CS 350 Algorithms and Complexity

Winter 2019

Lecture 9: Divide & Conquer
The Master Theorem, Mergesort & Quicksort

Andrew P. Black

Department of Computer Science
Portland State University

What is Divide-and-Conquer?

What is Divide-and-Conquer?

What is Divide-and-Conquer?

Solves a problem instance of size n by:

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

Time taken:

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

Time taken:

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

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

Time taken:

$$T(n) = aT\left(\frac{n}{b}\right)$$

A. True or B. False ?

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

Time taken:

$$T(n) = aT\left(\frac{n}{b}\right) + T_{split+combine}(n)$$

A. True or B. False ?

What is Divide-and-Conquer?

Solves a problem instance of size n by:

1. splitting it into b smaller instances, of size ~ n/b
2. solving some or all of them (in general, solving a of them), using the same algorithm recursively.
3. combining the solutions to the a smaller problems to get the solution to the original problem.

Time taken:

$$T(n) = aT\left(\frac{n}{b}\right) + T_{split+combine}(n)$$

A. True or B. False ?

MergeSort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Merge

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

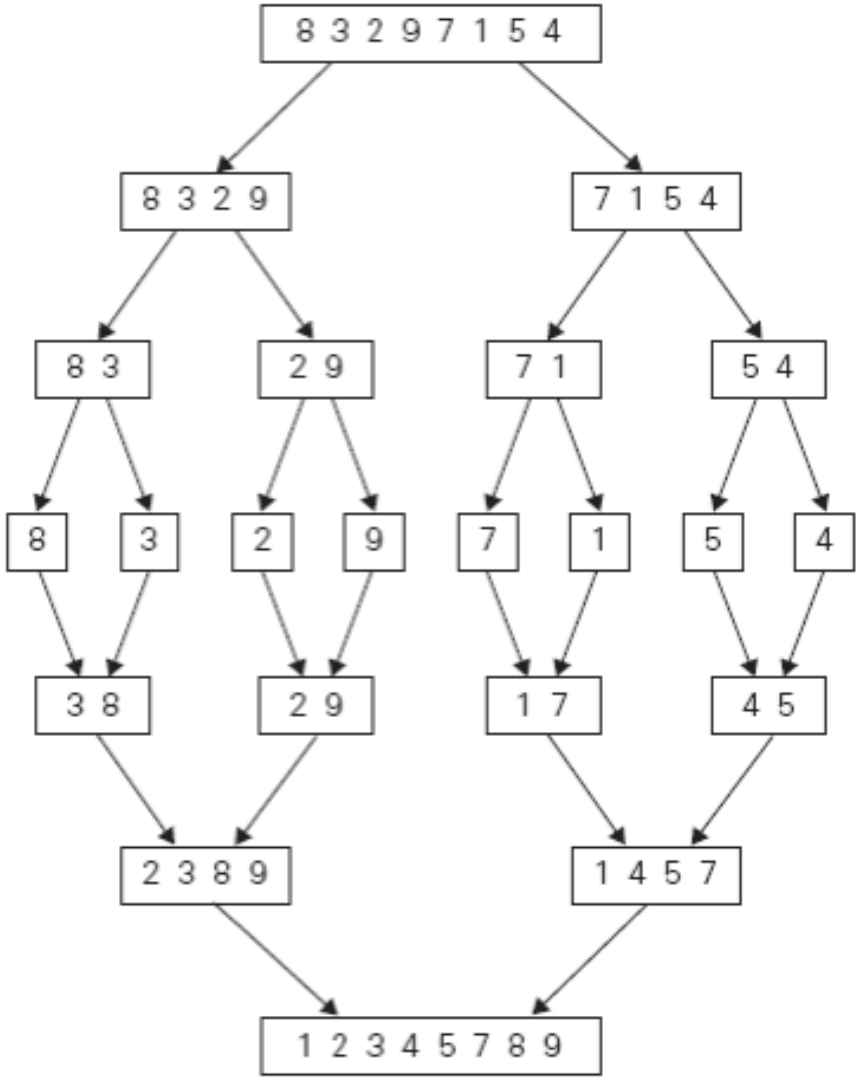


FIGURE 5.2 Example of mergesort operation.

Time Complexity of Mergesort

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing orc
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:
  - $C(1) = 0$

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing orc
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:
  - $C(1) = 0$
  - $C(n) = 2\ C(n/2) + C_{split}(n) + C_{merge}(n)$

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing orc
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:
  - $C(1) = 0$
  - $C(n) = 2\ C(n/2) + C_{split}(n) + C_{merge}(n)$
  - $C_{merge}(n) = n - 1$ in the worst case ($n/2$ in the best)

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing orc
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Time Complexity of Mergesort

- Suppose that we are sorting an array of $n=2^k$ elements
- Let $C(n)$ = number of comparisons when sorting n elements, then:
  - $C(1) = 0$
  - $C(n) = 2\ C(n/2) + C_{split}(n) + C_{merge}(n)$
  - $C_{merge}(n) = n - 1$ in the worst case ($n/2$ in the best)
  - $C_{worst}(n) = 2\ C_{worst}(n/2) + n - 1$

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing orc
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Solve the recurrence:

Solve the recurrence:

$C(n) = 2\,C(n/2) + n - 1$ (omitting *worst*)

Solve the recurrence:

$$C(n) = 2\,C(n/2) + n - 1 \qquad \text{(omitting } \textit{worst}\text{)}$$

$$C(n)/n = 2\,C(n/2)/n + 1 - 1/n \qquad \text{(divide by } n\text{)}$$

$$= C(n/2)/(n/2) + 1 - 1/n \qquad \text{(algebra)}$$

Solve the recurrence:

$$C(n) = 2\,C(n/2) + n - 1 \qquad \text{(omitting } \textit{worst}\text{)}$$

$$C(n)/n = 2\,C(n/2)/n + 1 - 1/n \qquad \text{(divide by } n\text{)}$$

$$= C(n/2)/(n/2) + 1 - 1/n \qquad \text{(algebra)}$$

$$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n \qquad \text{(subst. } n \Rightarrow n/2\text{)}$$

Solve the recurrence:

$$C(n) = 2\,C(n/2) + n - 1 \qquad \text{(omitting } \textit{worst}\text{)}$$

$$C(n)/n = 2\,C(n/2)/n + 1 - 1/n \qquad \text{(divide by } n\text{)}$$

$$= C(n/2)/(n/2) + 1 - 1/n \qquad \text{(algebra)}$$

$$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n \qquad \text{(subst. } n \Rightarrow n/2\text{)}$$

$$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$$

Solve the recurrence:

$C(n) = 2\,C(n/2) + n - 1$ (omitting *worst*)

$C(n)/n = 2\,C(n/2)/n + 1 - 1/n$ (divide by n)

$= C(n/2)/(n/2) + 1 - 1/n$ (algebra)

$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n$ (subst. $n \Rightarrow n/2$)

$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$

$\ldots$

Solve the recurrence:

$C(n) = 2\,C(n/2) + n - 1$ (omitting *worst*)

$C(n)/n = 2\,C(n/2)/n + 1 - 1/n$ (divide by n)

$= C(n/2)/(n/2) + 1 - 1/n$ (algebra)

$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n$ (subst. $n \Rightarrow n/2$)

$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$

$\ldots$

$= C(n/n)/(n/n) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$

Solve the recurrence:

$C(n) = 2\,C(n/2) + n - 1$ (omitting *worst*)

$C(n)/n = 2\,C(n/2)/n + 1 - 1/n$ (divide by n)

$= C(n/2)/(n/2) + 1 - 1/n$ (algebra)

$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n$ (subst. $n \Rightarrow n/2$)

$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$

$\ldots$

$= C(n/n)/(n/n) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$

$C(n)/n = C(1) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$

Solve the recurrence:

$$C(n) = 2\,C(n/2) + n - 1 \qquad \text{(omitting } \textit{worst}\text{)}$$

$$C(n)/n = 2\,C(n/2)/n + 1 - 1/n \qquad \text{(divide by } n\text{)}$$

$$= C(n/2)/(n/2) + 1 - 1/n \qquad \text{(algebra)}$$

$$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n \qquad \text{(subst. } n \Rightarrow n/2\text{)}$$

$$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$$

$$\ldots$$

$$= C(n/n)/(n/n) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$$

$$C(n)/n = C(1) \underbrace{+1+1+1+\ldots}_{\lg n \text{ terms}} \underbrace{- 1/n - 2/n - 4/n - \ldots}_{\lg n \text{ terms}}$$

$$\cong \lg n \qquad \text{(neglecting small terms)}$$

Solve the recurrence:

$$C(n) = 2\,C(n/2) + n - 1 \qquad \text{(omitting } \textit{worst}\text{)}$$

$$C(n)/n = 2\,C(n/2)/n + 1 - 1/n \qquad \text{(divide by } n\text{)}$$

$$= C(n/2)/(n/2) + 1 - 1/n \qquad \text{(algebra)}$$

$$= C(n/4)/(n/4) + 1 + 1 - 1/n - 2/n \qquad \text{(subst. } n \Rightarrow n/2\text{)}$$

$$= C(n/8)/(n/8) + 1 + 1 + 1 - 1/n - 2/n - 4/n$$

$$\ldots$$

$$= C(n/n)/(n/n) + 1 + 1 + 1 + \ldots - 1/n - 2/n - 4/n - \ldots$$

$$C(n)/n = C(1) \underbrace{+1+1+1+\ldots}_{\lg n \text{ terms}} \underbrace{- 1/n - 2/n - 4/n - \ldots}_{\lg n \text{ terms}}$$

$$\cong \lg n \qquad \text{(neglecting small terms)}$$

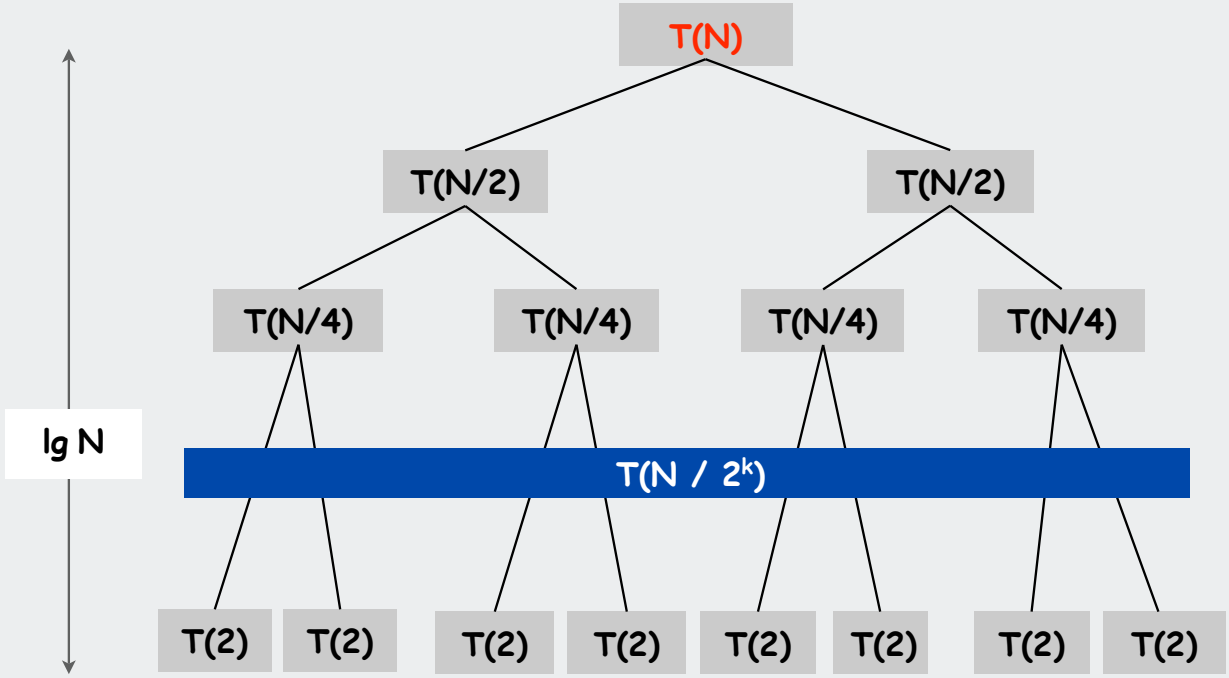
$$C(n) = n \lg n$$

Look at the recursion tree:

$T(N) = 2\,T(N/2) + N$

for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)



$T(N) = N \lg N$

10

from Sedgewick & Wayne http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/04MergeQuick.pdf

Divide & Conquer

- When designing a divide and conquer algorithm, the time for the "split" and "combine" phases contributes to the cost:

  A. not at all
  B. in a small way
  C. in a major way
  D. in a way that dominates the total cost
  E. in a way that depends on the algorithm

Divide & Conquer

- In mergesort, the total cost is dominated by the cost of:

  A. the split phase (sub-array copy)
  B. the combine phase (merge)
  C. the recursion
  D. something else

Divide & Conquer

- In Quicksort, the total cost is dominated by the cost of:

  A. the split phase (partition)
  B. the combine phase (sub-array composition)
  C. the recursion
  D. something else

The Master Theorem:

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

Application to Mergesort:

- We divide the problem into 2 (roughly equal) parts, so $b = 2$
- We have to solve *both* of them, so $a = 2$
- Combining the parts is linear, i.e., in $\Theta(n^1)$, so $d = 1$
- Hence, in the Master Theorem, $a = 2 = 2^1 = b^d$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

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

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

Does the base of this log matter? A: yes B: no

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

Does the base of this log matter? A: yes B: no

The Master Theorem:

$$T(n) = aT(n/b) + T_{combine}(n)$$

- Suppose that $T_{combine}(n) \in \Theta(n^d)$ Then:

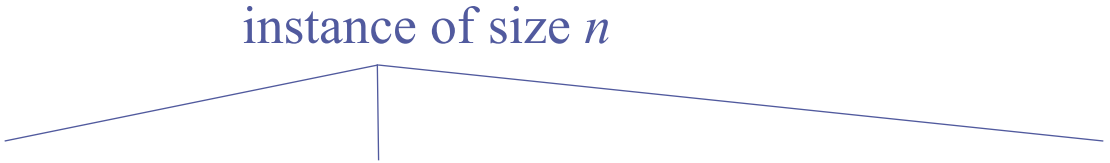
$$T(n) \in \begin{cases} \Theta(n^d) & \text{when } a < b^d \\ \Theta(n^d \log n) & \text{when } a = b^d \\ \Theta(n^{\log_b a}) & \text{when } a > b^d \end{cases}$$

Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:

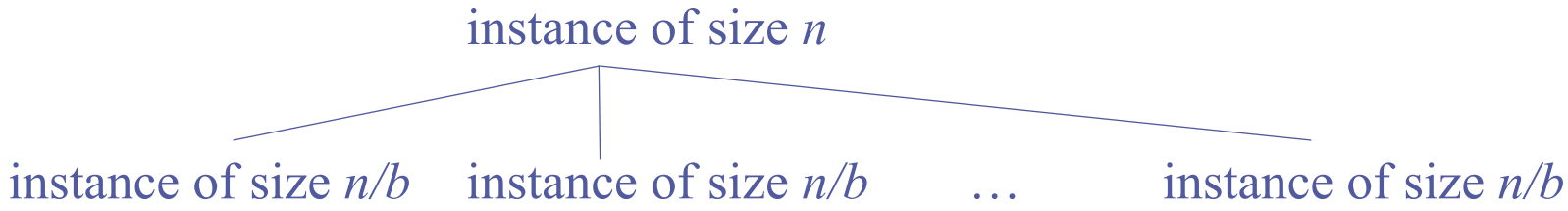
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



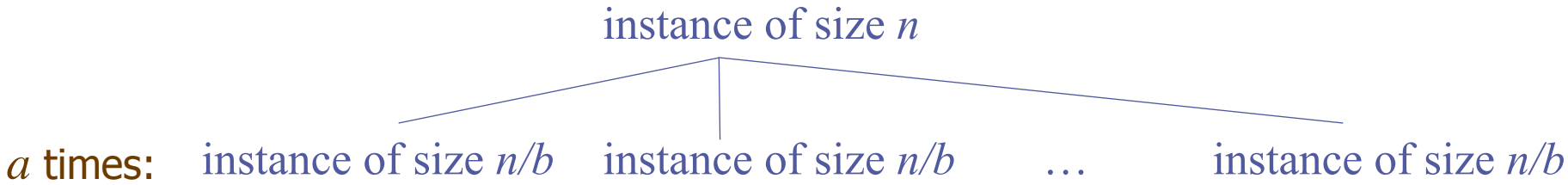
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



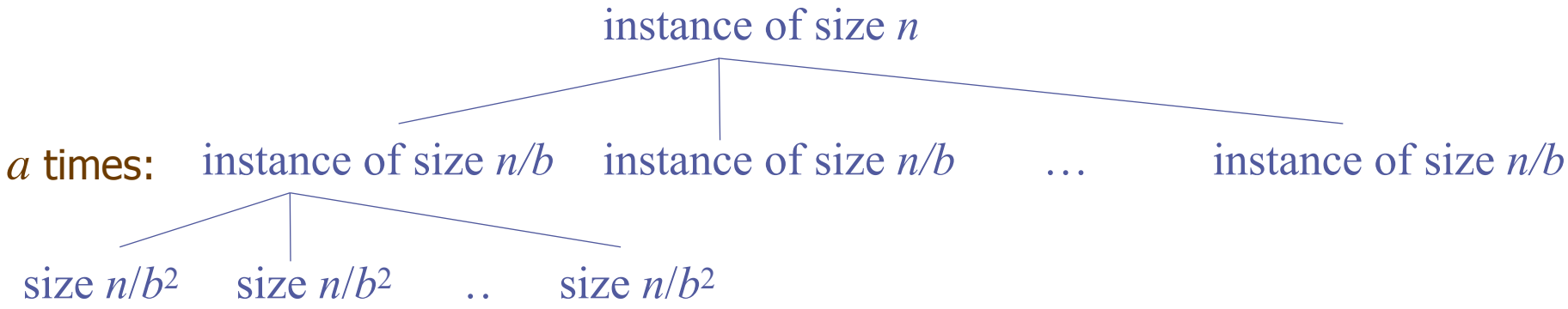
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



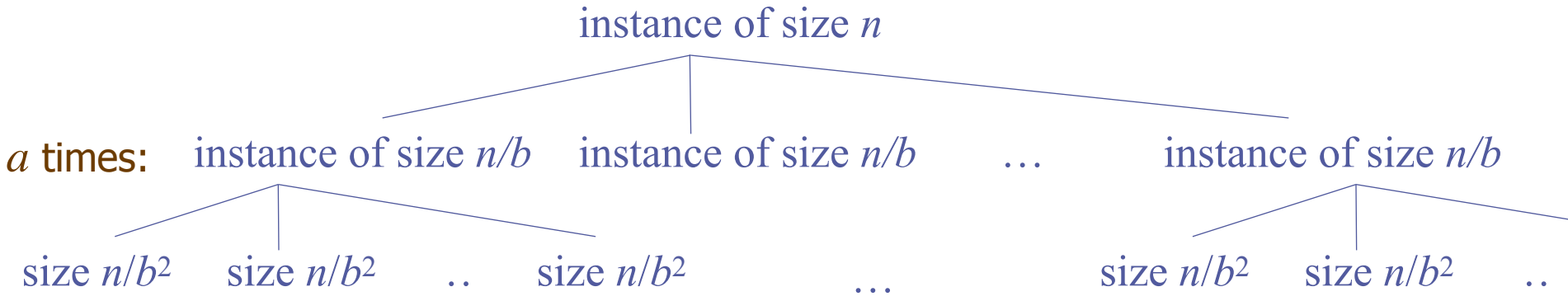
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



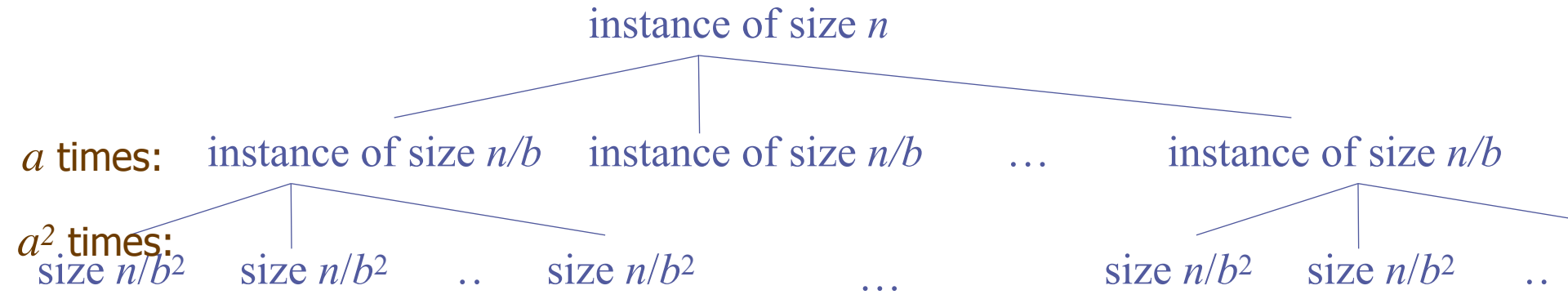
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



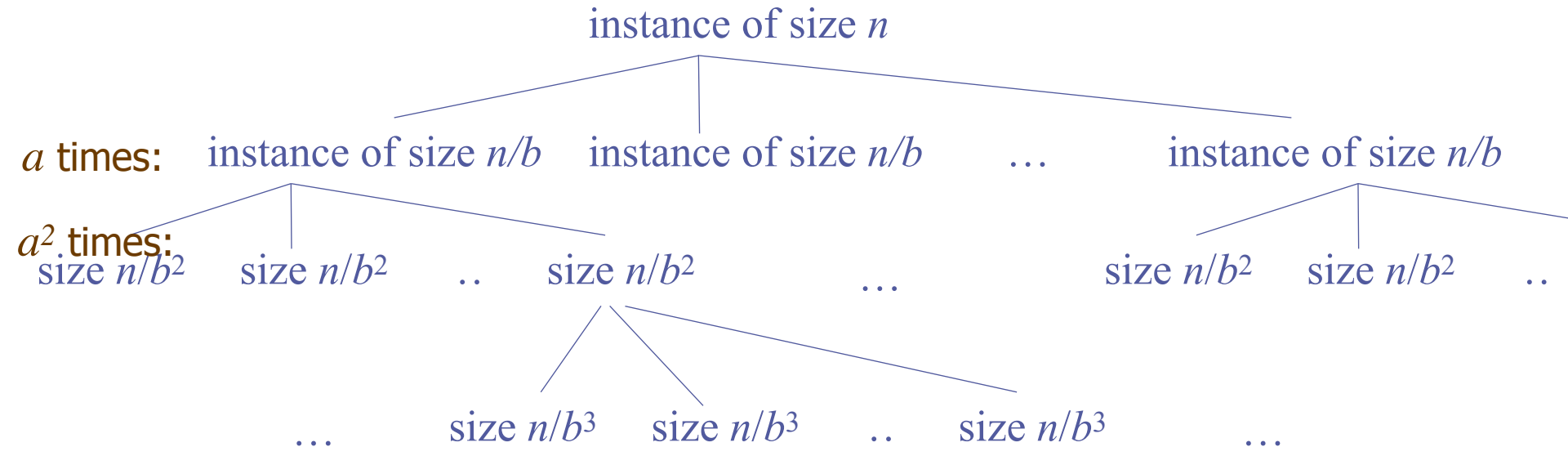
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



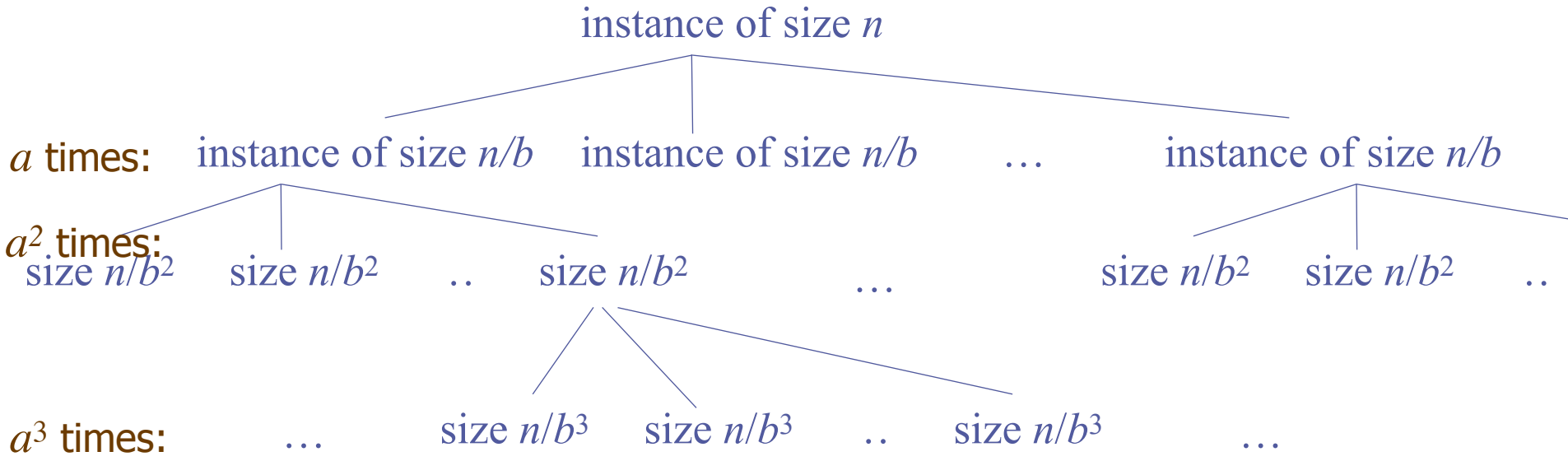
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



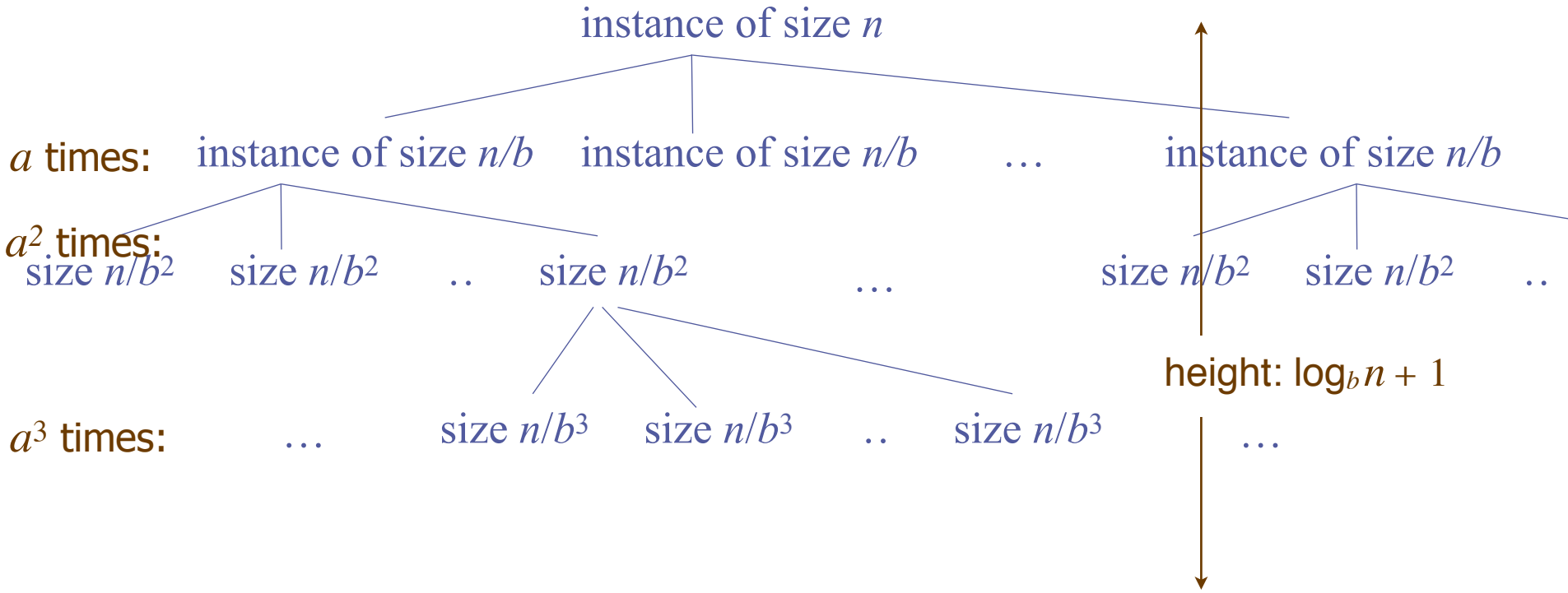
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



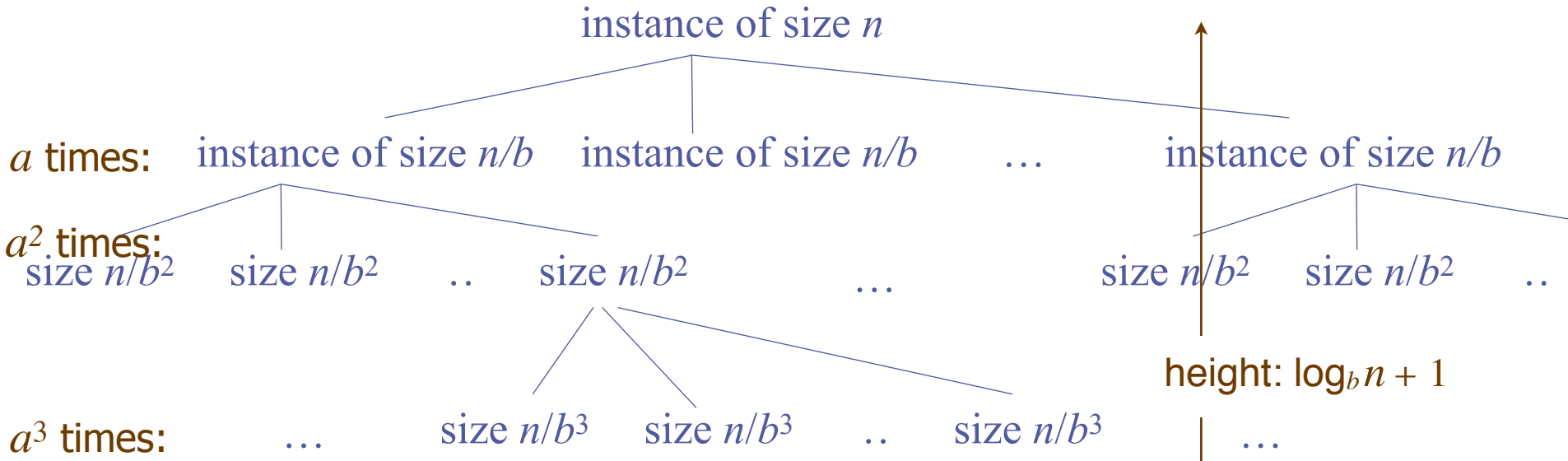
Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



Problem:

- Why are there three cases in the Master Theorem?
  - Look at the recursion tree:



At level i: a^i problems of size $\frac{n}{b^i}$, *each* requiring combining work in $\Theta\left((\frac{n}{b^i})^d\right)$

So total work is at level i is: $\Theta\left(a^i\left(\frac{n}{b^i}\right)^d\right) = \Theta\left(n^d\left(\frac{a}{b^d}\right)^i\right)$

Why are there 3 cases?

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

  - It depends on whether $\frac{a}{b^d}$ is less than, equal to, or greater than 1

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

  - It depends on whether $\frac{a}{b^d}$ is less than, equal to, or greater than 1

    $a < b^d$

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

  - It depends on whether $\frac{a}{b^d}$ is less than, equal to, or greater than 1

$a < b^d$

A. increases

B. decreases

C. constant

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

  - It depends on whether $\frac{a}{b^d}$ is less than, equal to, or greater than 1

$a < b^d$

$a = b^d$

A. increases

B. decreases

C. constant

Why are there 3 cases?

- Combining work at level i is in $\Theta\left(n^d \left(\frac{a}{b^d}\right)^i\right)$

  Does this term increase with i, stay constant, or decrease?

  - It depends on whether $\frac{a}{b^d}$ is less than, equal to, or greater than 1

$a < b^d$	A. increases
$a = b^d$	B. decreases
$a > b^d$	C. constant

Master Theorem

Master Theorem

- Total work at level i is:

$$\Theta\left(n^d\left(\frac{a}{b^d}\right)^i\right)$$

Master Theorem

- Total work at level i is:

$$\Theta\left(n^d\left(\frac{a}{b^d}\right)^i\right)$$

- But there are $\log_b n + 1$ levels

Master Theorem

- Total work at level i is:

$$\Theta\left(n^d\left(\frac{a}{b^d}\right)^i\right)$$

- But there are $\log_b n + 1$ levels
- So, total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d\left(\frac{a}{b^d}\right)^i\right)$$

Master Theorem: Case $a < b^d$

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right)$$

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right) = \Theta\left(n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right) = \Theta\left(n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

- This is n^d times the sum of a geometric series with ratio $r < 1$. (See Summation formula 5, p 476)

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right) = \Theta\left(n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

- This is n^d times the sum of a geometric series with ratio $r < 1$. (See Summation formula 5, p 476)

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} = \frac{1 - a^{n+1}}{1 - a} \quad (a \neq 1)$$

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right) = \Theta\left(n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

- This is n^d times the sum of a geometric series with ratio $r < 1$. (See Summation formula 5, p 476)

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r} \approx \frac{1}{1-r} \quad \text{as } n \to \infty$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} = \frac{1-a^{n+1}}{1-a} \quad (a \neq 1)$$

Master Theorem: Case $a < b^d$

- Total work is

$$\Theta\left(\sum_{i=0}^{\log_b n} n^d \left(\frac{a}{b^d}\right)^i\right) = \Theta\left(n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i\right)$$

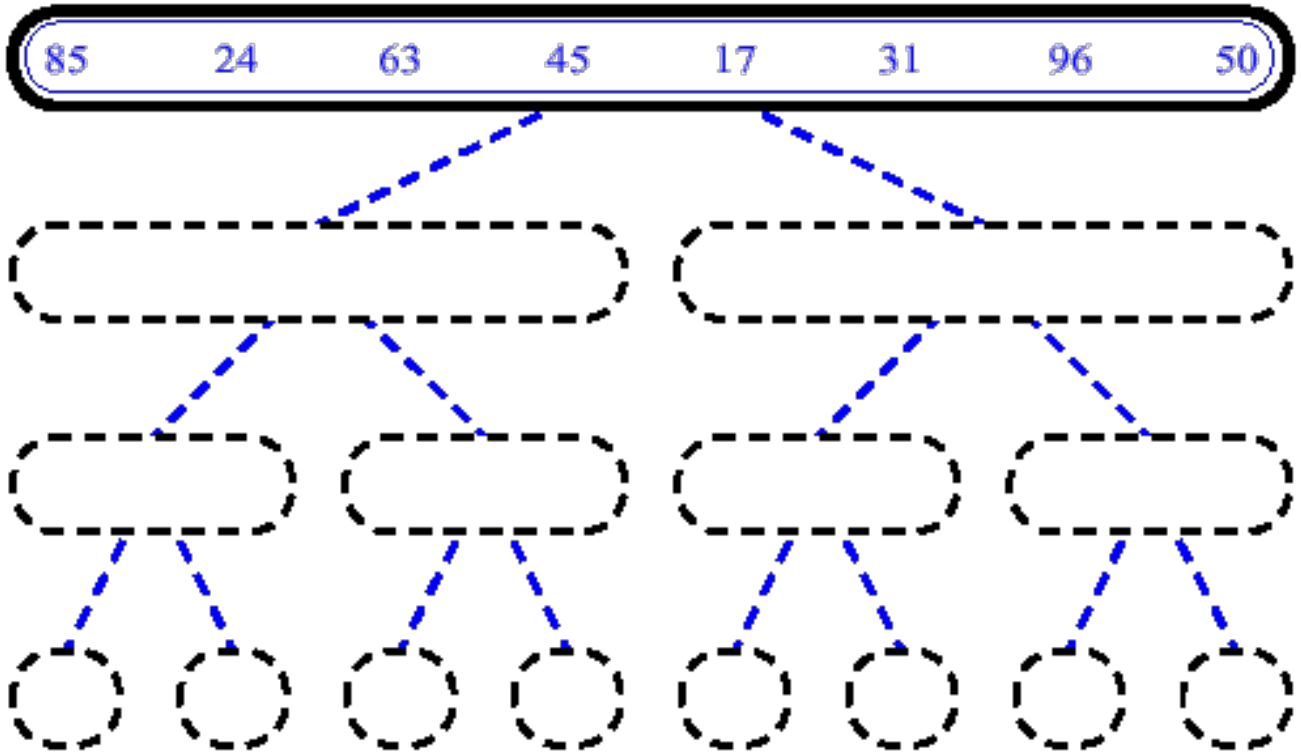
a constant

- This is n^d times the sum of a geometric series with ratio $r < 1$. (See Summation formula 5, p 476)

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r} \approx \frac{1}{1-r} \quad \text{as } n \to \infty$$

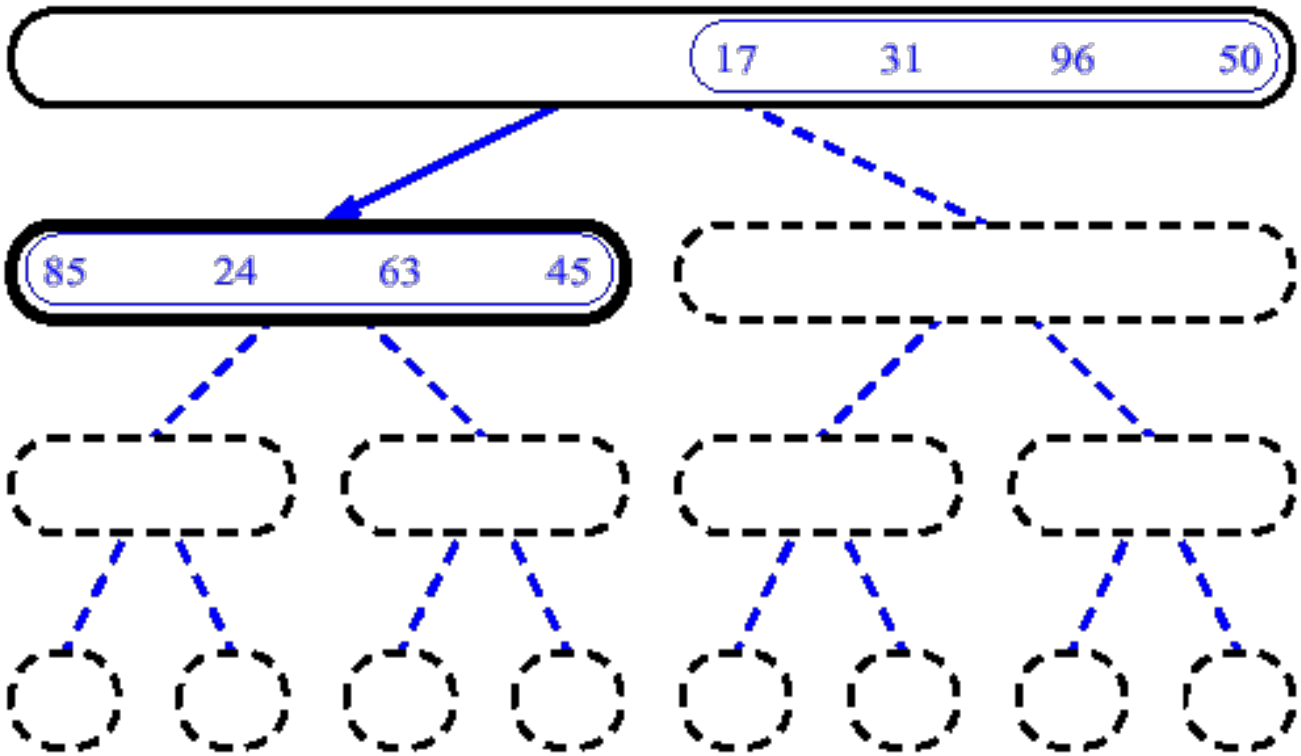
$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} = \frac{1-a^{n+1}}{1-a} \quad (a \neq 1)$$

MergeSort (Example) — 1

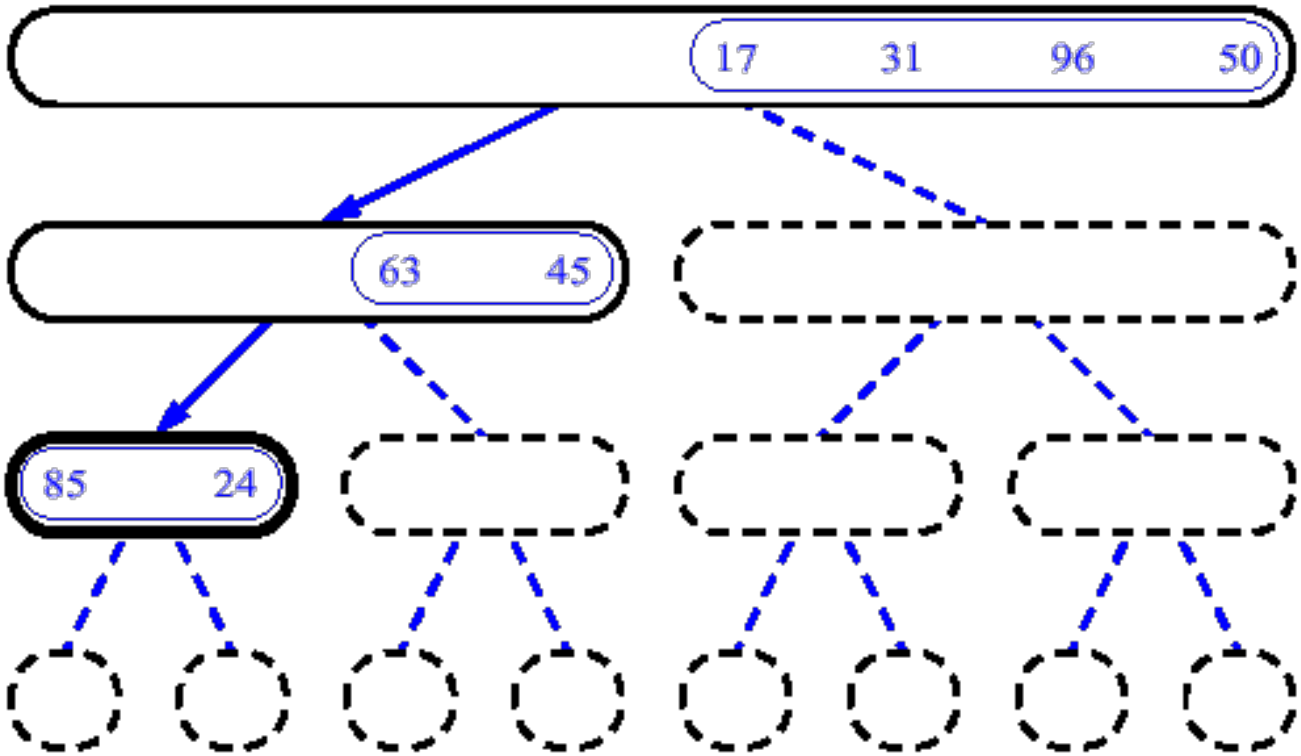


Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 2

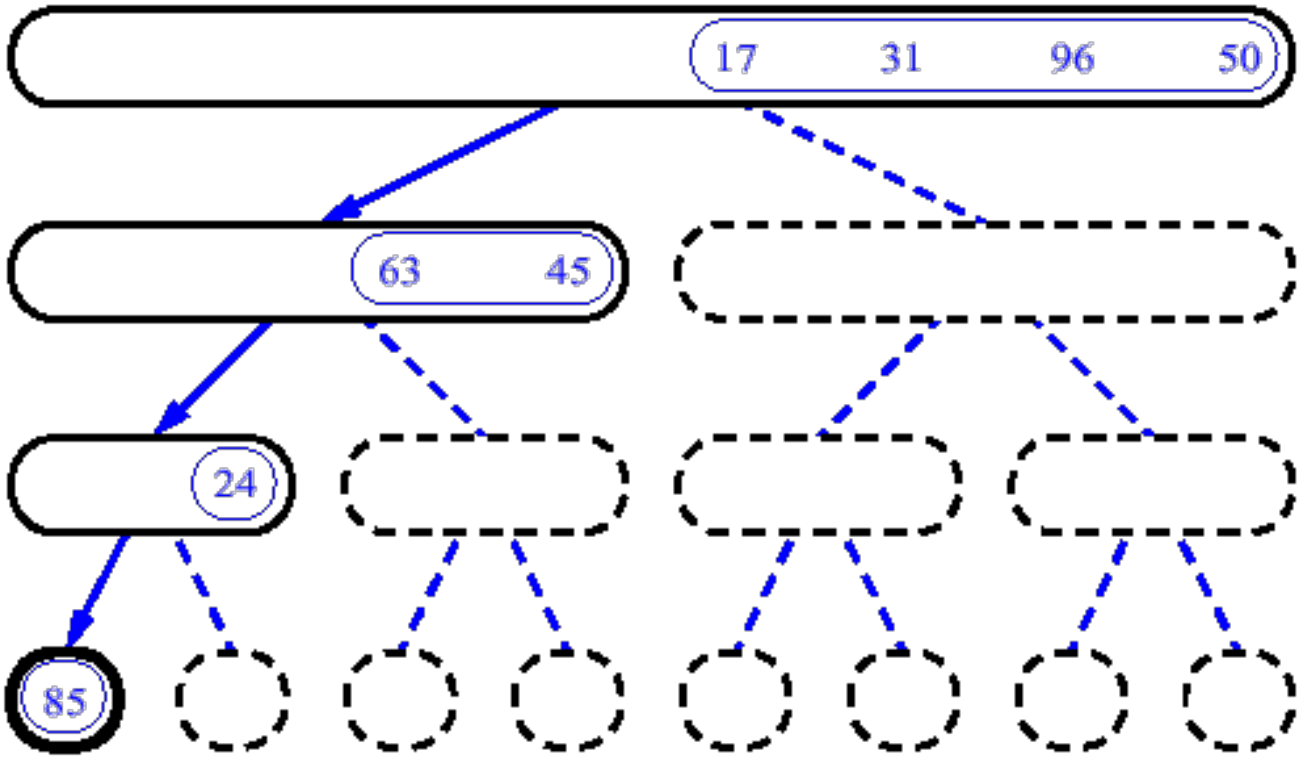


MergeSort (Example) — 3



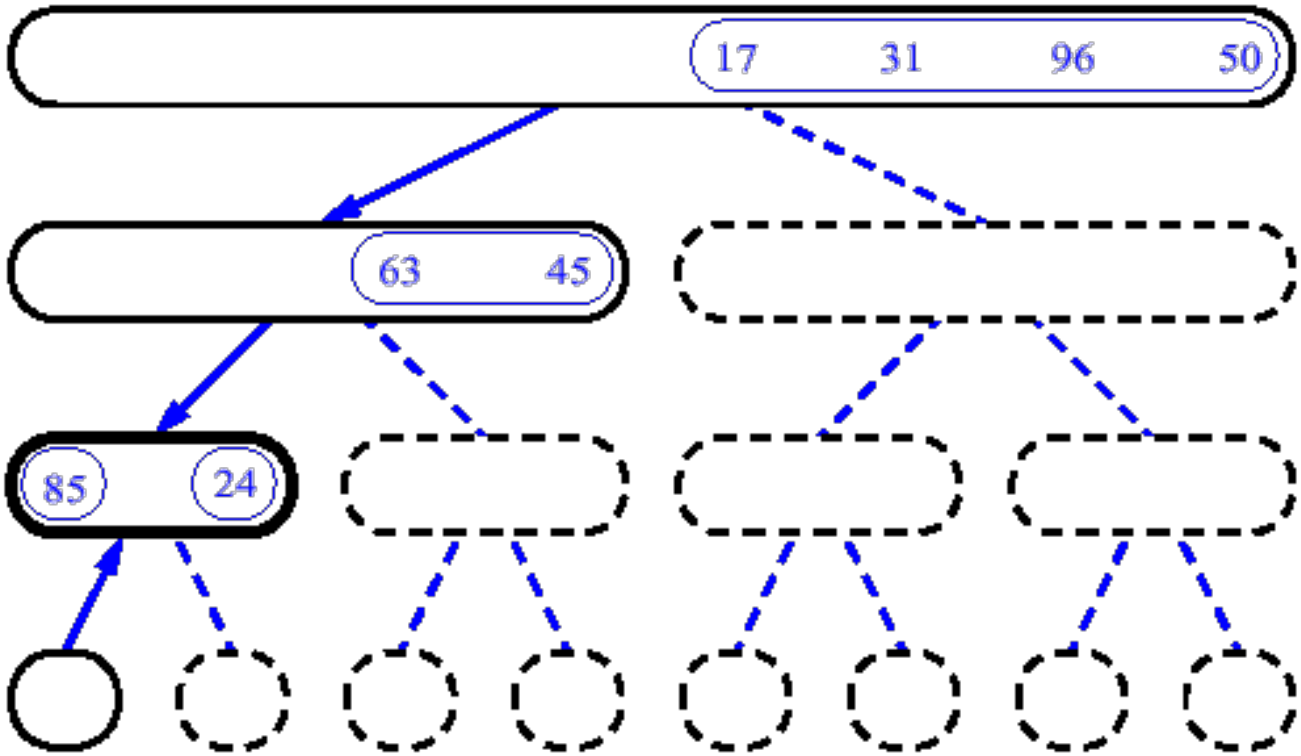
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 4



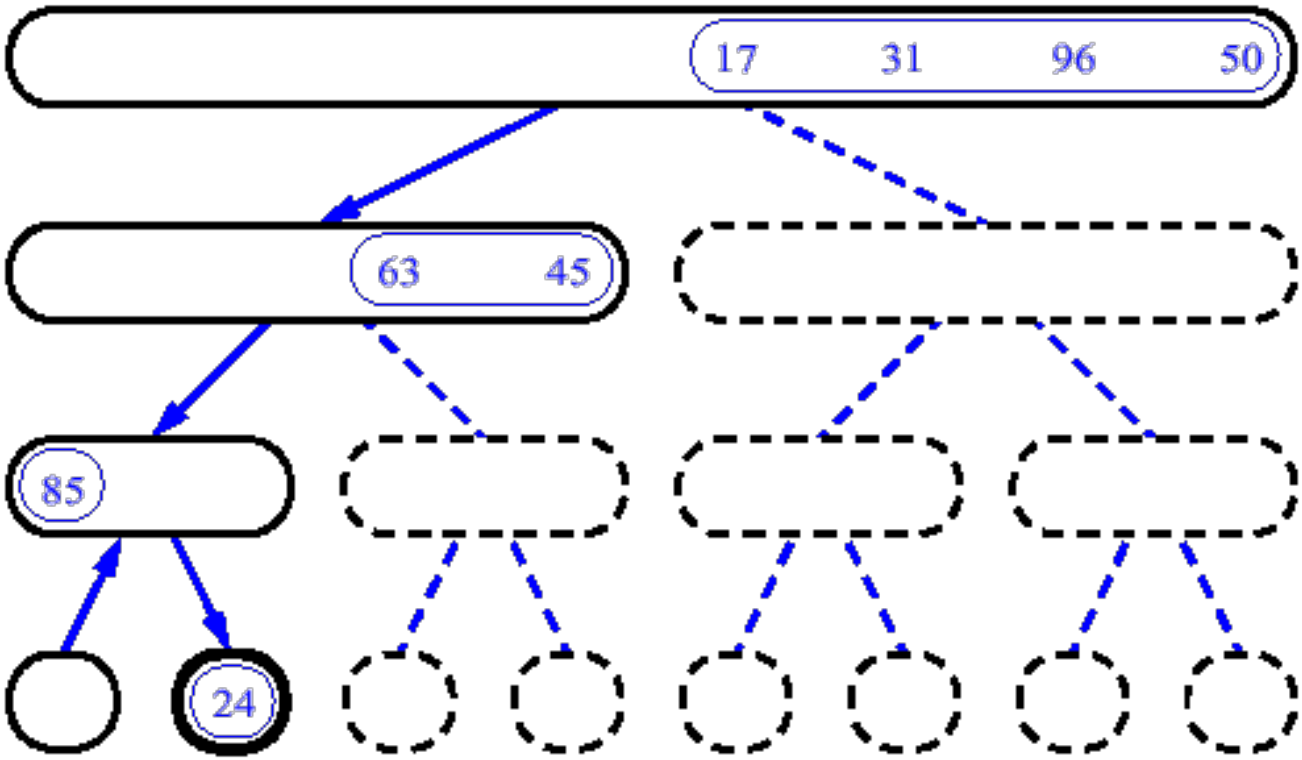
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 5



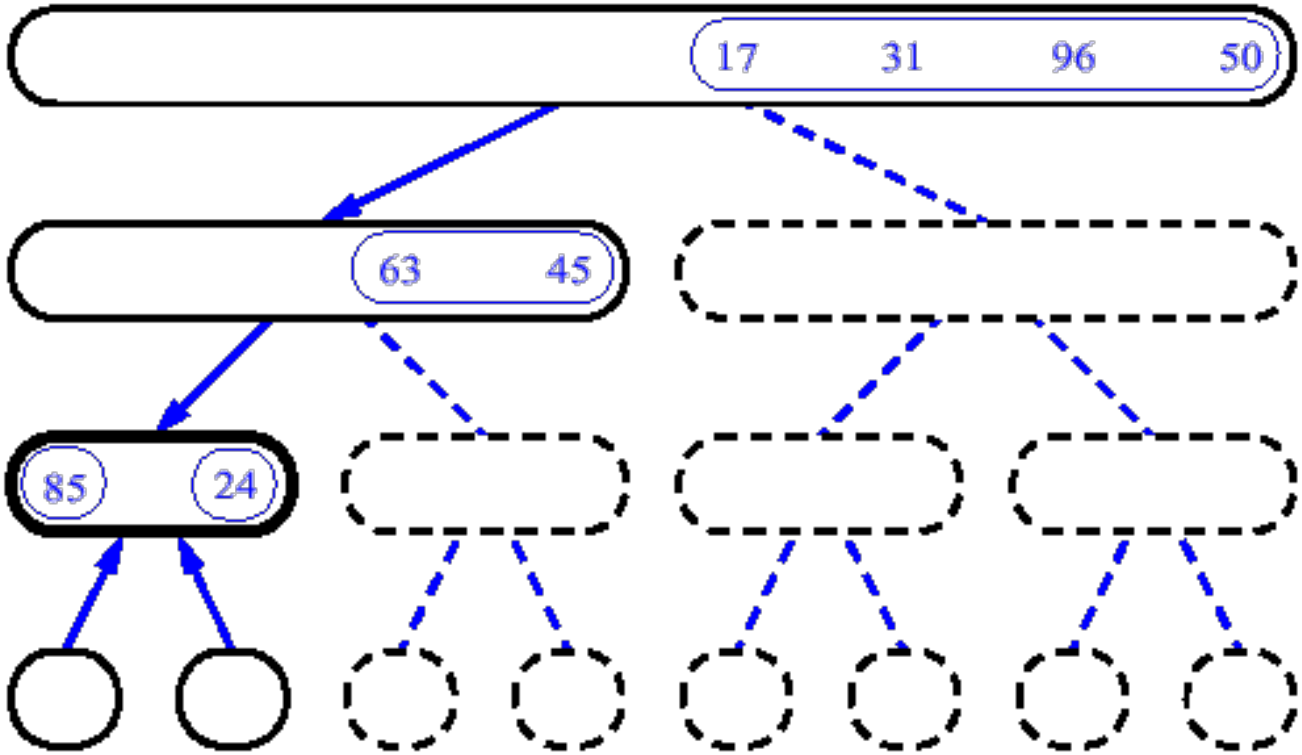
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 6



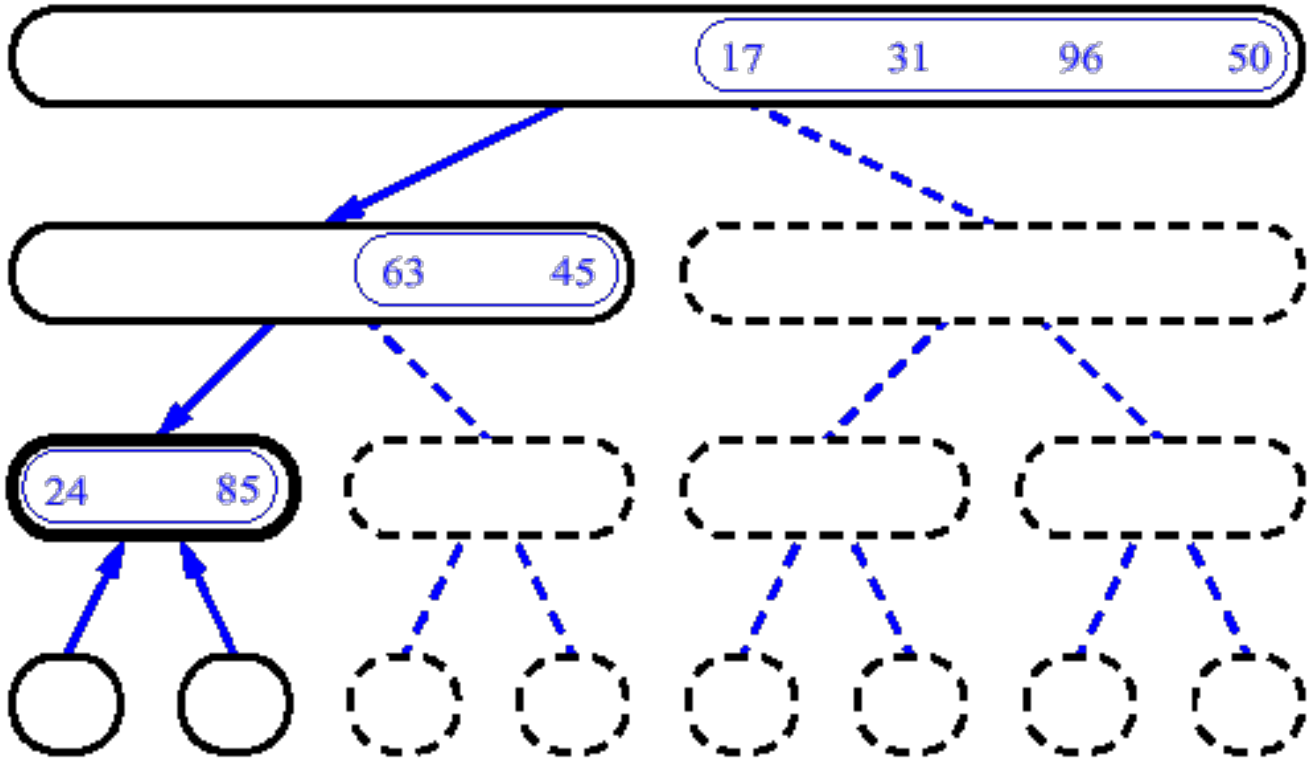
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 7



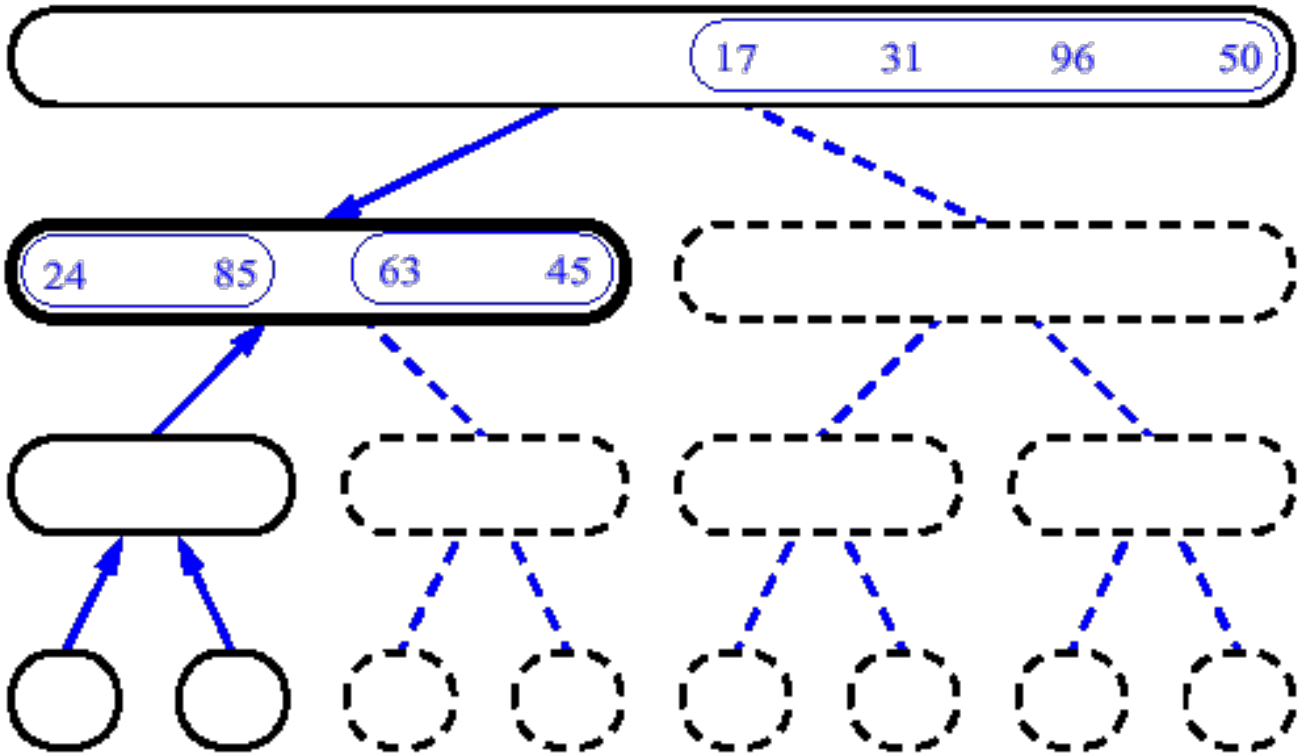
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 8



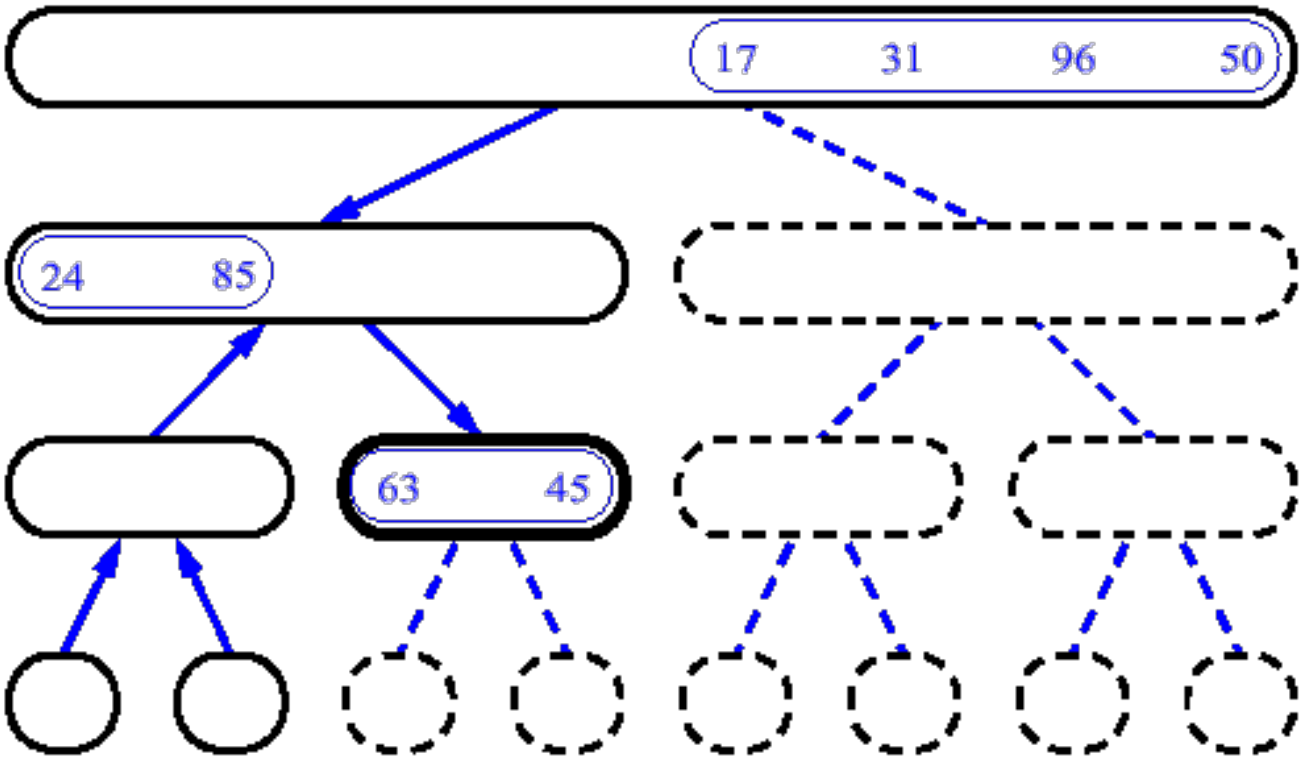
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 9



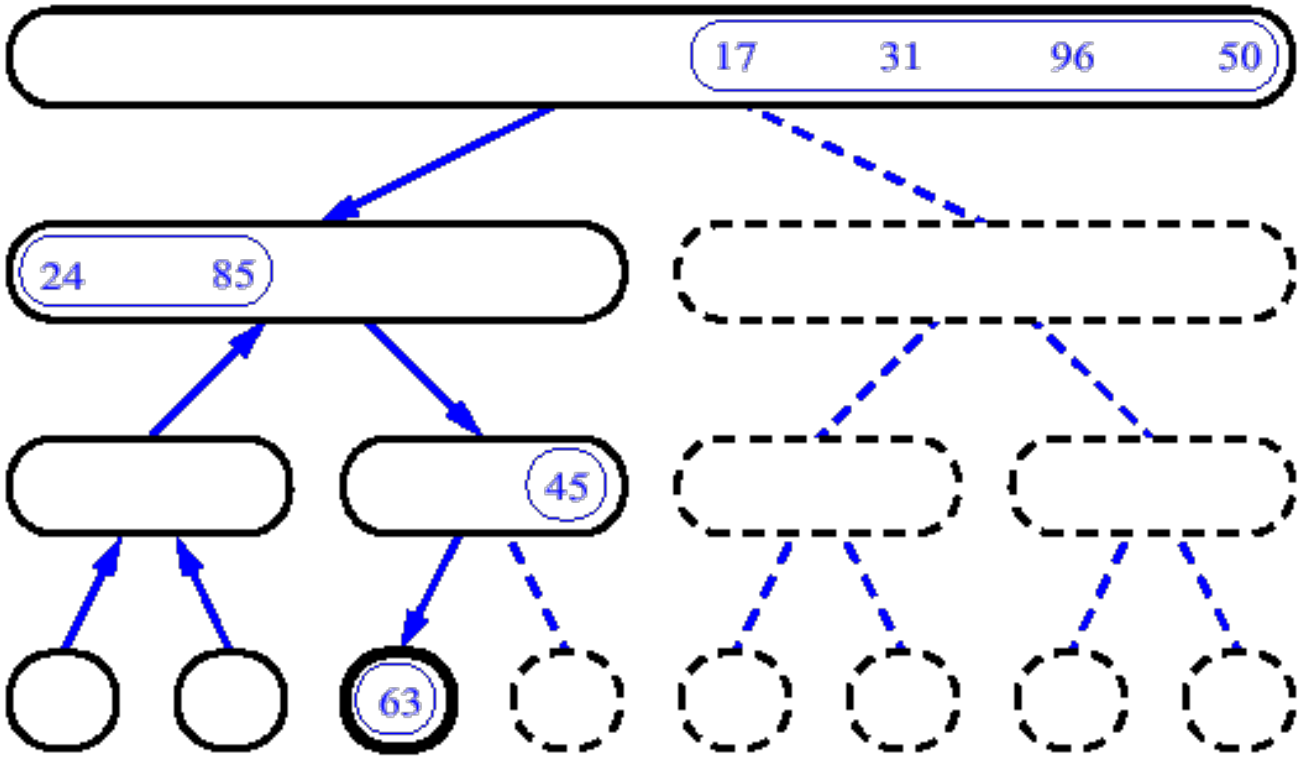
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 10



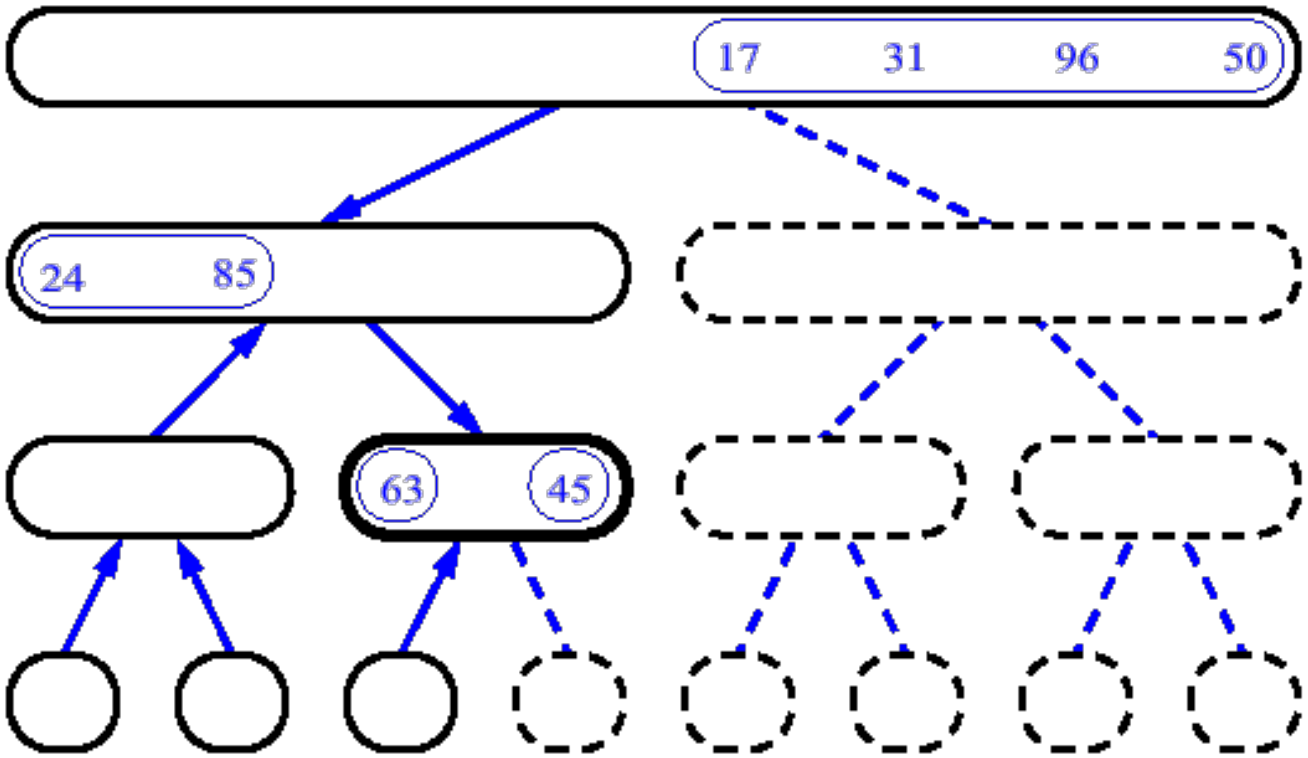
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 11



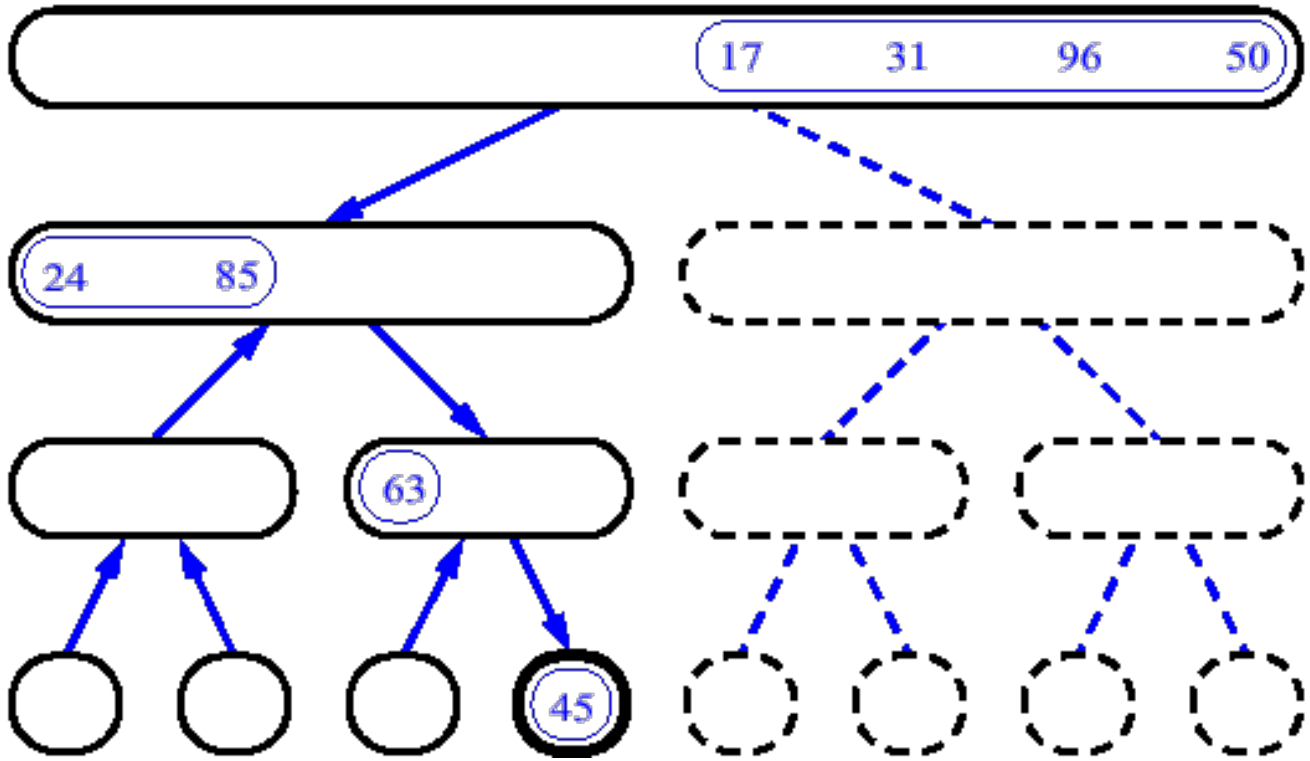
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 12



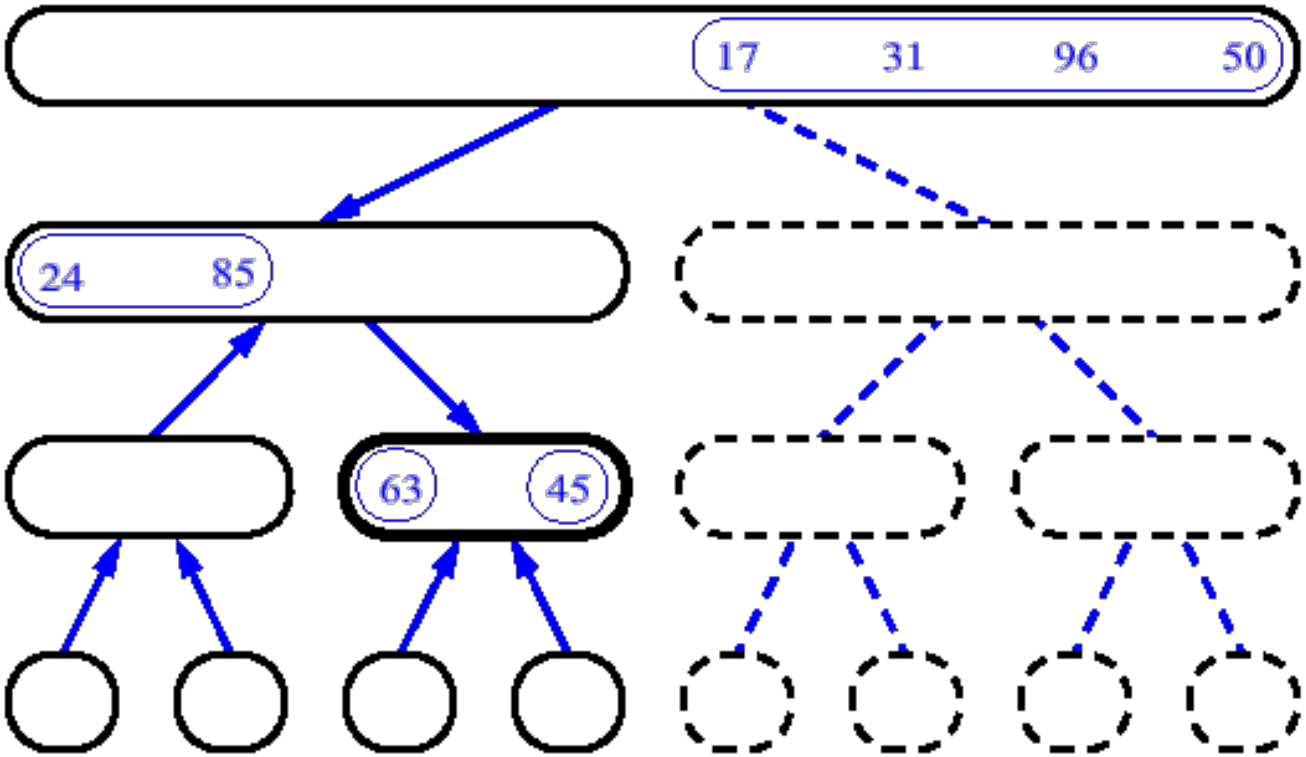
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 13



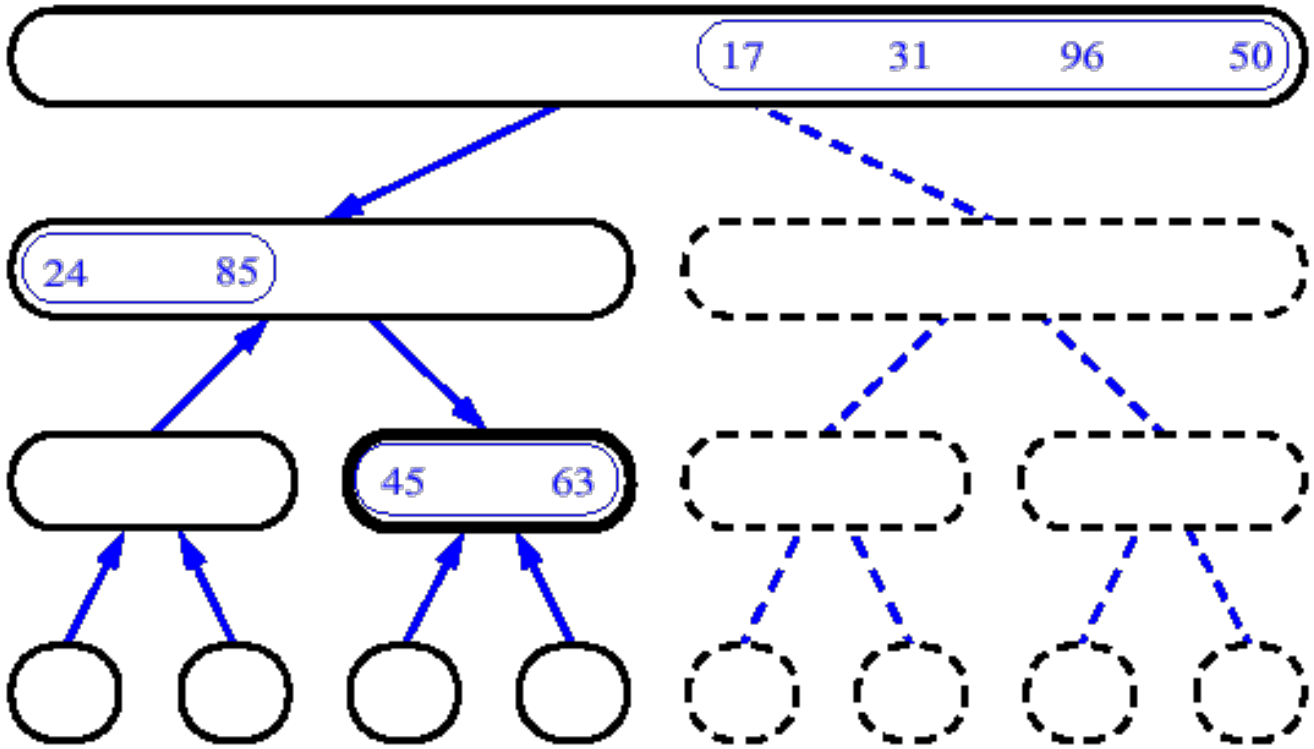
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 14



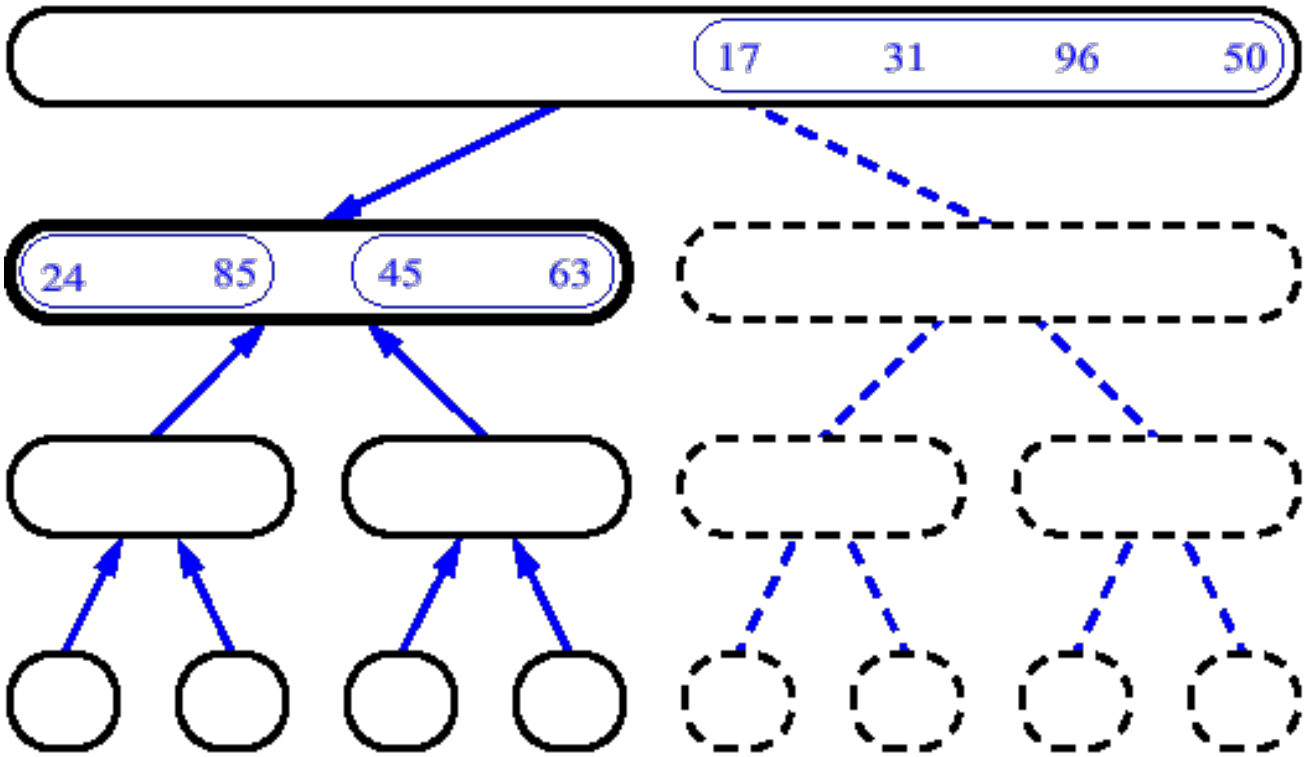
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 15



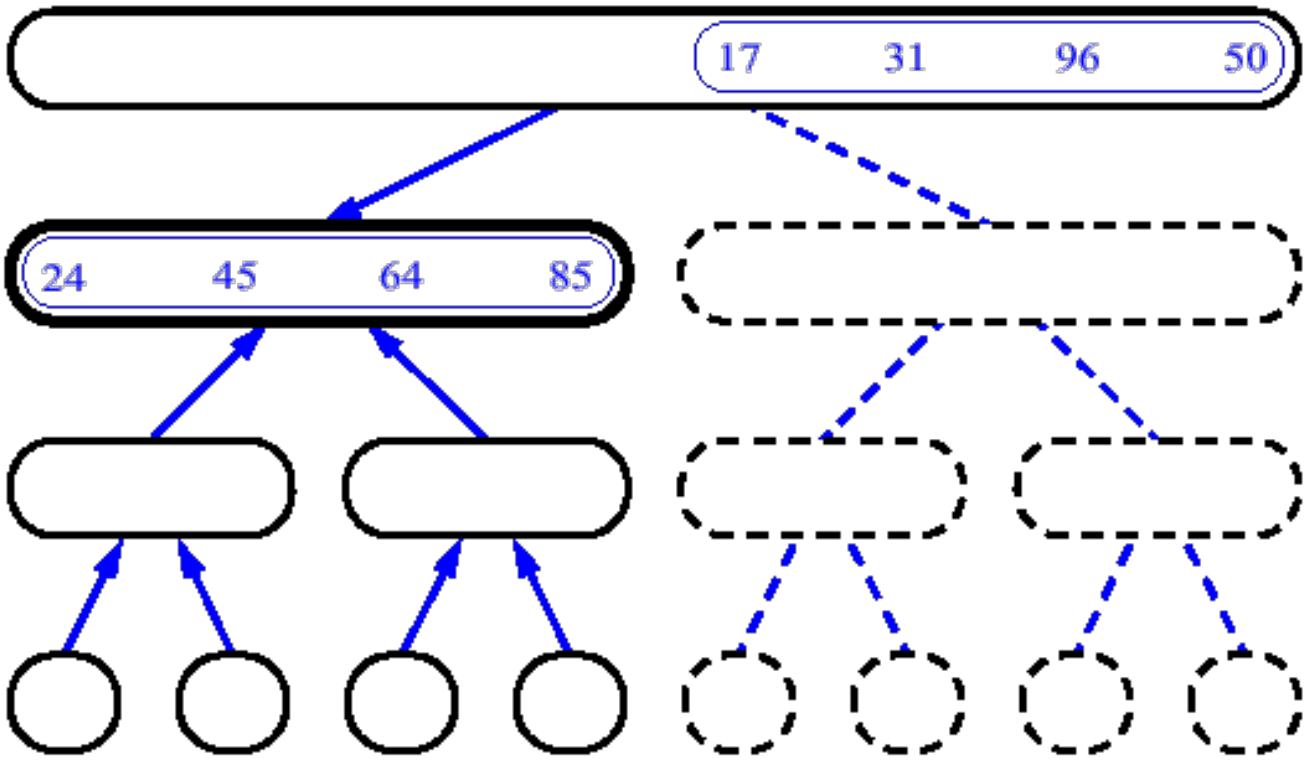
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 16



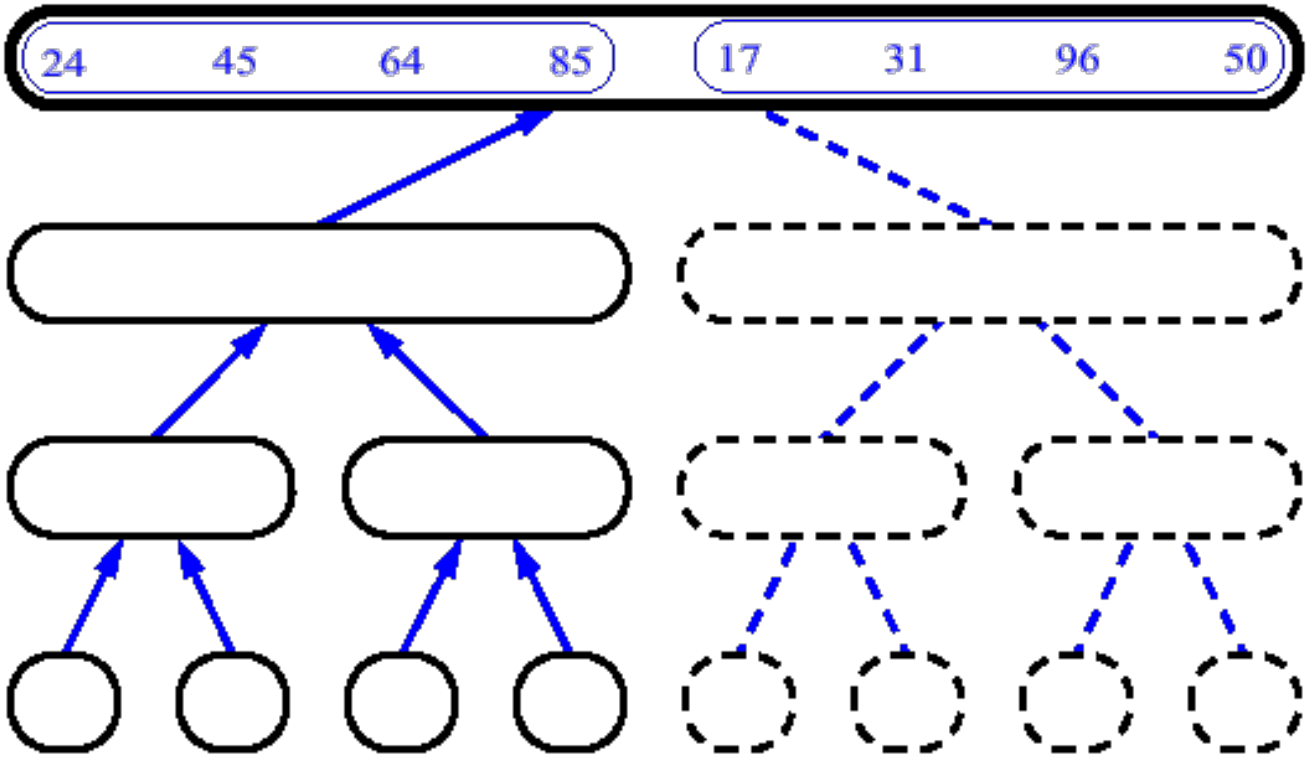
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 17



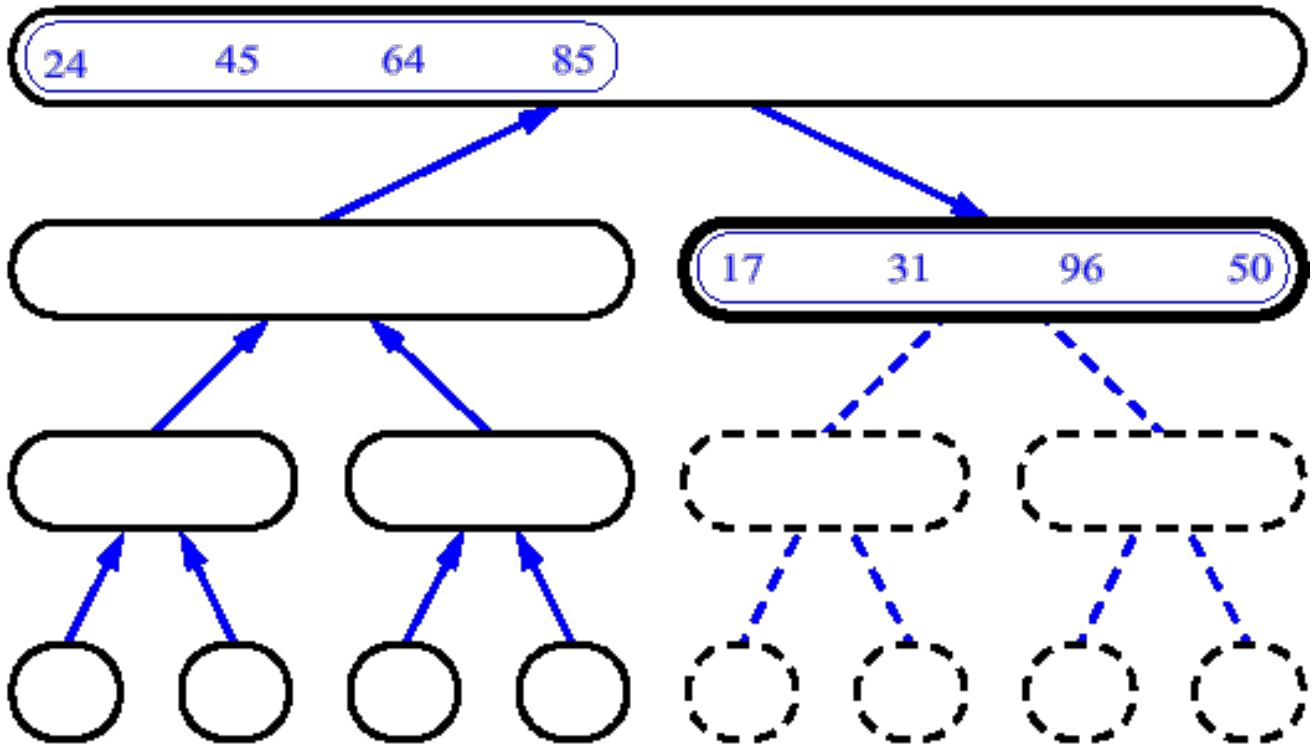
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 18



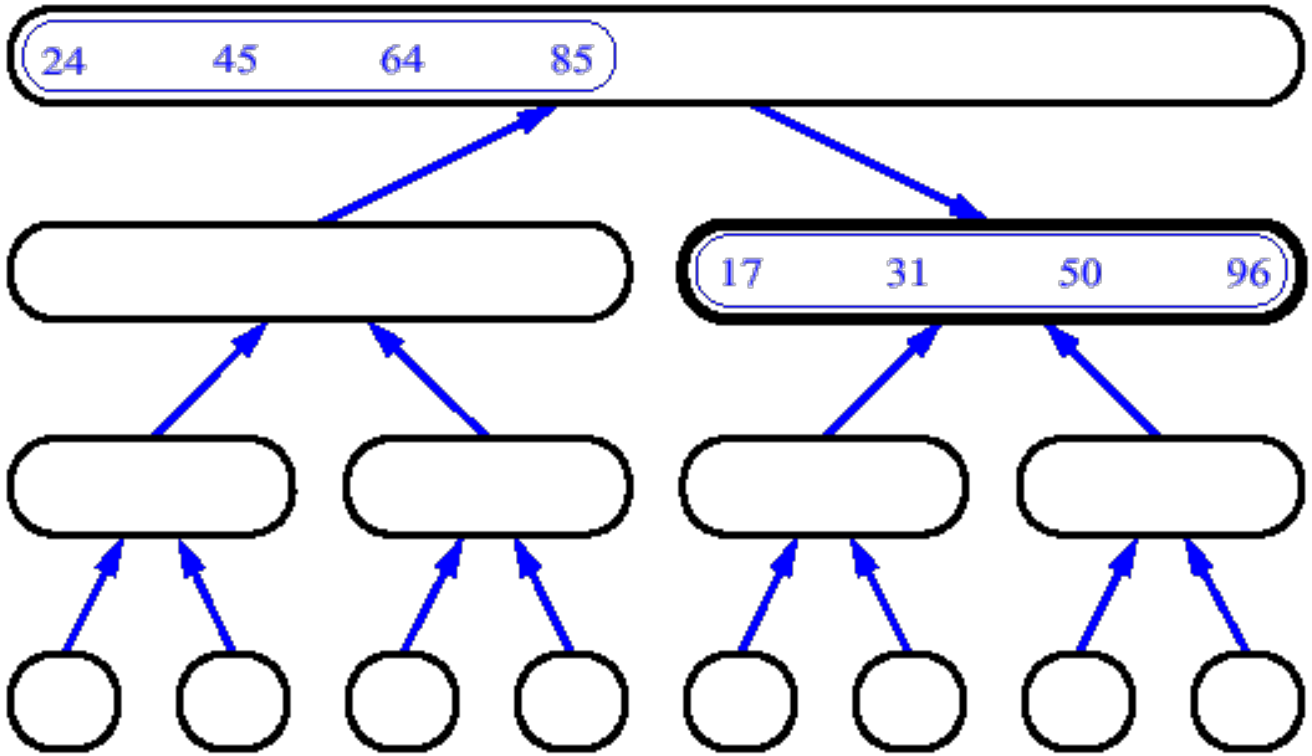
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 19



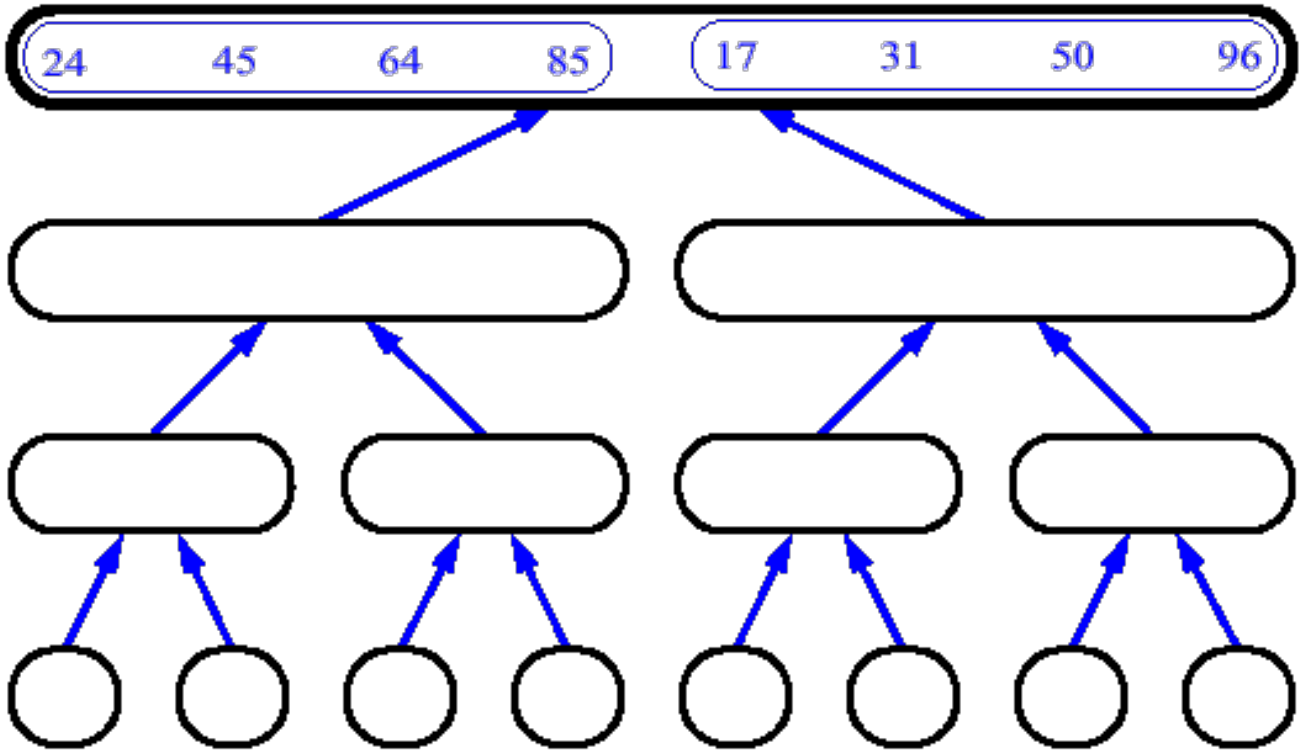
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 20



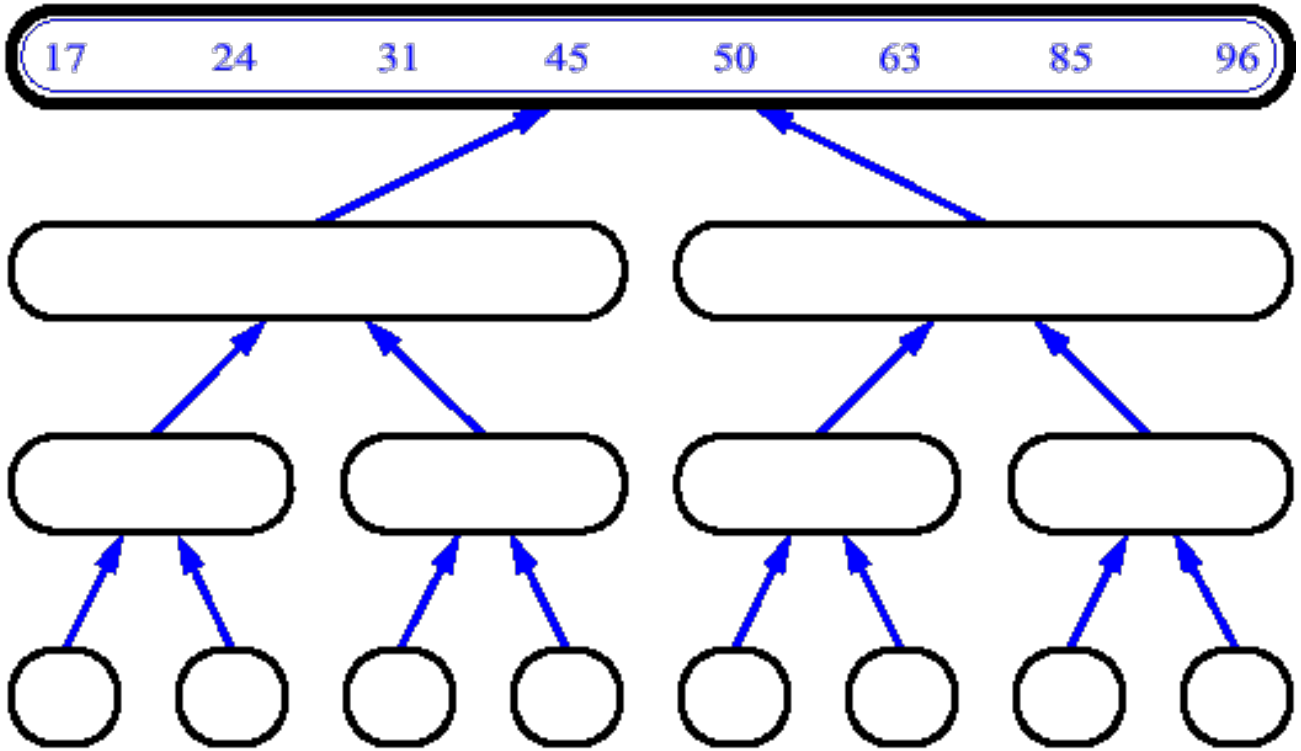
Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 21



Example from Ján Maňuch, Simon Fraser University

MergeSort (Example) — 22

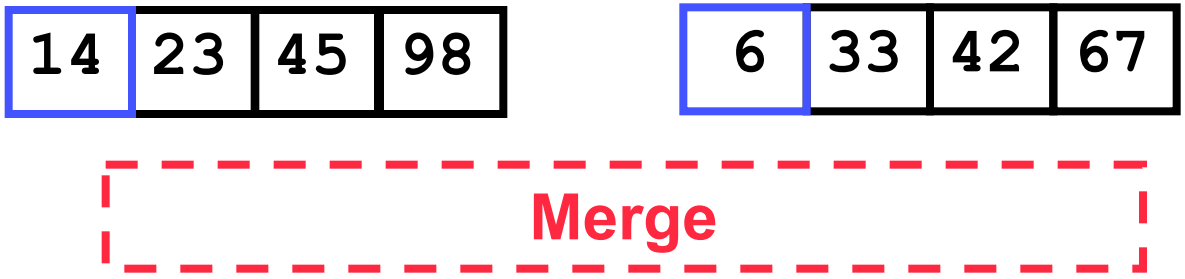


Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)

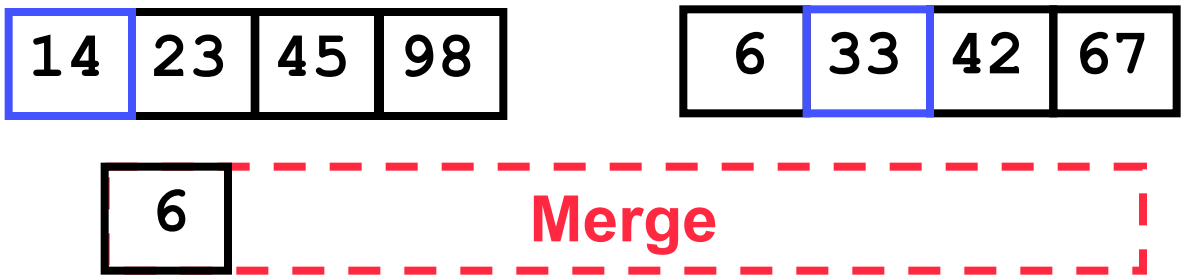
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



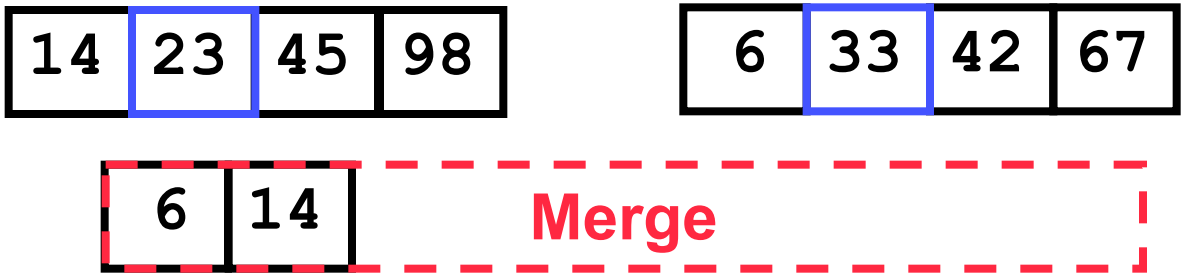
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



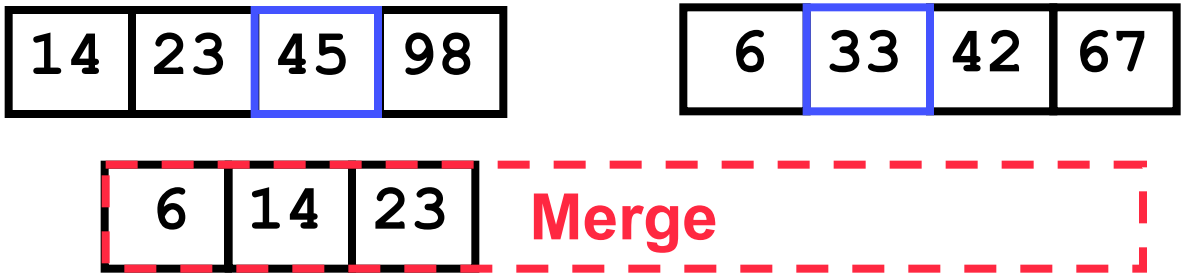
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



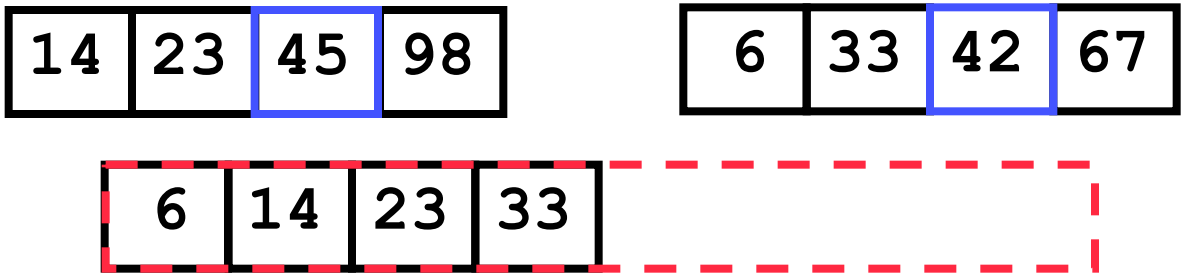
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



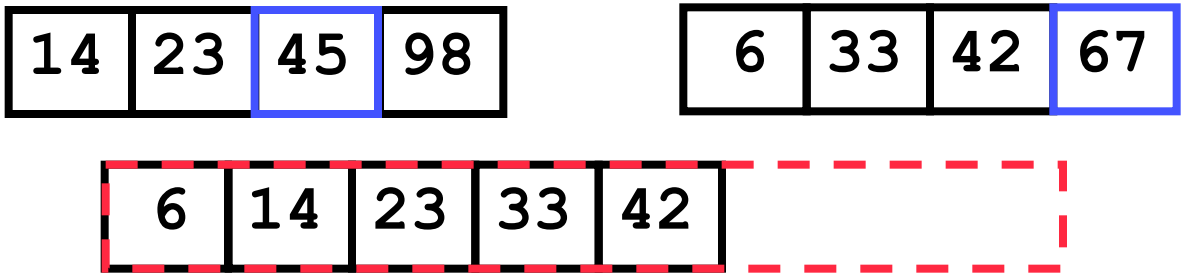
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



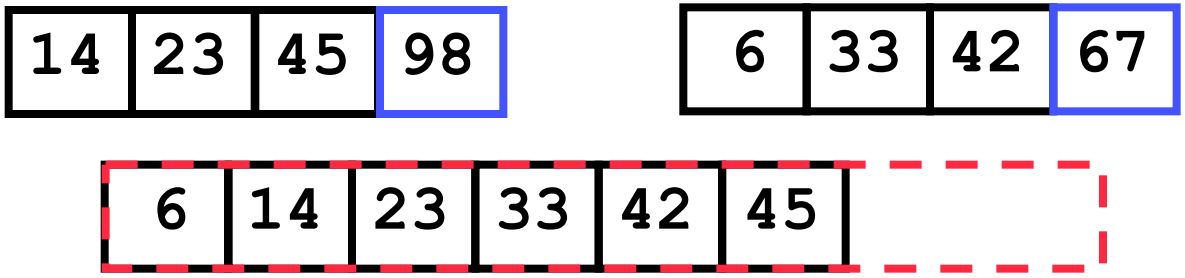
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



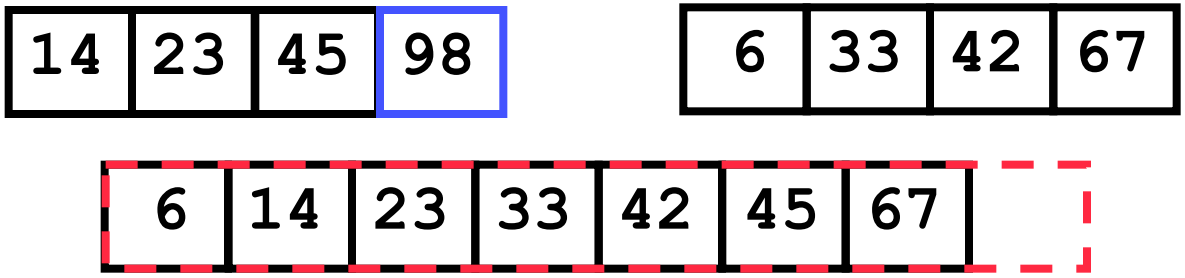
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



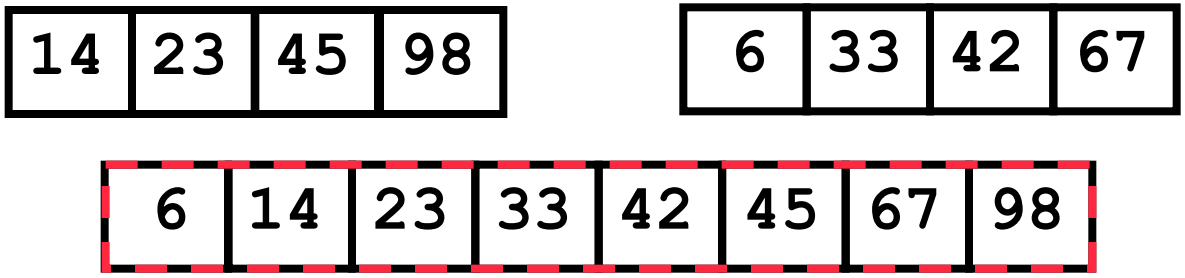
Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



Example from Ján Maňuch, Simon Fraser University

Merge Phase (Example)



Example from Ján Maňuch, Simon Fraser University

What's wrong with this Pseudocode?

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \le C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

```
1
2  /* Java program for Merge Sort from GeeksForGeeks.com */
3  class MergeSort
4  {
5      void merge(int arr[], int l, int m, int r) {
6          // Merges two subarrays of arr[].
7          // First subarray is arr[l..m]
8          // Second subarray is arr[m+1..r]
9
10         int n1 = m - l + 1;
11         int n2 = r - m;
12
13         /* Create temp arrays and copy data into them */
14         int L[] = new int [n1];
15         int R[] = new int [n2];
16         for (int i=0; i<n1; ++i)
17             L[i] = arr[l + i];
18         for (int j=0; j<n2; ++j)
19             R[j] = arr[m + 1+ j];
20
21
22         /* Merge the temp arrays */
23
24
```

```
25          int i = 0, j = 0;  // indexes of first and second subarrays
26          int k = l;         // index of merged subarry array
27
28          while (i < n1 && j < n2) {
29              if (L[i] <= R[j]) {
30                  arr[k] = L[i];
31                  i++;
32              } else {
33                  arr[k] = R[j];
34                  j++;
35              }
36              k++;
37          }
38
39          /* Copy remaining elements of L[] if any */
40          while (i < n1) {
41              arr[k] = L[i];
42              i++;
43              k++;
44          }
45
46          /* Copy remaining elements of R[] if any */
47          while (j < n2) {
48              arr[k] = R[j];
49              j++;
50              k++;
51          }
52      }
```

```
54      // Main function that sorts arr[l..r] using
55      // merge()
56      void sort(int arr[], int l, int r)
57      {
58          if (l < r) {
59              // Find the middle point
60              int m = (l+r)/2;
61
62              // Sort first and second halves
63              sort(arr, l, m);
64              sort(arr , m+1, r);
65
66              // Merge the sorted halves
67              merge(arr, l, m, r);
68          }
69      }
```

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

from Sedgewick & Wayne http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/04MergeQuick.pdf

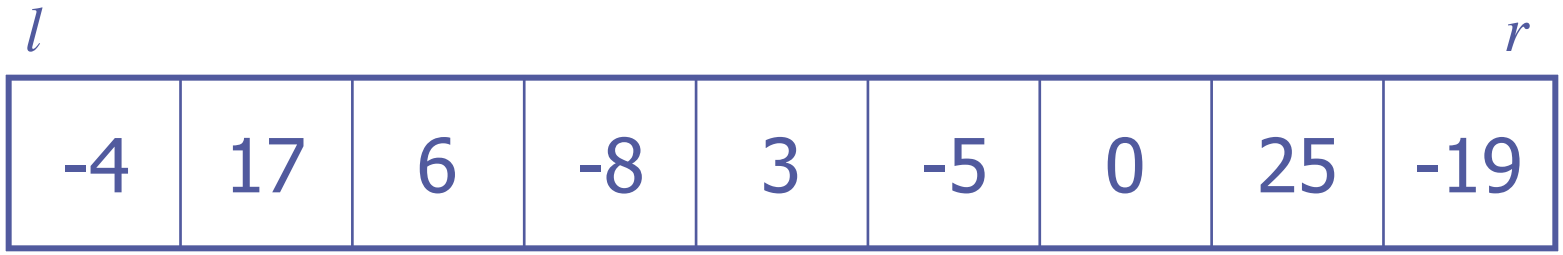
16

Problem

- Is Mergesort a stable sorting algorithm?
  - A yes
  - B no

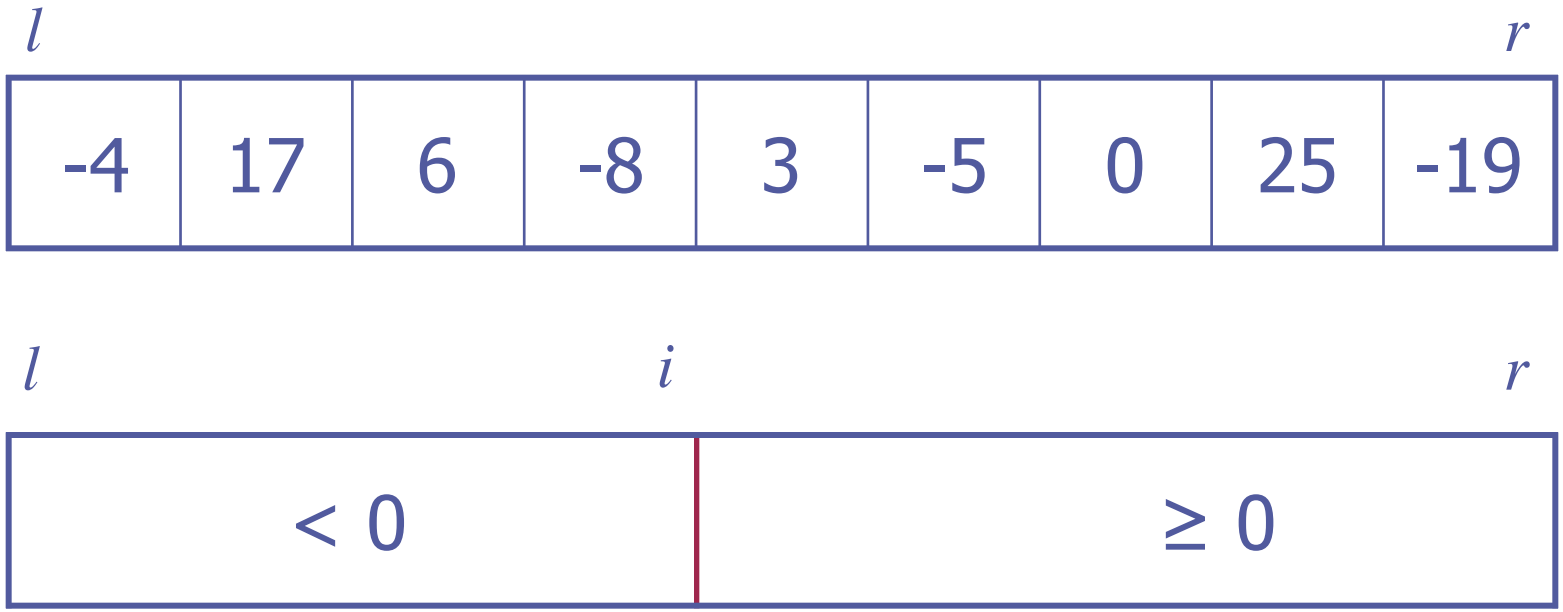
Problem

- Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its non-negative elements. Your algorithm should be both time-efficient and space-efficient.



Problem

- Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its non-negative elements. Your algorithm should be both time-efficient and space-efficient.



Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by
increment s;
swap $A[i]$ and $A[s]$;
increment i

Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by

increment s;

swap $A[i]$ and $A[s]$;

increment i

Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by

increment s;

swap $A[i]$ and $A[s]$;

increment i

Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by

increment s;

swap $A[i]$ and $A[s]$;

increment i

Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by

increment s;

swap $A[i]$ and $A[s]$;

increment i

Partition based on Lomuto's idea

- Examine $A[i]$:

≥ 0: just increment i

< 0: make room in the segment $A[i..s]$ by

increment s;

swap $A[i]$ and $A[s]$;

increment i

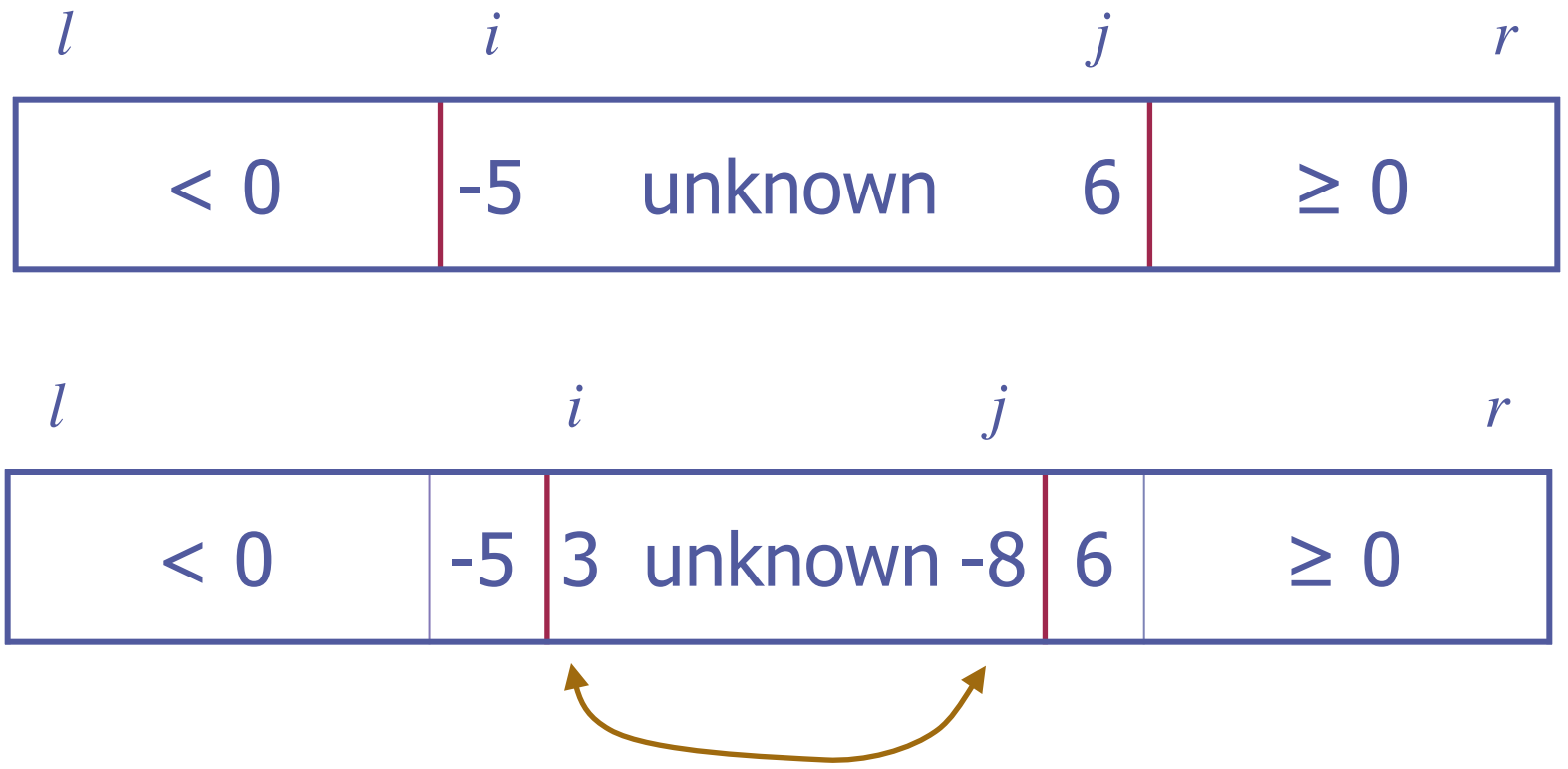
Partition based on Hoare's idea

Partition based on Hoare's idea

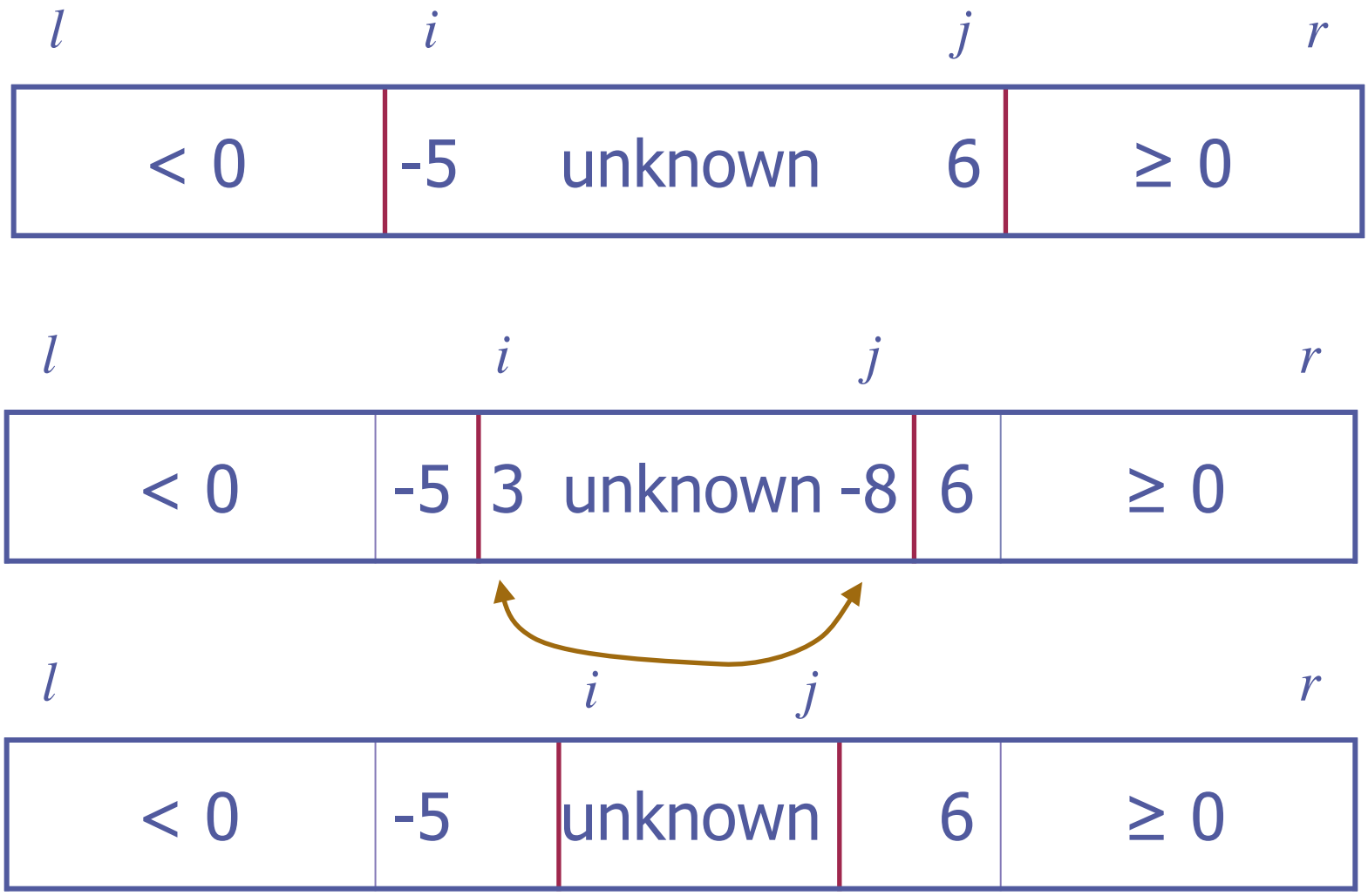
Partition based on Hoare's idea

Partition based on Hoare's idea

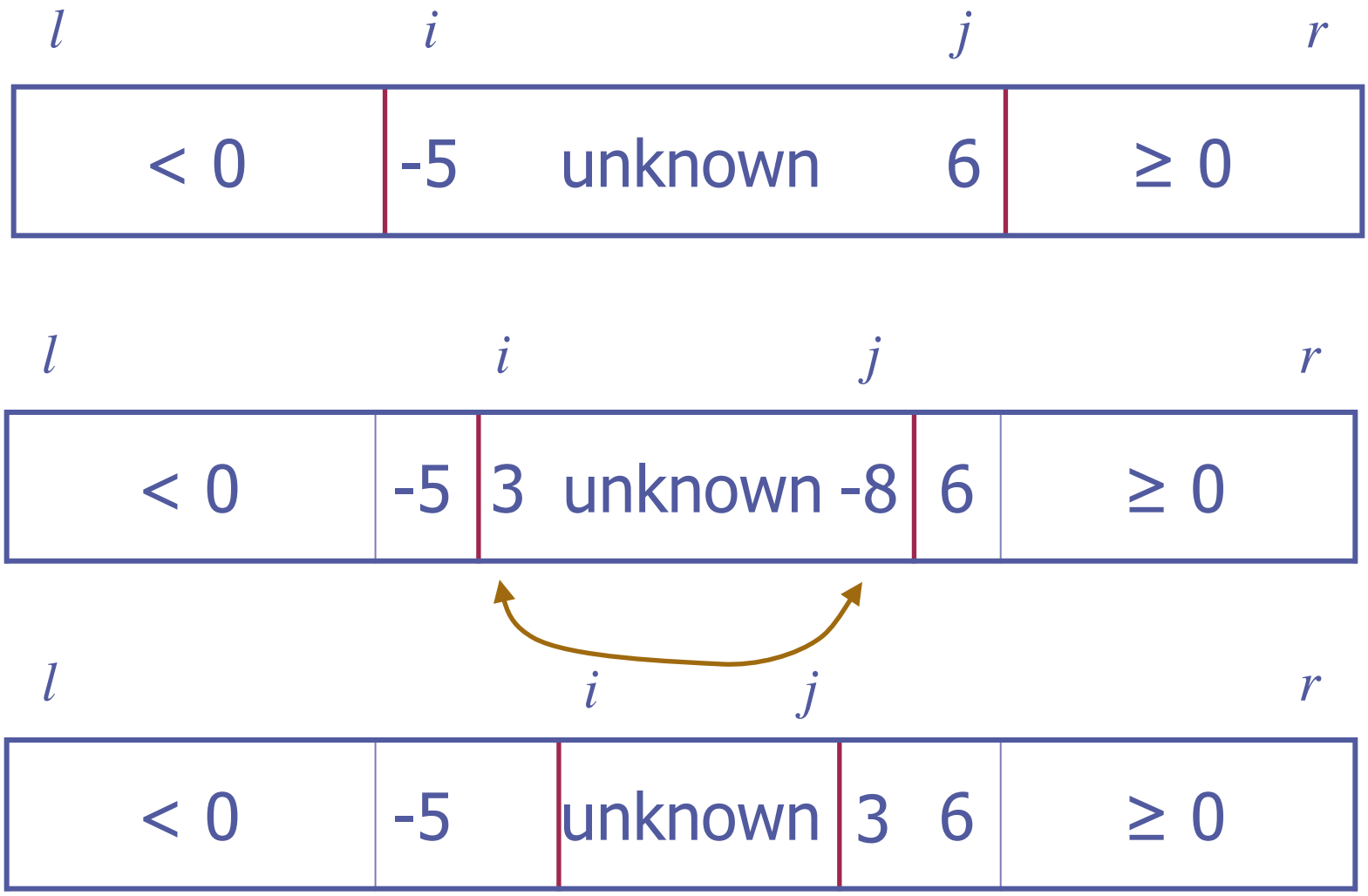
Partition based on Hoare's idea



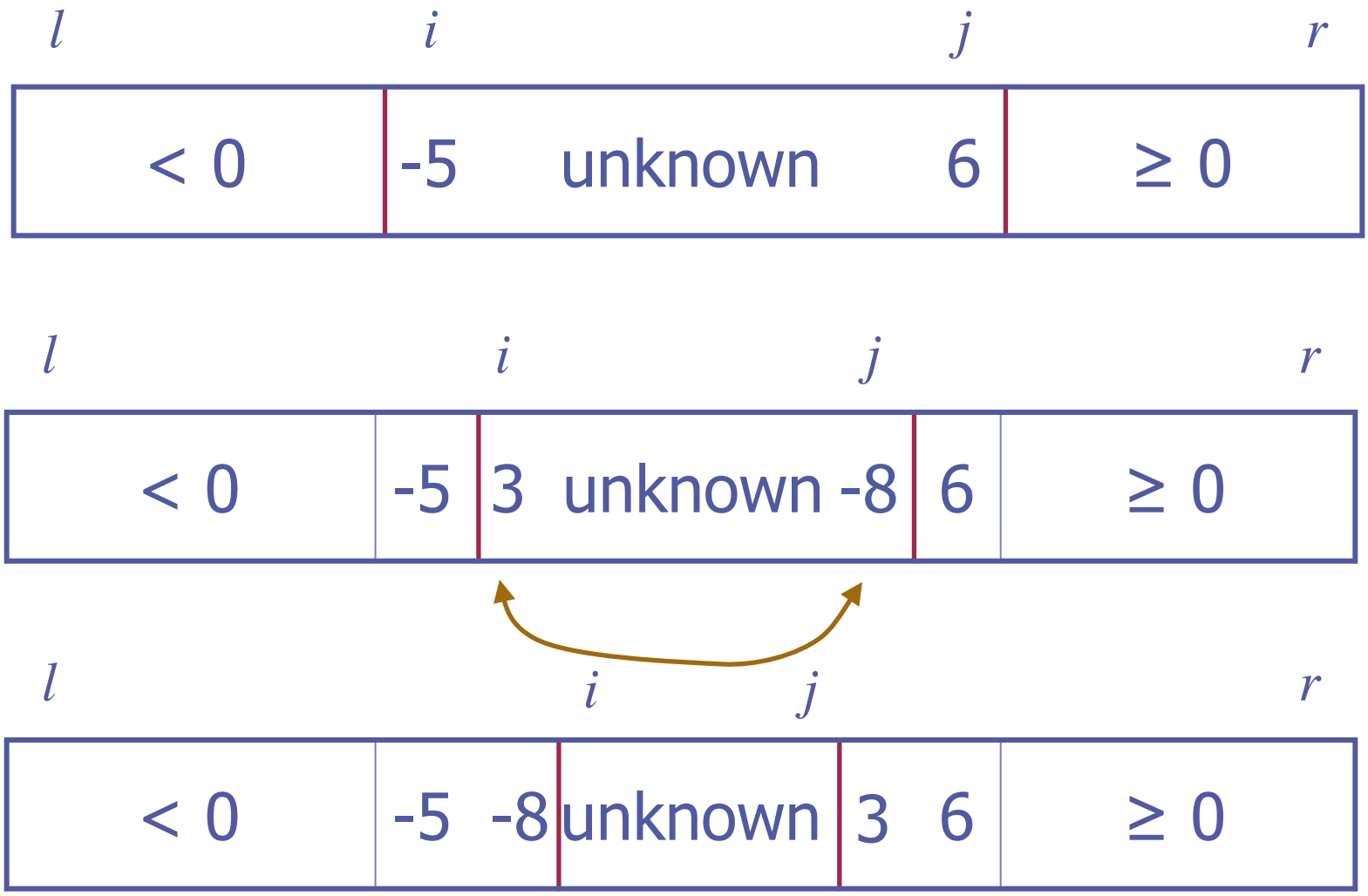
Partition based on Hoare's idea



Partition based on Hoare's idea

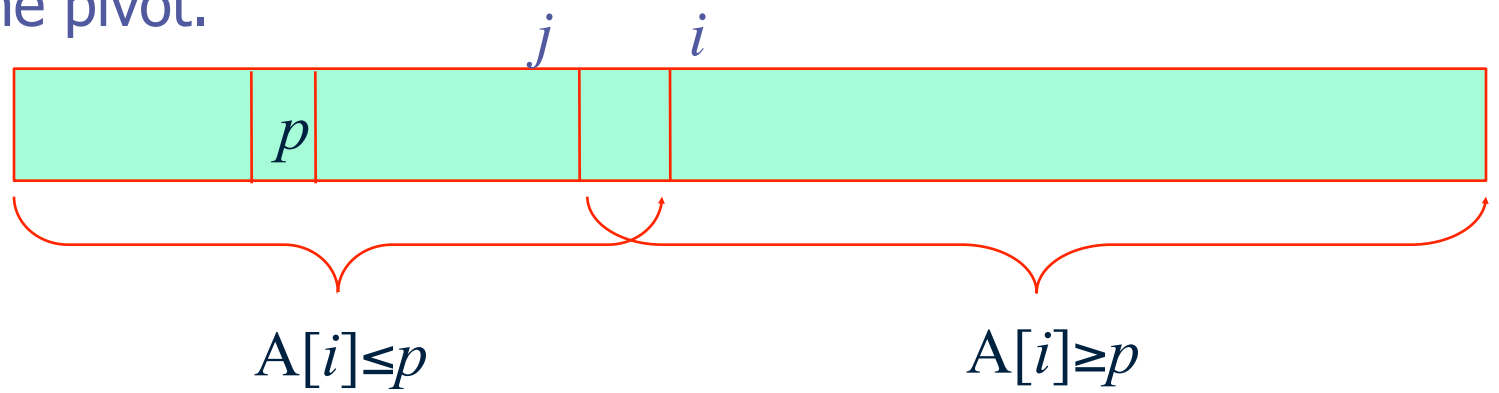


Partition based on Hoare's idea



Quicksort

- Select a pivot (partitioning element) – a **random** element
- Rearrange the list so that all the elements in the first j positions are smaller than or equal to the pivot and all the elements in the last n-i positions are larger than or equal to the pivot.



- Exchange the pivot with the element in its partition closest to the center — the pivot is now in its final position
- Sort the two subarrays recursively

Hoare Partition Algorithm

```
method partition(A, lo, hi) {
        // A is an array; lo and hi are valid indices into A.
        // Returns a 2-element list [i, j] such that
        //     lo ≤ j < i ≤ hi,  provided lo < hi
        //     ∃ pivot st:
        //         A[r] ≤ pivot ∀ lo ≤ r ≤ j
        //         A[r] = pivot ∀ j < r < i
        //         A[r] ≥ pivot ∀ i ≤ r
    def pivotIndex = randomBetween(lo)and(hi)
    def pivot = A[pivotIndex]
    var i := lo−1
    var j := hi+1
    while {
        do { i := i + 1 } while { (i ≤ hi) && {A[i] ≤ pivot} }
        do { j := j − 1 } while { (j ≥ lo) && {A[j] ≥ pivot} }
        i < j
    } do { exchange(A, i, j) }
    if (i < pivotIndex) then {
                exchange(A, i, pivotIndex) ;  i := i + 1
    } elseif {j > pivotIndex} then {
                exchange(A, pivotIndex, j) ; j := j − 1
    }
    return [i, j]
}
```

Which Algorithmic Paradigm?

- **Partition**:

A. Brute force
B. Decrease by a constant
C. Decrease by a Variable Amount
D. Divide and Conquer

Which Algorithmic Paradigm?

- Median Finding using Partition:

A. Brute force
B. Decrease by a constant
C. Decrease by a Variable Amount
D. Divide and Conquer

Which Algorithmic Paradigm?

- Sorting using Partition (Quicksort):

A. Brute force
B. Decrease by a constant
C. Decrease by a Variable Amount
D. Divide and Conquer

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: choose 1st element from sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median-of-three partitioning
  - separate partition for keys equal to pivot
  - switch to insertion sort on small sub-problems
  - elimination of recursion
  - These combine to give 20–25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Dual-pivot Quicksort

Vladimir Yaroslavskiy | 11 Sep 12:35 2009

Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort

```
Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is
faster than the known implementations (theoretically and
experimental). I'd like to propose to replace the JDK's
Quicksort implementation by new one.
```

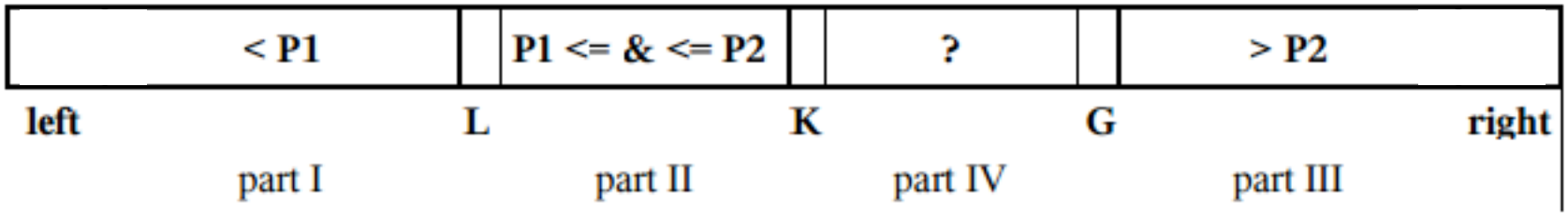
...

```
It is proved that for the Dual-Pivot Quicksort the average number of
comparisons is 2*n*ln(n), the average number of swaps is 0.8*n*ln(n),
whereas classical Quicksort algorithm has 2*n*ln(n) and 1*n*ln(n)
respectively.
```

- Read for yourself: https://web.archive.org/web/20150120151928/http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

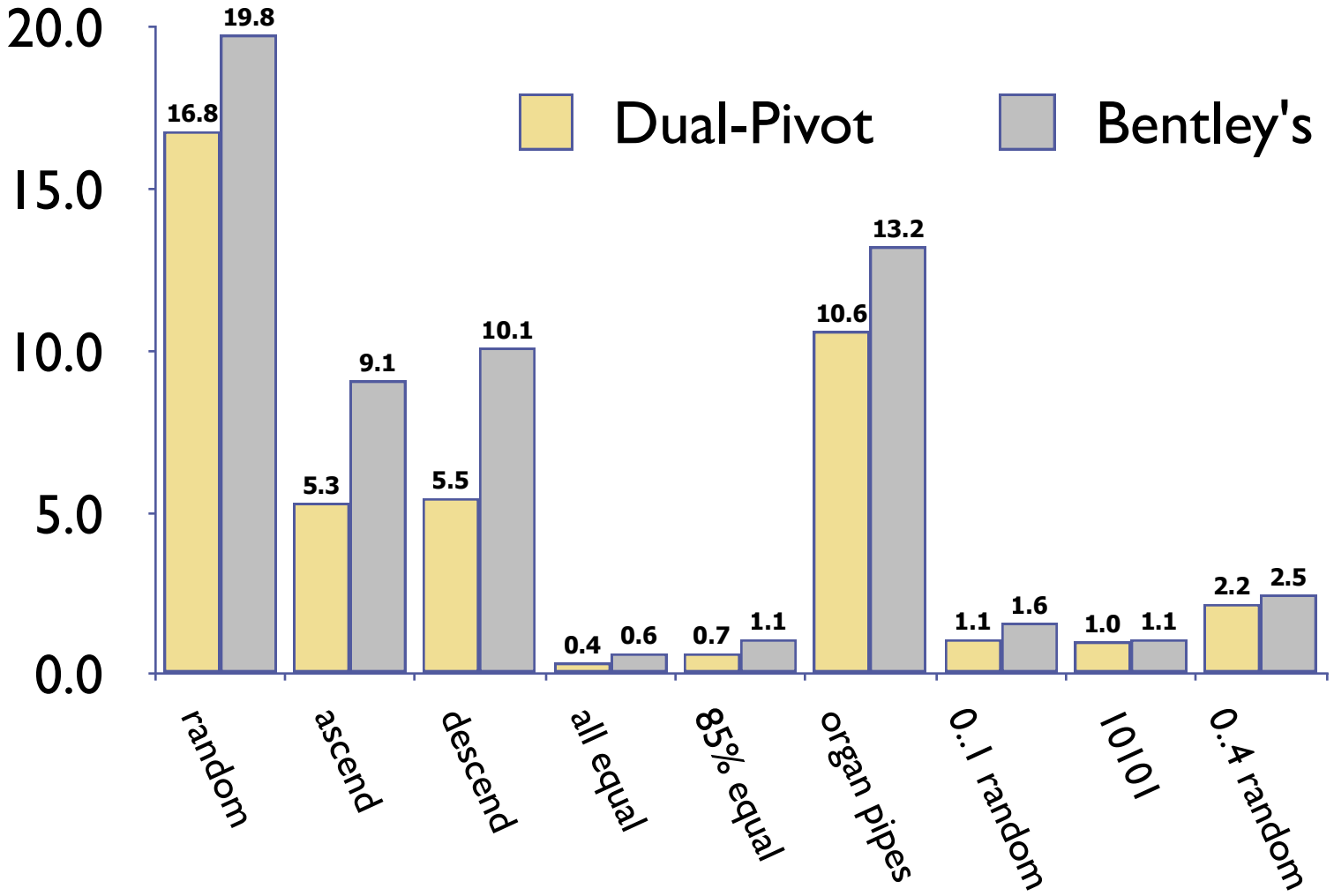
Basic idea

Figure 1.



- Uses <u>two</u> pivots, p_1 and p_2, such that $p_1 \leq p_2$

Performance Comparison



Jon Bentley (9th Sept 2009):

Vladimir, Josh,

I *finally* feel like I understand what is going on. Now that I (think that) I see it, it seems straightforward and obvious.

Tony Hoare developed Quicksort in the early 1960s. I was very proud to make minor contributions to a particularly clean (binary) quicksort in the mid 1980s, to a relatively straightforward, industrial-strength Quicksort with McIlroy in the early 1990s, and then to algorithms and data structures for strings with Sedgewick in the mid 1990s.

I think that Vladimir's contributions to Quicksort go way beyond anything that I've ever done, and rank up there with Hoare's original design and Sedgewick's analysis. I feel so privileged to play a very, very minor role in helping Vladimir with the most excellent work!

Problem:

- Is Quicksort stable?

A. Yes
B. No

Problem:

Should we stop scanning when we find an element = pivot?

- Levitin: **repeat** $i \leftarrow i + 1$ **until** $A[i] \geq pivot$
- Hoare: **if** $pivot < A[I]$ **then goto** $down$

✧

Problem:

Should we stop scanning when we find an element = pivot?

Levitin: **repeat** $i \leftarrow i + 1$ **until** $A[i] \geq pivot$

Hoare: **if** $pivot < A[I]$ **then goto** *down*

Levitin claims: *"Why is it worth stopping the scans after encountering an element equal to the pivot? Because doing this tends to yield more-even splits for arrays with a lot of duplicates, which makes the algorithm run faster."*

QUICKSORT IS OPTIMAL

Robert Sedgewick
Jon Bentley

- "Knuthfest", Stanford University, January, 2002.
  - http://www.sorting-algorithms.com/static/QuicksortIsOptimal.pdf
- Examines an interesting detail: How to handle keys equal to the pivot

Partitioning with equal keys

How to handle keys equal to the partitioning element?

METHOD A: Put equal keys all on one side?

```
4  4  4  4  4  4  4  4  4  4  4  4  4
4  4  4  4  4  4  4  4  4  4  4  4 (4)
```

NO: quadratic for n=1 (all keys equal)

METHOD B: Scan over equal keys? (linear for n=1)

```
1  4  1  1  4  4  4  1  4  1  1  4  4
1 (1) 1  1  4  4  4  1  4  1 (4) 4  4
```

NO: quadratic for n=2

METHOD C: Stop both pointers on equal keys?

```
4  9  4  4  1  4  4  4  9  4  4  1  4
1  4  4  4  1 (4) 4  4  9  4  9  4  4
```

YES: NlgN guarantee for small n, no overhead if no equal keys

Partitioning with equal keys

How to handle keys equal to the partitioning element?

METHOD C: Stop both pointers on equal keys?

YES: NlgN guarantee for small n, no overhead if no equal keys

METHOD D (3-way partitioning): Put all equal keys into position?

yes, BUT: early implementations cumbersome and/or expensive

Quicksort common wisdom (last millennium}

1. Method of choice in practice
   - tiny inner loop, with locality of reference
   - NlogN worst-case "guarantee" (randomized)
   - but use a radix sort for small number of key values
2. Equal keys can be handled (with care)
   - NlogN worst-case guarantee, using proper implementation
3. Three-way partitioning adds too much overhead
   - "Dutch National Flag" problem
4. Average case analysis with equal keys is intractable
   - keys equal to partitioning element end up in both subfiles

Changes in Quicksort common wisdom

1. Equal keys abound in practice.
   - never can anticipate how clients will use library
   - linear time required for huge files with few key values
2. 3-way partitioning is the method of choice.
   - greatly expands applicability, with little overhead
   - easy to adapt to multikey sort
   - no need for separate radix sort
3. Average case analysis already done!
   - Burge, 1975
   - Sedgewick, 1978
   - Allen, Munro, Melhorn, 1978

Bentley-McIlroy 3-way partitioning

Partitioning invariant

- move from left to find an element that is not less
- move from right to find an element that is not greater
- stop if pointers have crossed
- exchange
- if left element equal, exchange to left end
- if right element equal, exchange to right end

Swap equals to center after partition

KEY FEATURES

- always uses N-1 (three-way) compares
- no extra overhead if no equal keys
- only one "extra" exchange per equal key

Quicksort with 3-way partitioning

```
void quicksort(Item a[], int l, int r)
{ int i = l-1, j = r, p = l-1, q = r; Item v = a[r];
  if (r <= l) return;
  for (;;)
    {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
      if (a[i] == v) { p++; exch(a[p], a[i]); }
      if (v == a[j]) { q--; exch(a[j], a[q]); }
    }
  exch(a[i], a[r]); j = i-1; i = i+1;
  for (k = l; k < p; k++, j--) exch(a[k], a[j]);
  for (k = r-1; k > q; k--, i++) exch(a[i], a[k]);
  quicksort(a, l, j);
  quicksort(a, i, r);
}
```

Sorting analysis summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort (N log N)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

The Problem of the Dutch National Flag

- Levitin §5.2, Q9

The ***Dutch flag problem*** is to rearrange an array of characters R, W, and B (red, white, and blue are the colors of the Dutch national flag) so that all the R's come first, the W's come next, and the B's come last. Design a linear in-place algorithm for this problem.

Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B$\in$P that are both on the convex hull

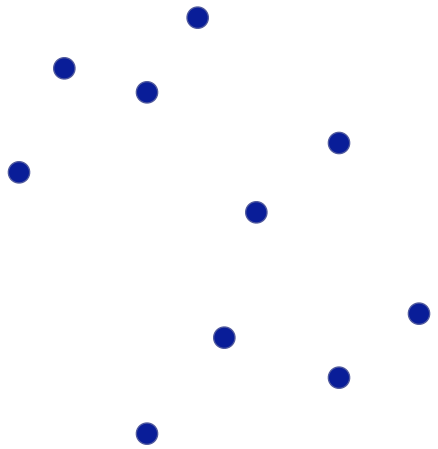
divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside $\triangle$ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B$\in$P that are both on the convex hull

divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside $\triangle$ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r

Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B$\in$P that are both on the convex hull

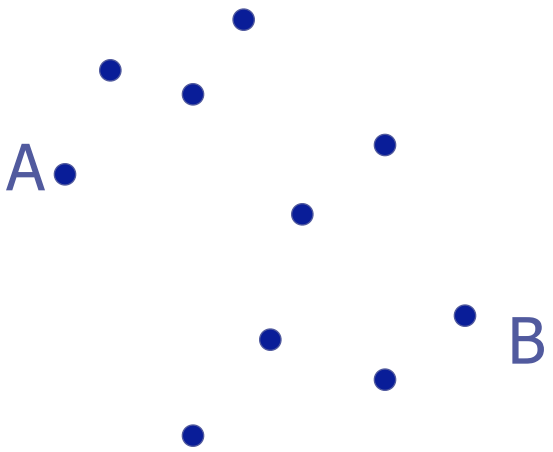
divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside $\triangle$ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B$\in$P that are both on the convex hull

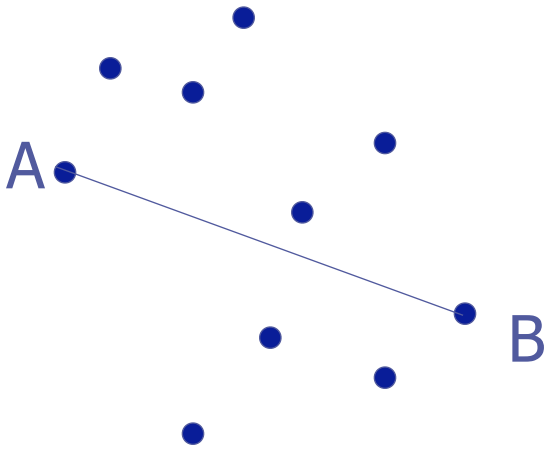
divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside $\triangle$ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B$\in$P that are both on the convex hull

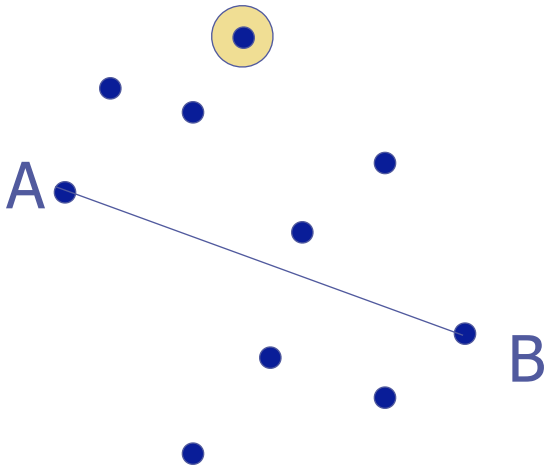
divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside $\triangle$ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

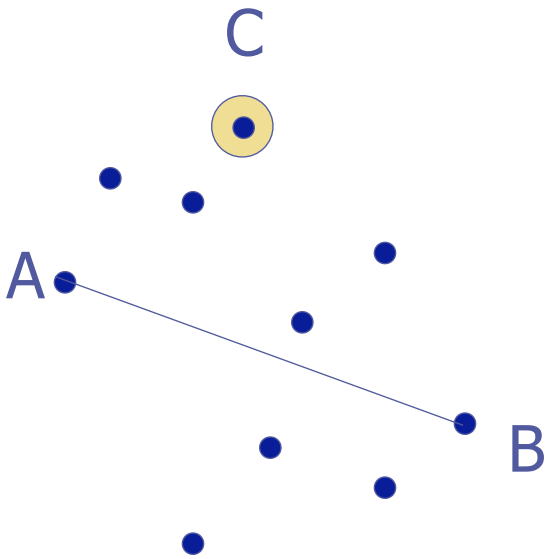
divide the set P into two subsets, P_l left and P_r right of chord AB.

Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

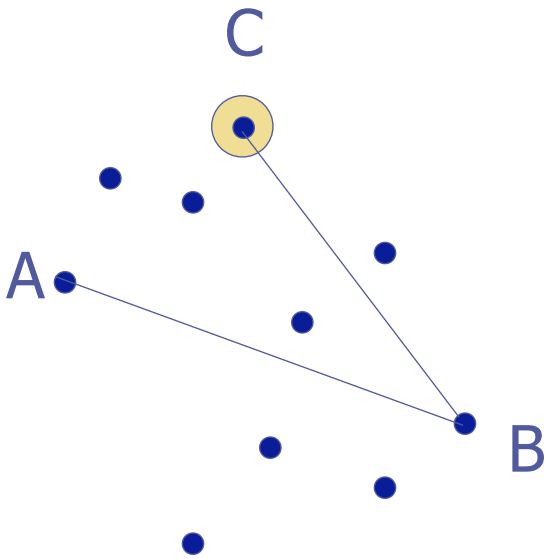
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

  repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

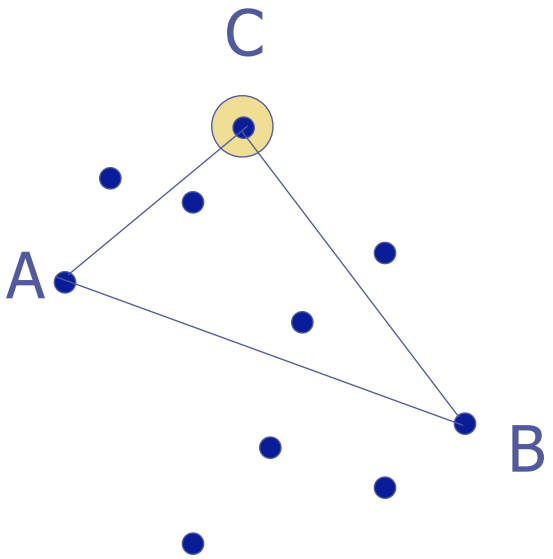
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

  repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

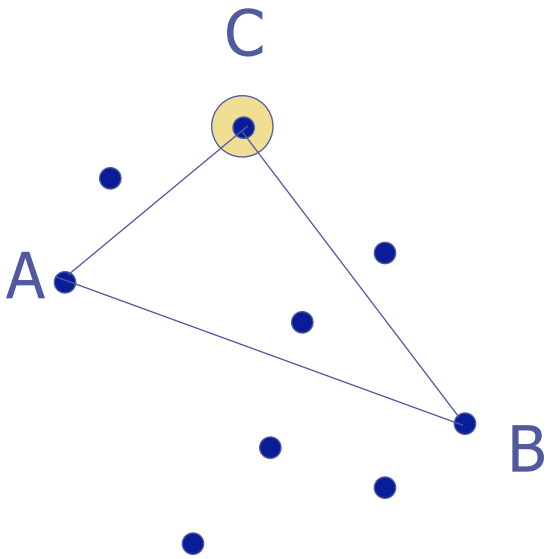
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

  repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

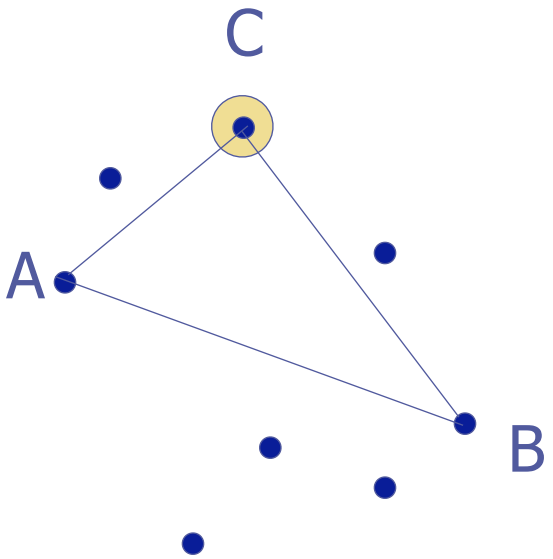
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

  repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

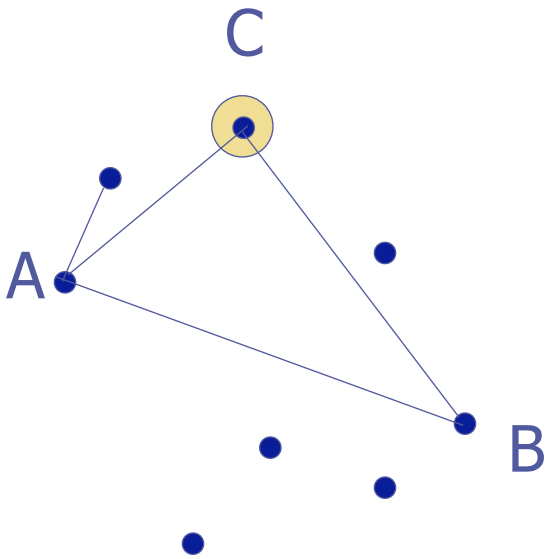
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

    …

  repeat with chord AB and P_r



Quickhull

- Divide-and-conquer algorithm for Convex Hull of a set of points P

  - Find two points A,B∈P that are both on the convex hull

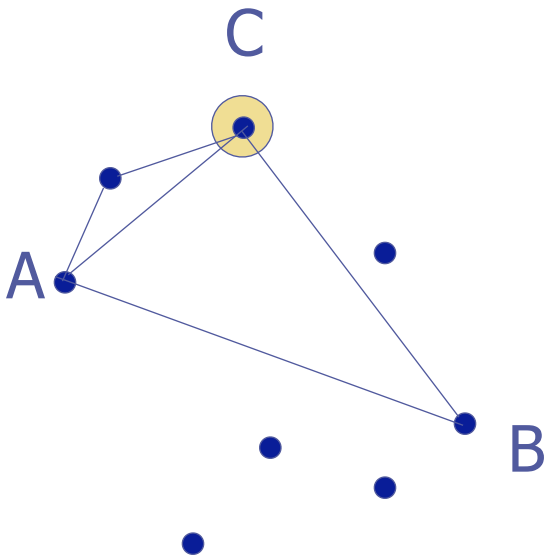
  divide the set P into two subsets, P_l left and P_r right of chord AB.

  Find point C in P_l furthest from AB.

  - discard the points inside △ABC
  - recurse with chords AC and CB

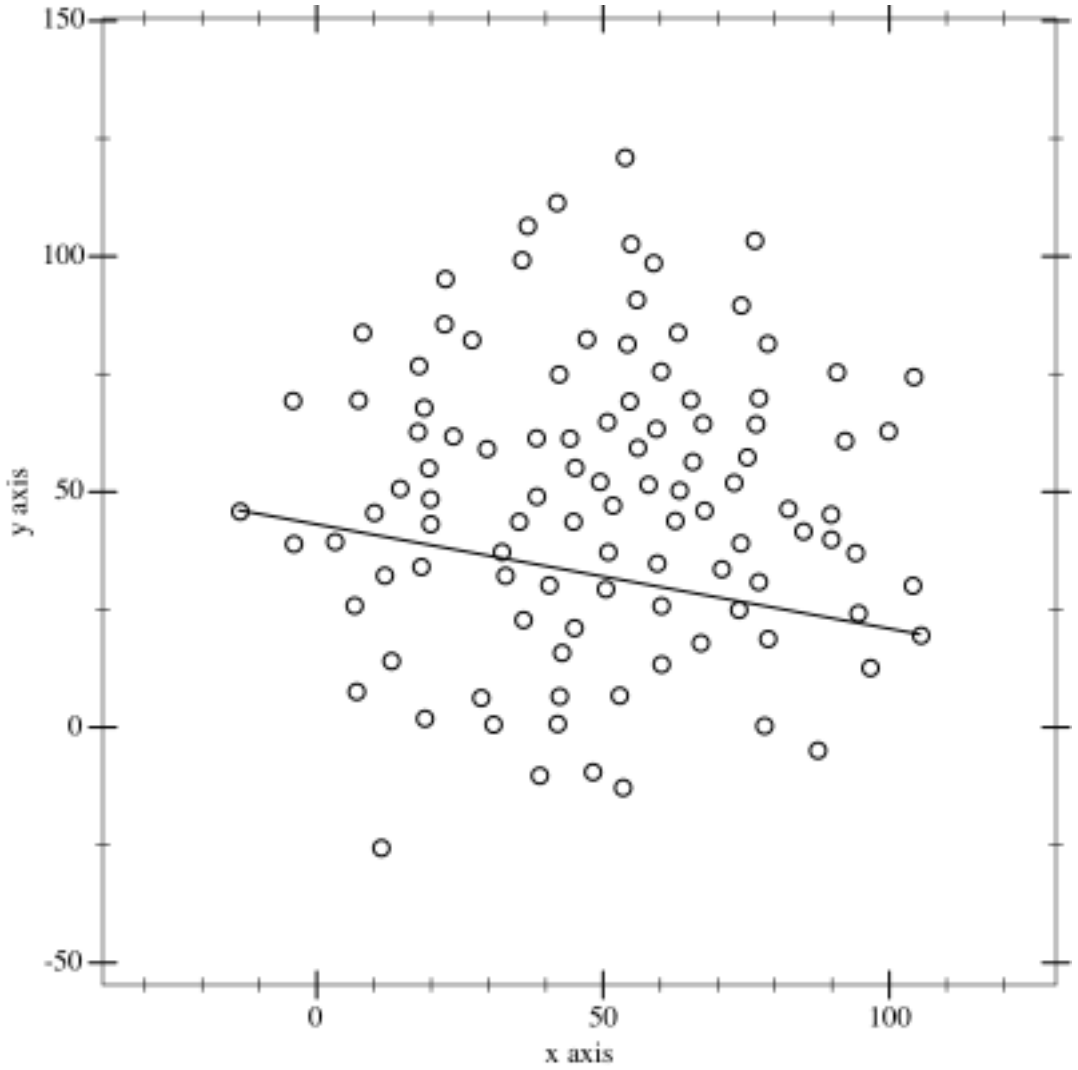
    …

  repeat with chord AB and P_r

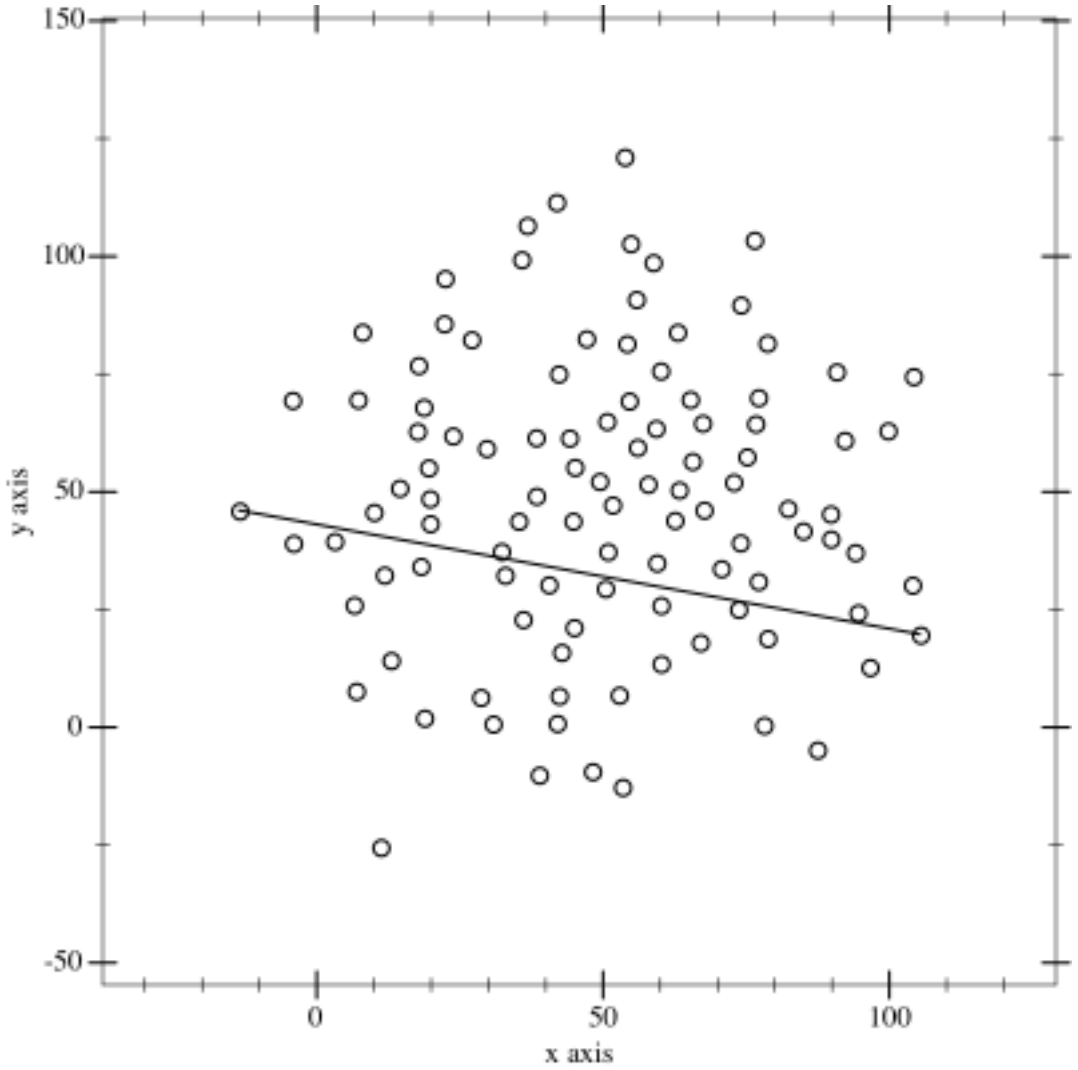


Animation:

- Quickhull at Princeton (needs Java)



Animation by Maonus - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=37505459



Animation by Maonus - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=37505459