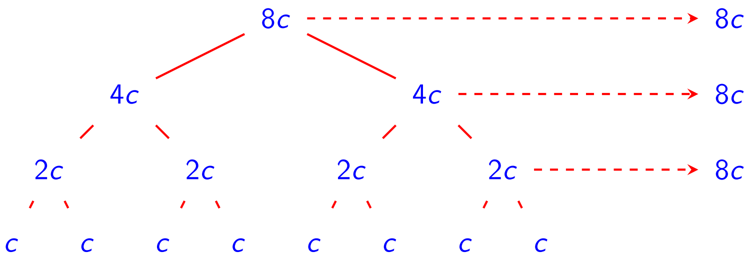


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn\lg n + cn$

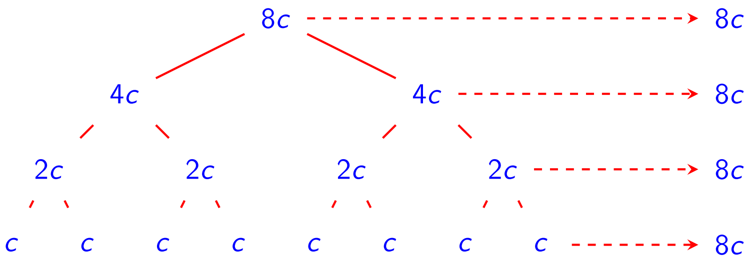


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn\lg n + cn$

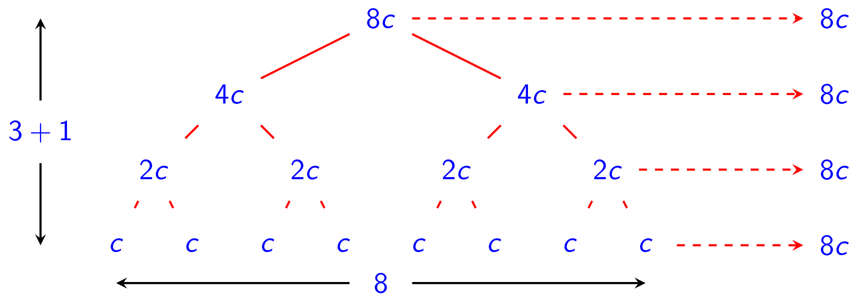


# The Recursion Tree method for Merge Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

$n = 8$

$T(n) = cn\lg n + cn$



# The Master Method and Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Then, $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$
- If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log(n))$
- If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$, and if $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some constant $c < 1$ and sufficiently large $n$, then $T(n) = \Theta(f(n))$.

# The Master Method for Merge Sort

Consider the recurrence equation of Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Then, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$. Case 2 of the Master Theorem applies since $f(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$. Thus we have:

$$T(n) = \Theta(n^{\log_a(b)} \cdot \log(n)) = \Theta(n \cdot \log(n))$$

# The Substitution Method for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The substitution method consists in:

- Guess a solution
- Verify the correctness of the solution using mathematical induction

# The Substitution Method for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We rewrite $T(n)$ as follows:

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

and try to find an asymptotic upper bound on $T(n)$.

# The Substitution Method for Merge Sort

We guess that $T(n) = O(n \cdot \log(n))$, and try to show that for all $n \geq n_0 = 2$, $0 \leq T(n) \leq a \cdot n \cdot \log(n)$ with $a = 2c$.

- Base Case ($n = n_0$):
  $T(n_0) = T(2) = 2T(1) + 2c = 4c \leq a \cdot 2 \cdot \log(2)$.

- Induction Step: Let $n > n_0$.

  Assume (the I.H.): $T(i) \leq a \cdot i \cdot \log(i)$ for all $i < n$

  Show: $T(n) \leq a \cdot n \cdot \log(n)$

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn && \text{(by definition of } T(n)) \\
&\leq 2(a \cdot \tfrac{n}{2} \cdot \log(\tfrac{n}{2})) + cn && \text{(by the I.H.)} \\
&\leq a \cdot n \cdot \log(n) - an + cn \\
&\leq a \cdot n \cdot \log(n)
\end{aligned}
$$

# Can we do asymptotically better?

Can we sort with less than $n \log n$ comparison operations in the worst case?

# Can we do asymptotically better?

Can we sort with less than $n \log n$ comparison operations in the worst case?

No!

The worst-case run time of *any* comparison-based sorting algorithm is $\Omega(n \log n)$

# Can we do asymptotically better?

Can we sort with less than $n \log n$ comparison operations in the worst case?

No!

The worst-case run time of *any* comparison-based sorting algorithm is $\Omega(n \log n)$

(But for sorting certain things we can do better by not using plain comparison operations.)