Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

$i\ j$

17	37	15	29	38	45	25	64	69	76	94	92	75	82	85

- To partition the array into two parts, we only need $O(1)$ extra space.

partition(A, ℓ, r)

$p \leftarrow$ random integer between ℓ and r, swap $A[p]$ and $A[\ell]$
$i \leftarrow \ell, j \leftarrow r$
while true **do**
  while $i < j$ and $A[i] < A[j]$ do $j \leftarrow j - 1$
  if $i = j$ **then** break
  swap $A[i]$ and $A[j]$; $i \leftarrow i + 1$
  while $i < j$ and $A[i] < A[j]$ **do** $i \leftarrow i + 1$
  if $i = j$ **then** break
  swap $A[i]$ and $A[j]$; $j \leftarrow j - 1$
return i

In-Place Implementation of Quick-Sort

quicksort(A, ℓ, r)

1: **if** $\ell \geq r$ **then return**
2: $m \leftarrow$ patition(A, ℓ, r)
3: quicksort$(A, \ell, m-1)$
4: quicksort$(A, m+1, r)$

- To sort an array A of size n, call quicksort$(A, 1, n)$.

Note: We pass the array A by reference, instead of by copying.

Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

Merge-Sort is Not In-Place

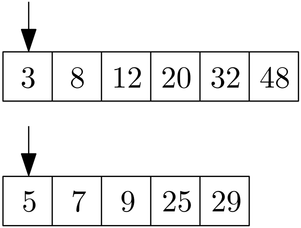
- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

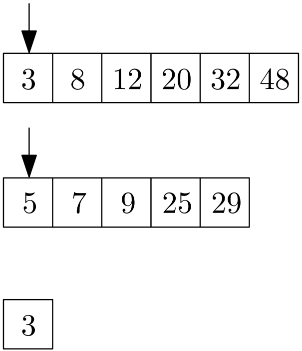
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



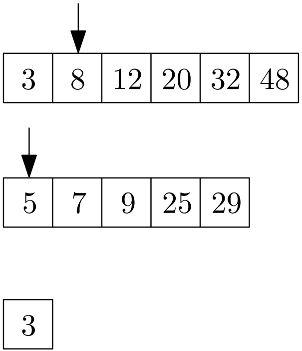
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

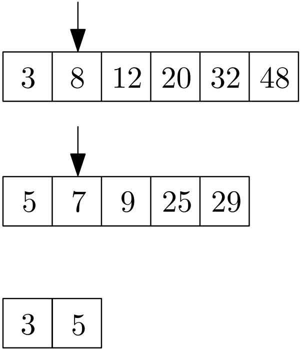
3	8	12	20	32	48

5	7	9	25	29

3	5

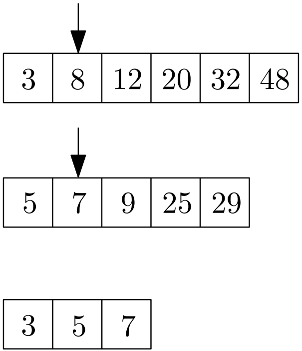
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



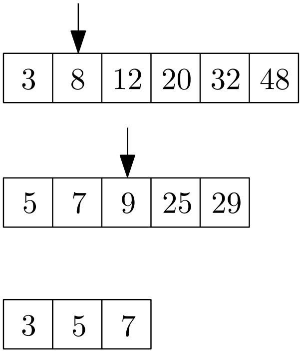
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



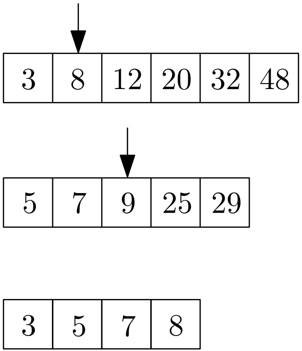
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



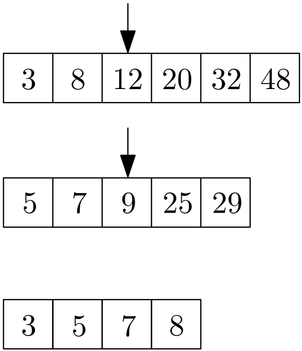
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



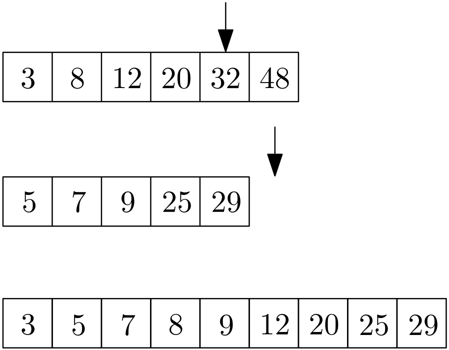
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays



Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

3	8	12	20	32	48

5	7	9	25	29

3	5	7	8	9	12	20	25	29	32	48

Outline

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We can not use "internal structures" of the elements

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

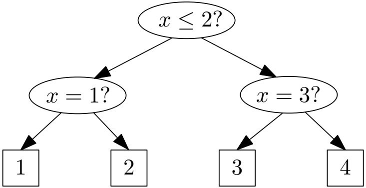
A: $\lceil \log_2 N \rceil$.

Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number x in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob "yes/no" questions about x.

Q: How many questions do you need to ask Bob in order to know x?

A: $\lceil \log_2 N \rceil$.



Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Q: How many questions do you need to ask in order to get the permutation π?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob "yes/no" questions about π.

Q: How many questions do you need to ask in order to get the permutation π?

A: $\log_2 n! = \Theta(n \lg n)$

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Q: How many questions do you need to ask in order to get the permutation π?

Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation π over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob questions of the form "does i appear before j in π?"

Q: How many questions do you need to ask in order to get the permutation π?

A: At least $\log_2 n! = \Theta(n \lg n)$

Outline

Selection Problem

Input: a set A of n numbers, and $1 \leq i \leq n$

Output: the i-th smallest number in A

Selection Problem

Input: a set A of n numbers, and $1 \leq i \leq n$

Output: the i-th smallest number in A

- Sorting solves the problem in time $O(n \lg n)$.

Selection Problem

Input: a set A of n numbers, and $1 \leq i \leq n$

Output: the i-th smallest number in A

- Sorting solves the problem in time $O(n \lg n)$.
- Our goal: $O(n)$ running time

Recall: Quicksort with Median Finder

quicksort(A, n)

if $n \leq 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
$B_L \leftarrow$ quicksort(A_L, A_L.size) ▷ Conquer
$B_R \leftarrow$ quicksort(A_R, A_R.size) ▷ Conquer
$t \leftarrow$ number of times x appear A
return the array obtained by concatenating B_L, the array containing t copies of x, and B_R

Selection Algorithm with Median Finder

selection(A, n, i)

if $n = 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L$.size **then**
    return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
    return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
    return x

Selection Algorithm with Median Finder

selection(A, n, i)

if $n = 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L$.size **then**
  return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
  return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
  return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$

Selection Algorithm with Median Finder

selection(A, n, i)

if $n = 1$ **then return** A
$x \leftarrow$ lower median of A
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \le A_L$.size **then**
    return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
    return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
    return x

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
- Solving recurrence: $T(n) = O(n)$

Randomized Selection Algorithm

selection(A, n, i)

if $n = 1$ **thenreturn** A
$x \leftarrow$ random element of A (called pivot)
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L.\text{size}$ **then**
    return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R.\text{size}$ **then**
    return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
    return x

Randomized Selection Algorithm

selection(A, n, i)

if $n = 1$ **thenreturn** A
$x \leftarrow$ random element of A (called pivot)
$A_L \leftarrow$ elements in A that are less than x ▷ Divide
$A_R \leftarrow$ elements in A that are greater than x ▷ Divide
if $i \leq A_L$.size **then**
    return selection$(A_L, A_L.\text{size}, i)$ ▷ Conquer
else if $i > n - A_R$.size **then**
    return selection$(A_R, A_R.\text{size}, i - (n - A_R.\text{size}))$ ▷ Conquer
else
    return x

- expected running time $= O(n)$

Outline

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$
\begin{aligned}
&(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5) \\
= {} & 6x^6 - 9x^5 + 18x^4 - 15x^3 \\
& + 4x^5 - 6x^4 + 12x^3 - 10x^2 \\
& - 10x^4 + 15x^3 - 30x^2 + 25x \\
& + 8x^3 - 12x^2 + 24x - 20 \\
= {} & 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20
\end{aligned}
$$

Polynomial Multiplication

Input: two polynomials of degree $n-1$

Output: product of two polynomials

Example:

$$
\begin{aligned}
&(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5) \\
=\ & 6x^6 - 9x^5 + 18x^4 - 15x^3 \\
& + 4x^5 - 6x^4 + 12x^3 - 10x^2 \\
& - 10x^4 + 15x^3 - 30x^2 + 25x \\
& + 8x^3 - 12x^2 + 24x - 20 \\
=\ & 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20
\end{aligned}
$$

- **Input**: $(4, -5, 2, 3), (-5, 6, -3, 2)$
- **Output**: $(-20, 49, -52, 20, 2, -5, 6)$

Naïve Algorithm

polynomial-multiplication(A, B, n)

let $C[k] \leftarrow 0$ for every $k = 0, 1, 2, \cdots, 2n-2$
for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
        $C[i+j] \leftarrow C[i+j] + A[i] \times B[j]$
return C

Naïve Algorithm

polynomial-multiplication(A, B, n)

let $C[k] \leftarrow 0$ for every $k = 0, 1, 2, \cdots, 2n-2$
for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
        $C[i+j] \leftarrow C[i+j] + A[i] \times B[j]$
return C

Running time: $O(n^2)$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

- $p(x)$: degree of $n-1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x+2)x^2 + (-5x+4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x-3)x^2 + (6x-5)$$

- $p(x)$: degree of $n-1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

$$pq = \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right)$$

Divide-and-Conquer for Polynomial Multiplication

$$p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)$$
$$q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)$$

- $p(x)$: degree of $n - 1$ (assume n is even)
- $p(x) = p_H(x)x^{n/2} + p_L(x)$,
- $p_H(x), p_L(x)$: polynomials of degree $n/2 - 1$.

$$\begin{aligned} pq &= \left(p_H x^{n/2} + p_L\right)\left(q_H x^{n/2} + q_L\right) \\ &= p_H q_H x^n + \left(p_H q_L + p_L q_H\right)x^{n/2} + p_L q_L \end{aligned}$$