Data Structures and Algorithms

Part 3

Werner Nutt

Acknowledgments

- The course follows the book “Introduction to Algorithms‘”, by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for this course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Part 3: Overview

- Divide and conquer
- Merge sort, repeated substitutions
- Tiling
- Recurrences

Divide and Conquer

Principle:

If the problem size is small enough to solve it trivially, solve it. Else:

- Divide: Decompose the problem into two or more disjoint subproblems.
- Conquer: Use divide and conquer recursively to solve the subproblems.
- Combine: Take the solutions to the subproblems and combine the solutions into a solution for the original problem.

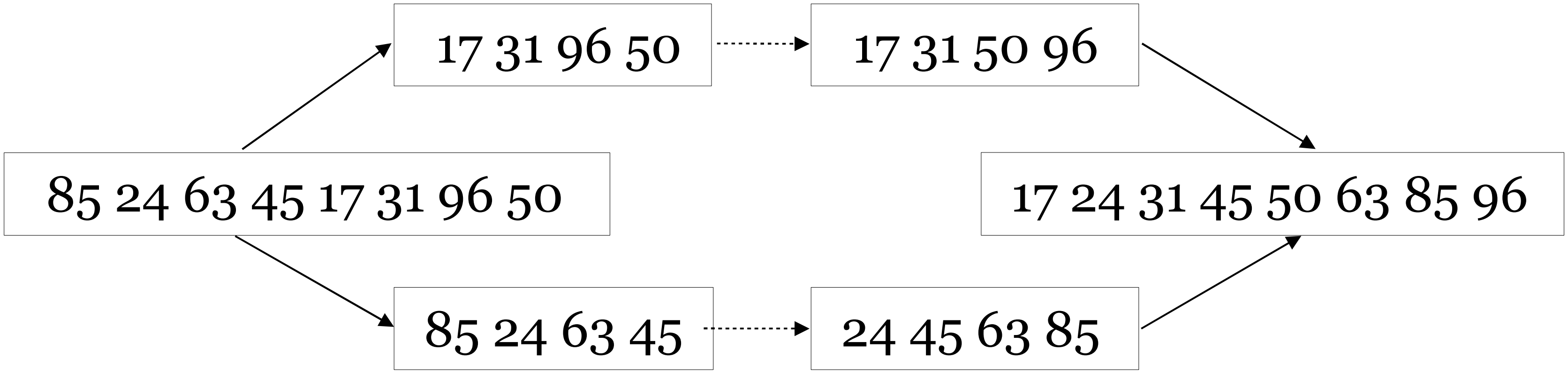
Picking a Decomposition

- Finding a decomposition requires some practice and is the key part.
- The decomposition has the following properties:
  - It reduces the problem to a “smaller problem”.
  - Often the smaller problem is identical to the original problem.
  - A sequence of decompositions eventually yields the base case.
  - The decomposition must contribute to solving the original problem.

Merge Sort

Sort an array by

- Dividing it into two arrays.
- Sorting each of the arrays.
- Merging the two arrays.



85 24 63 45 17 31 96 50

17 31 96 50

17 31 50 96

85 24 63 45

24 45 63 85

17 24 31 45 50 63 85 96

Merge Sort Algorithm

Divide: If S has at least two elements,
put them into sequences S_1 and S_2.
S_1 contains the first $\lceil n/2 \rceil$ elements and
S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.

Conquer: Sort sequences S_1 and S_2
using merge sort.

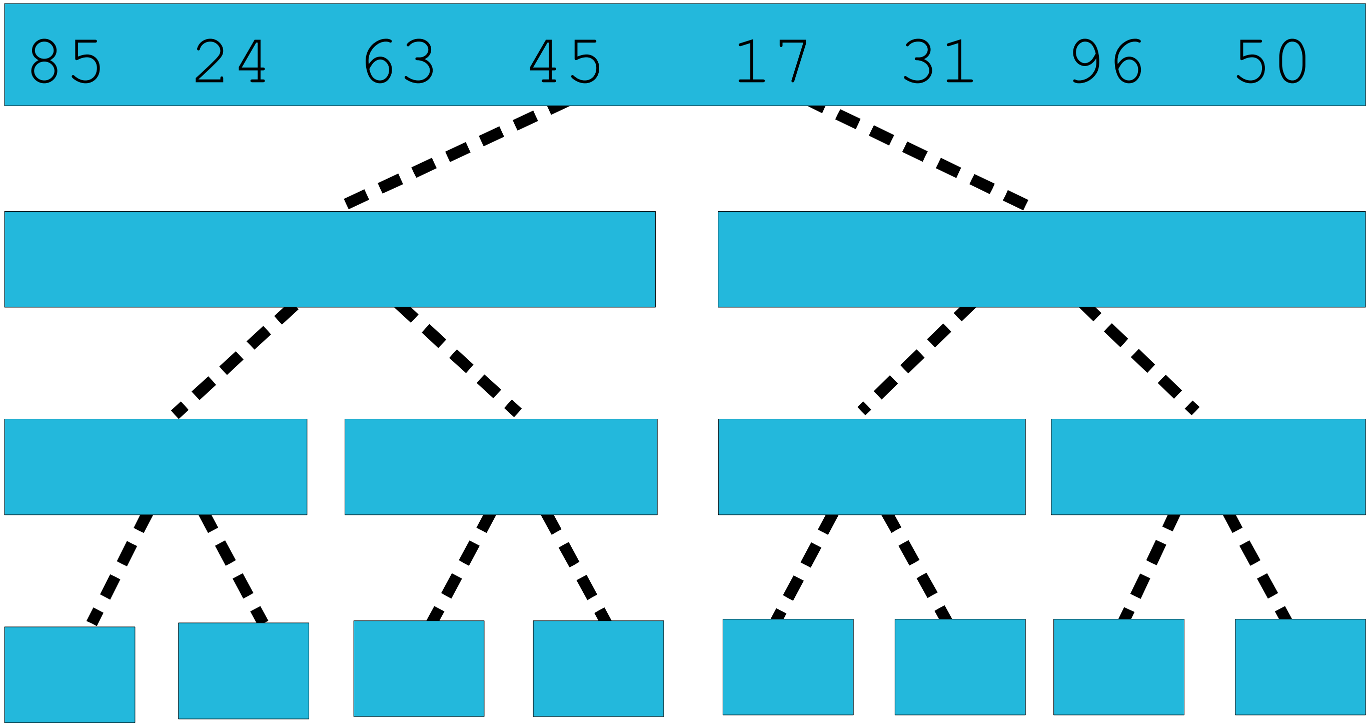
Combine: Put back the elements into S
by merging the sorted sequences S_1 and S_2
into one sorted sequence.

Merge Sort: Algorithm

```
MergeSort(l, r)
    if l < r then
        m := (l+r)/2
        MergeSort(l,m)
        MergeSort(m+1,r)
        Merge(l,m,r)
```

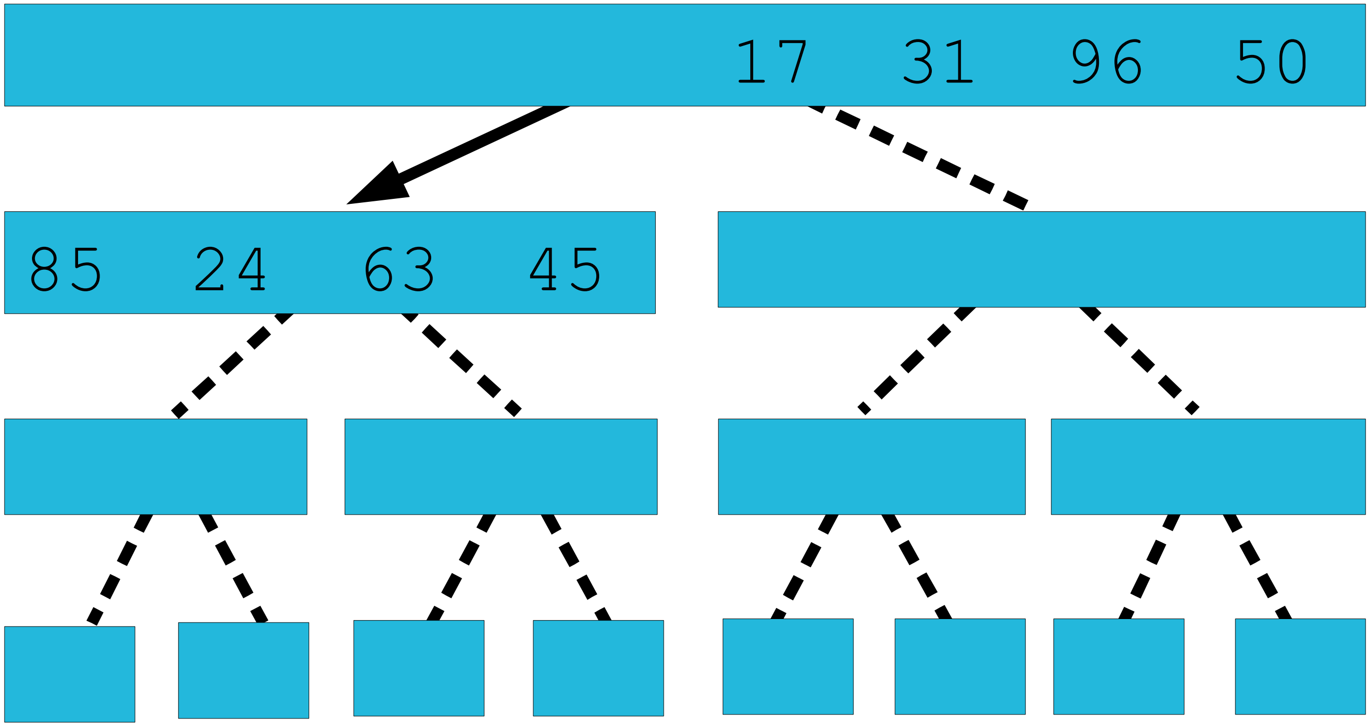
```
Merge(l, m, r)
Take the smallest of the two first elements
of sequences A[l..m] and A[m+1..r]
and put it into the resulting sequence.
Repeat this, until both sequences are empty.
Copy the resulting sequence into A[l..r].
```

MergeSort Example/1



85 24 63 45 17 31 96 50

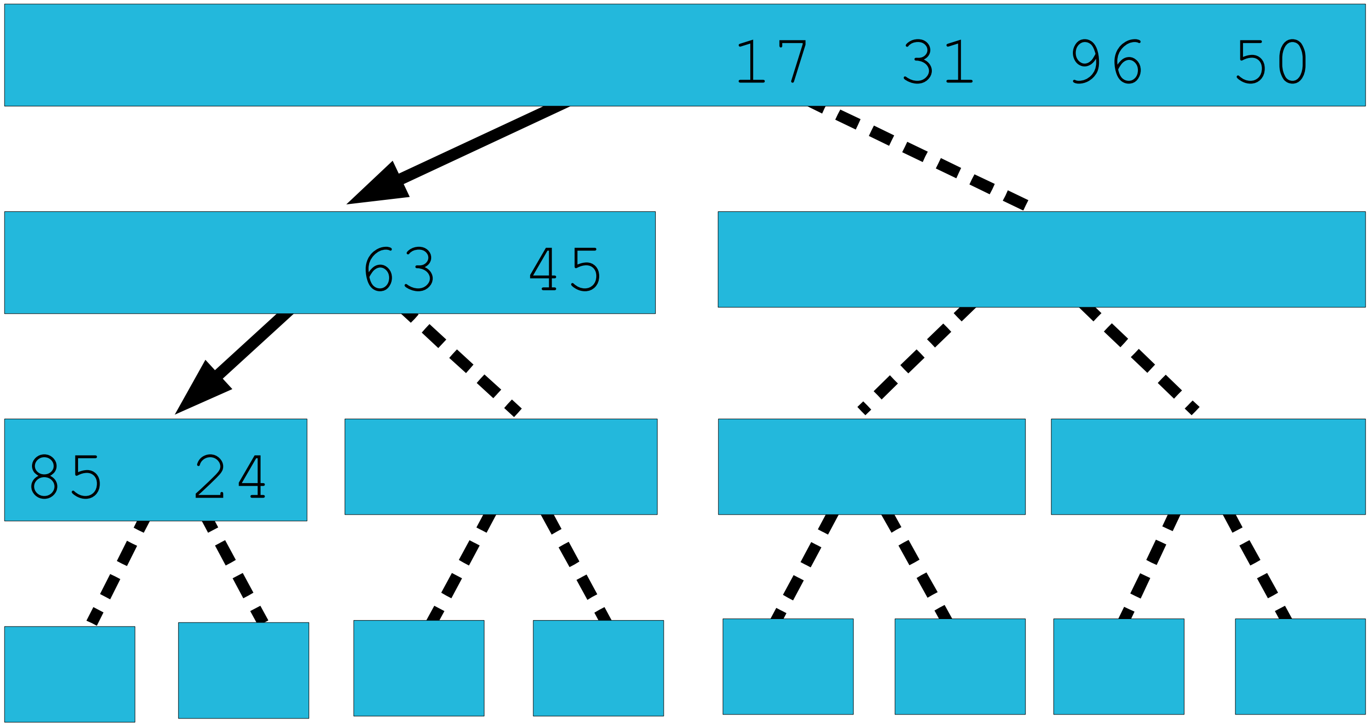
MergeSort Example/2



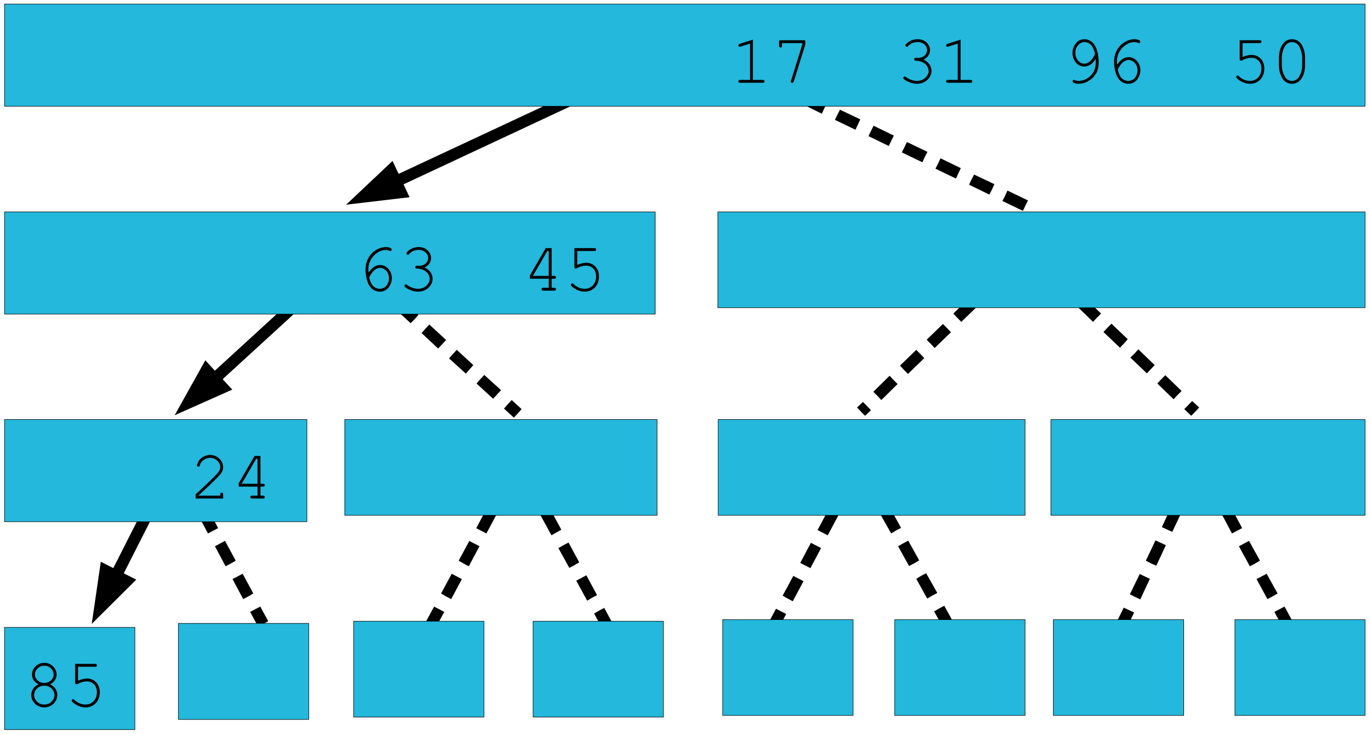
17 31 96 50

85 24 63 45

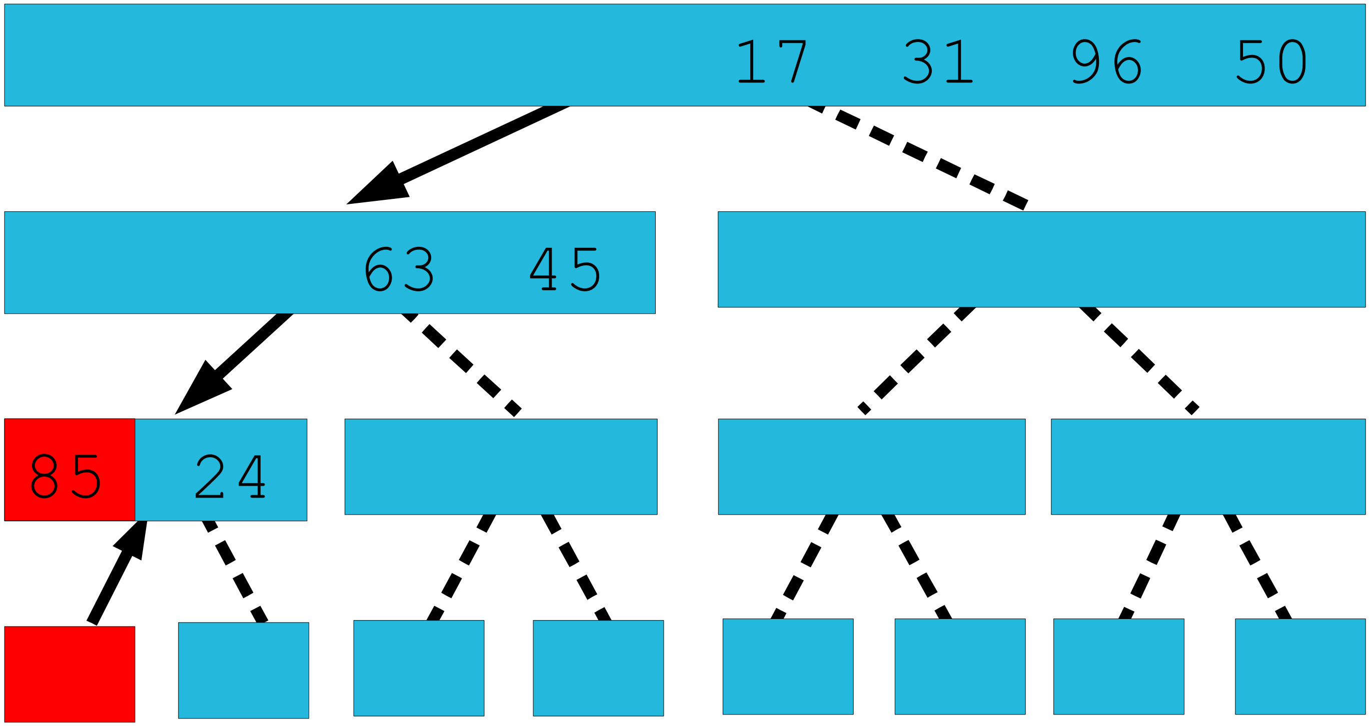
MergeSort Example/3



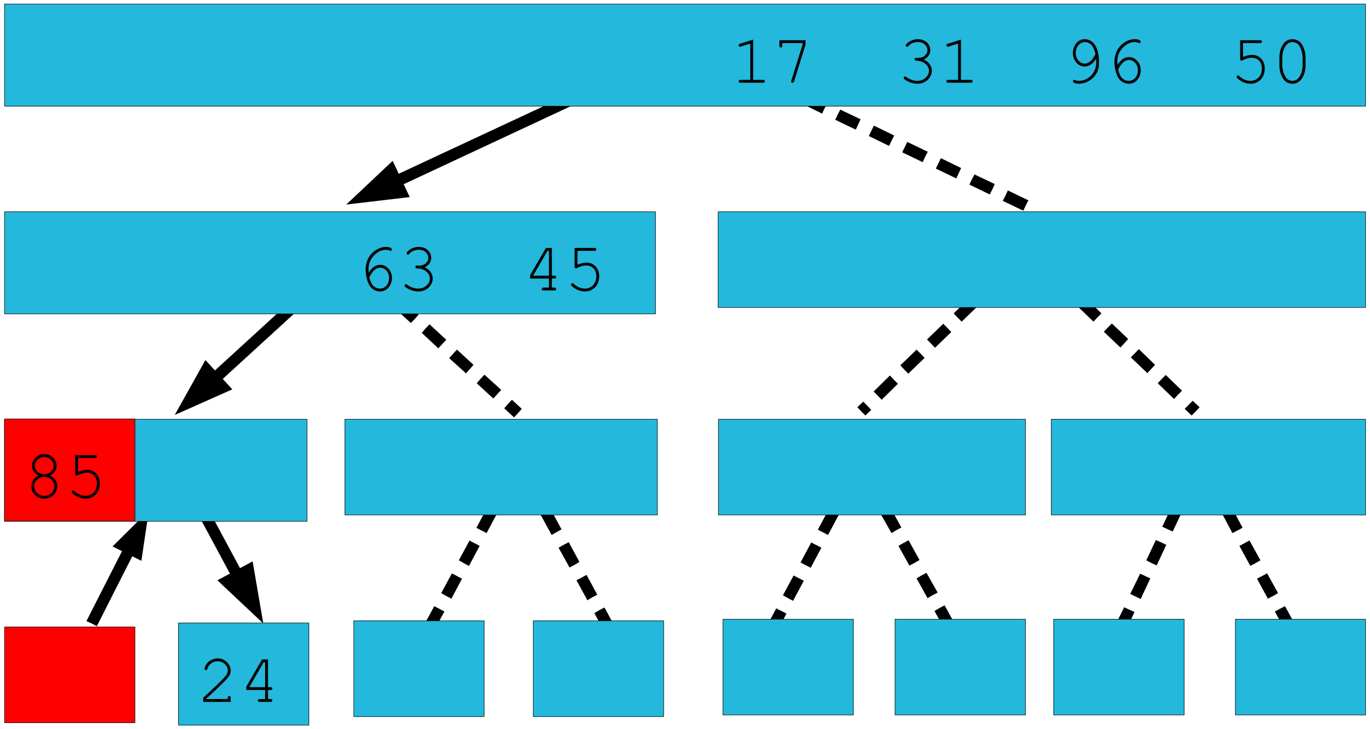
MergeSort Example/4



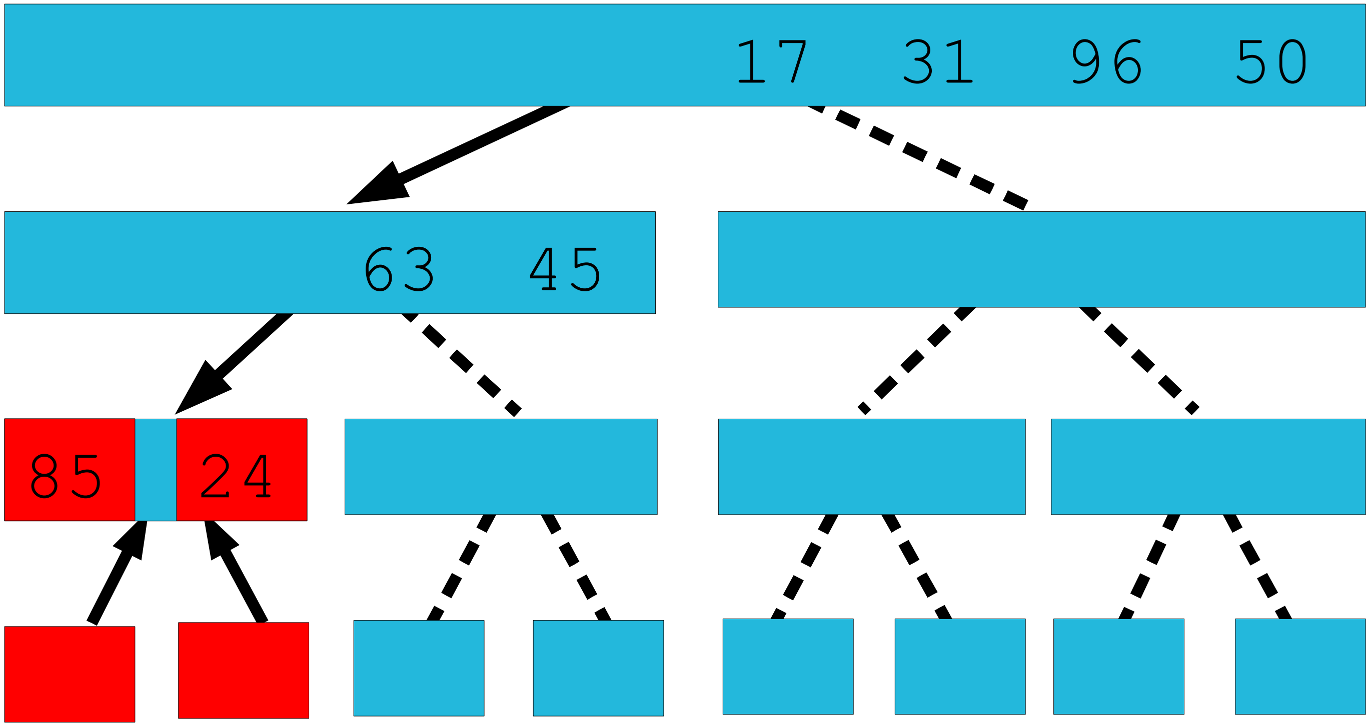
MergeSort Example/5



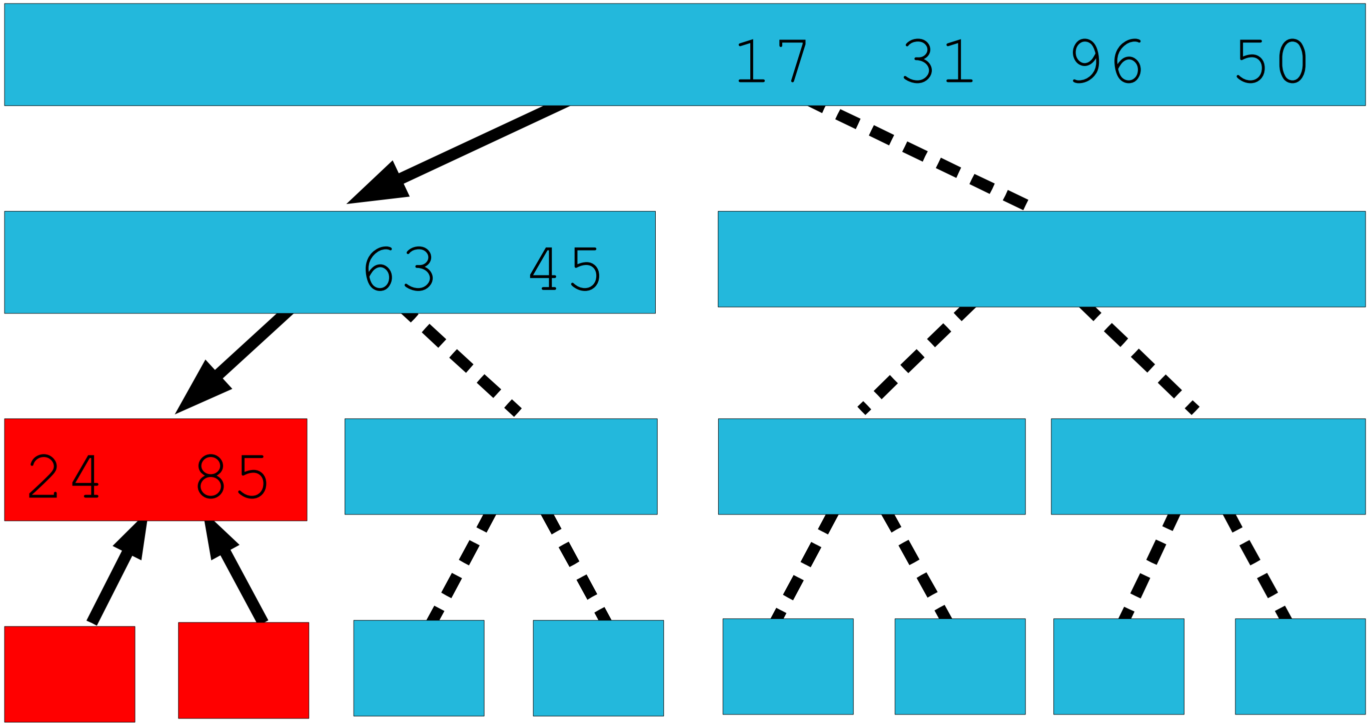
MergeSort Example/6



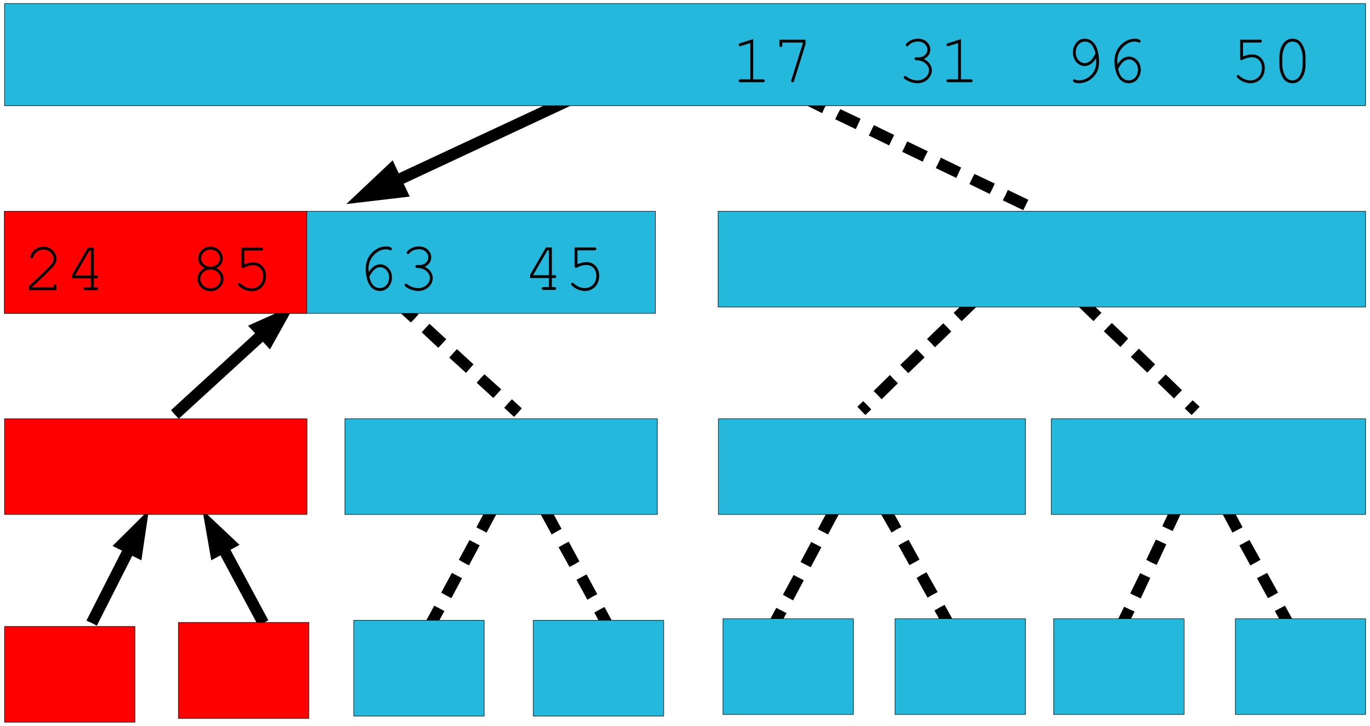
MergeSort Example/7



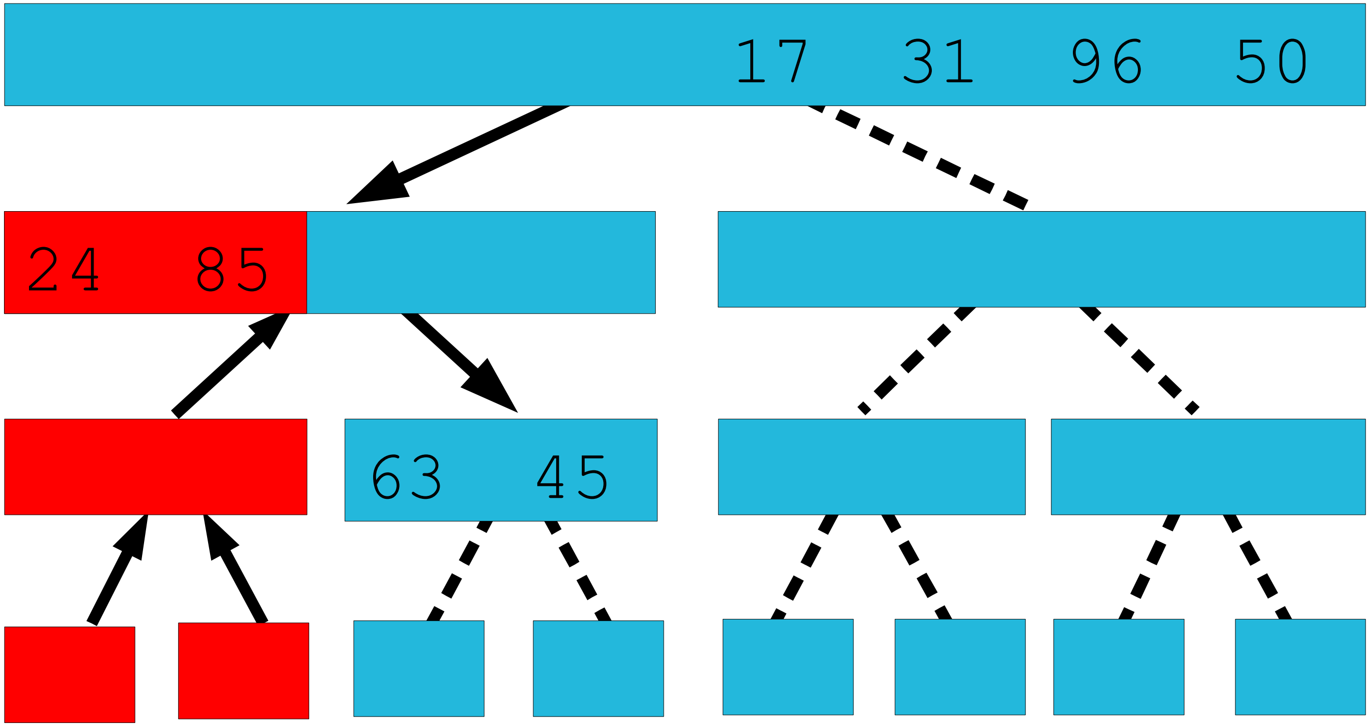
MergeSort Example/8



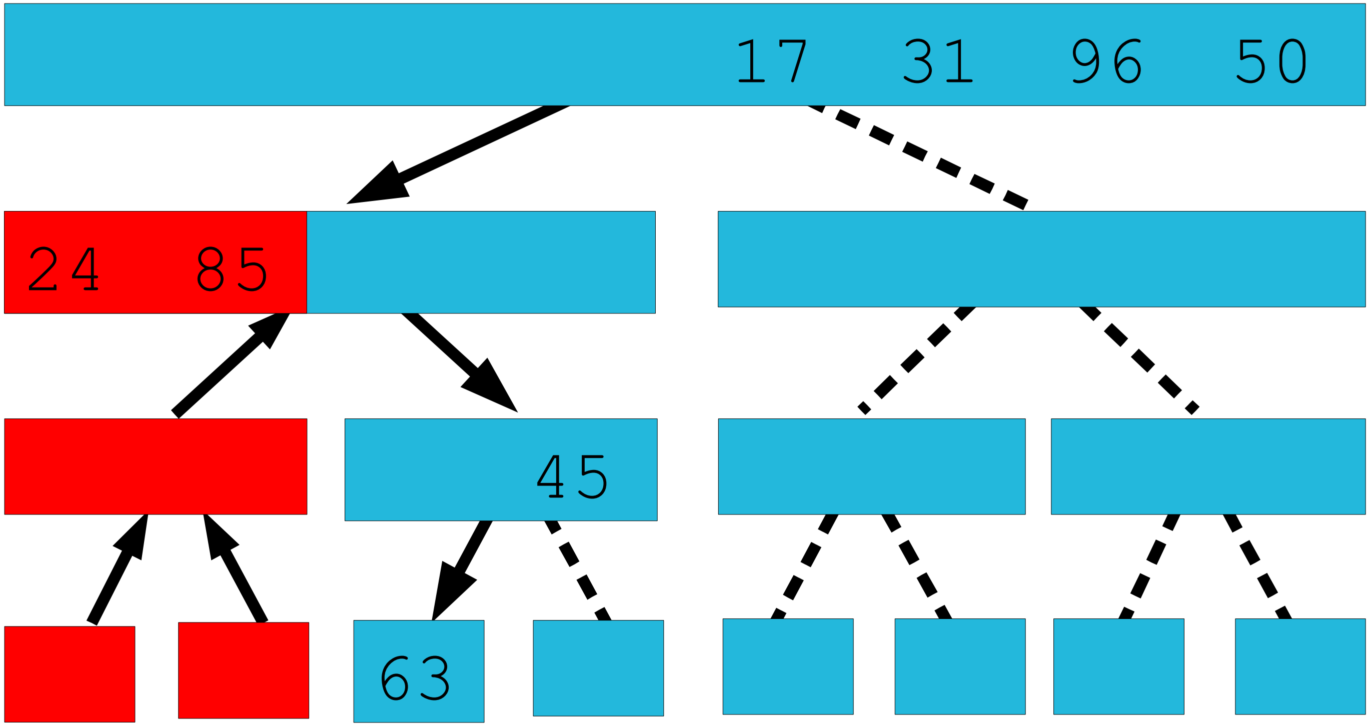
MergeSort Example/9



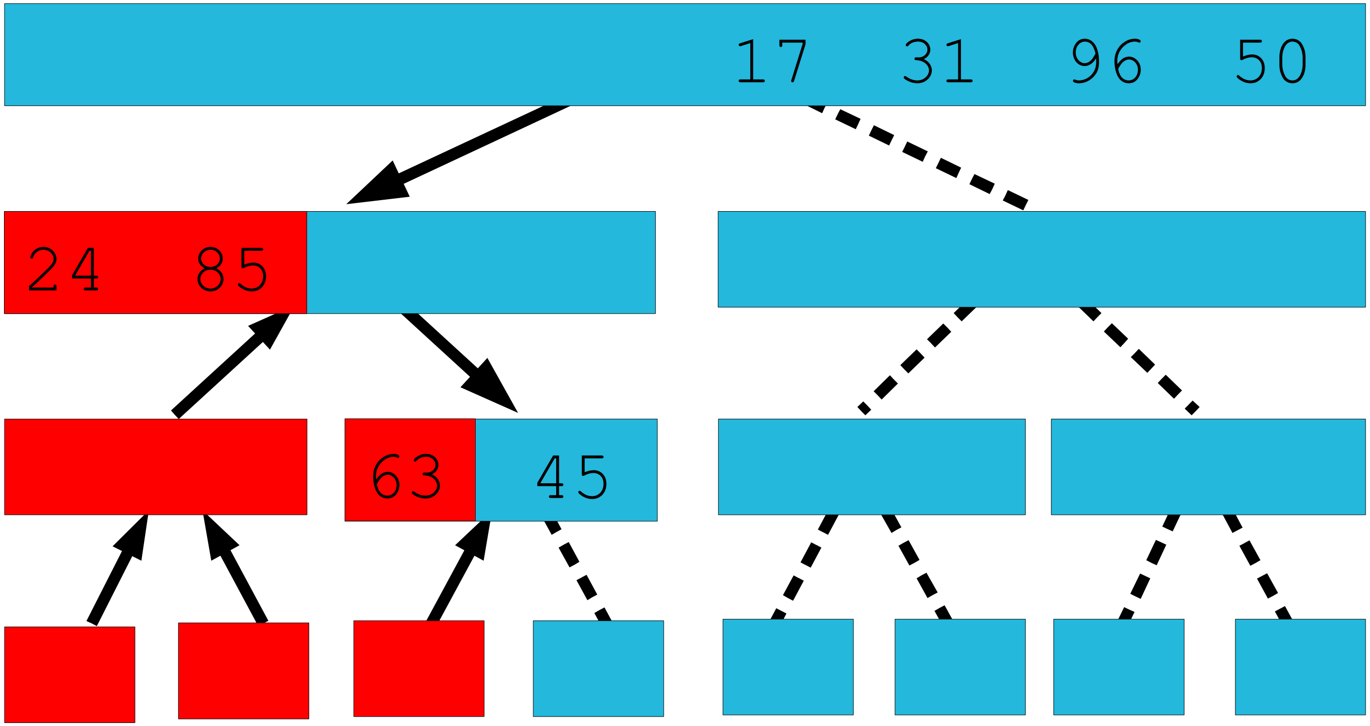
MergeSort Example/10



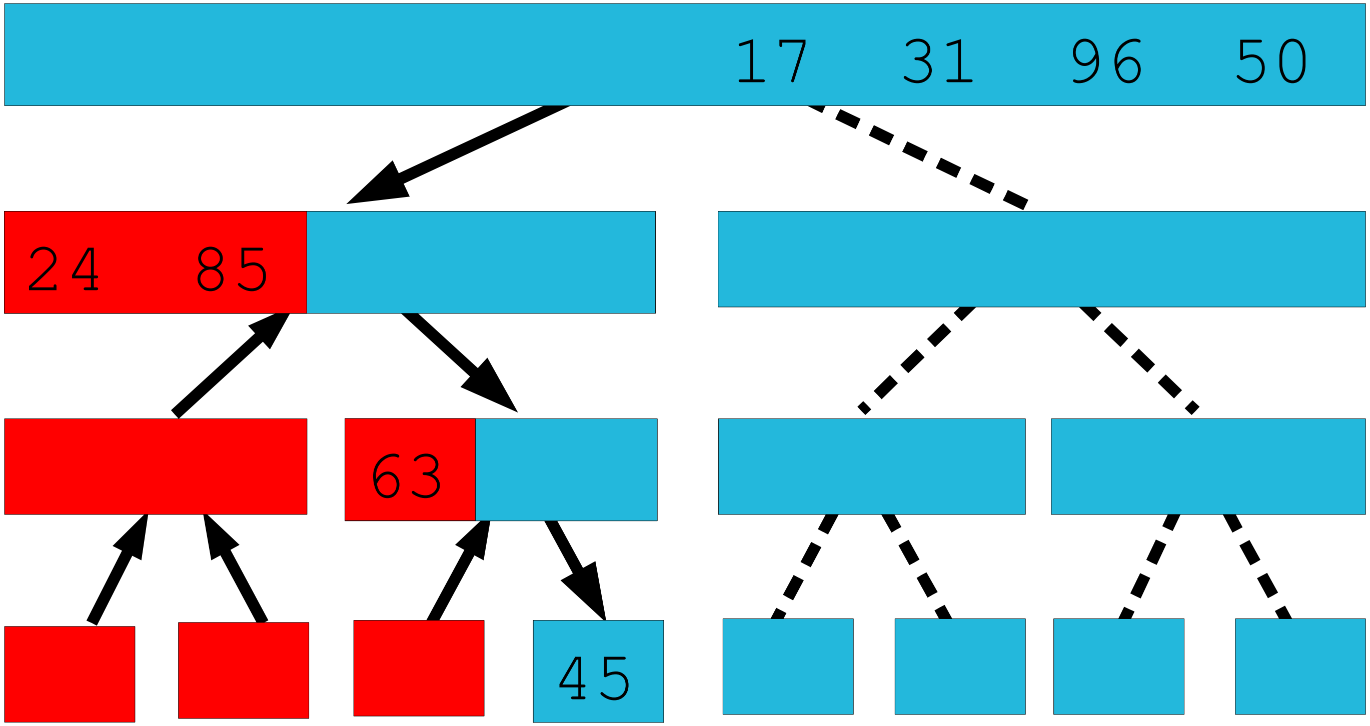
MergeSort Example/11



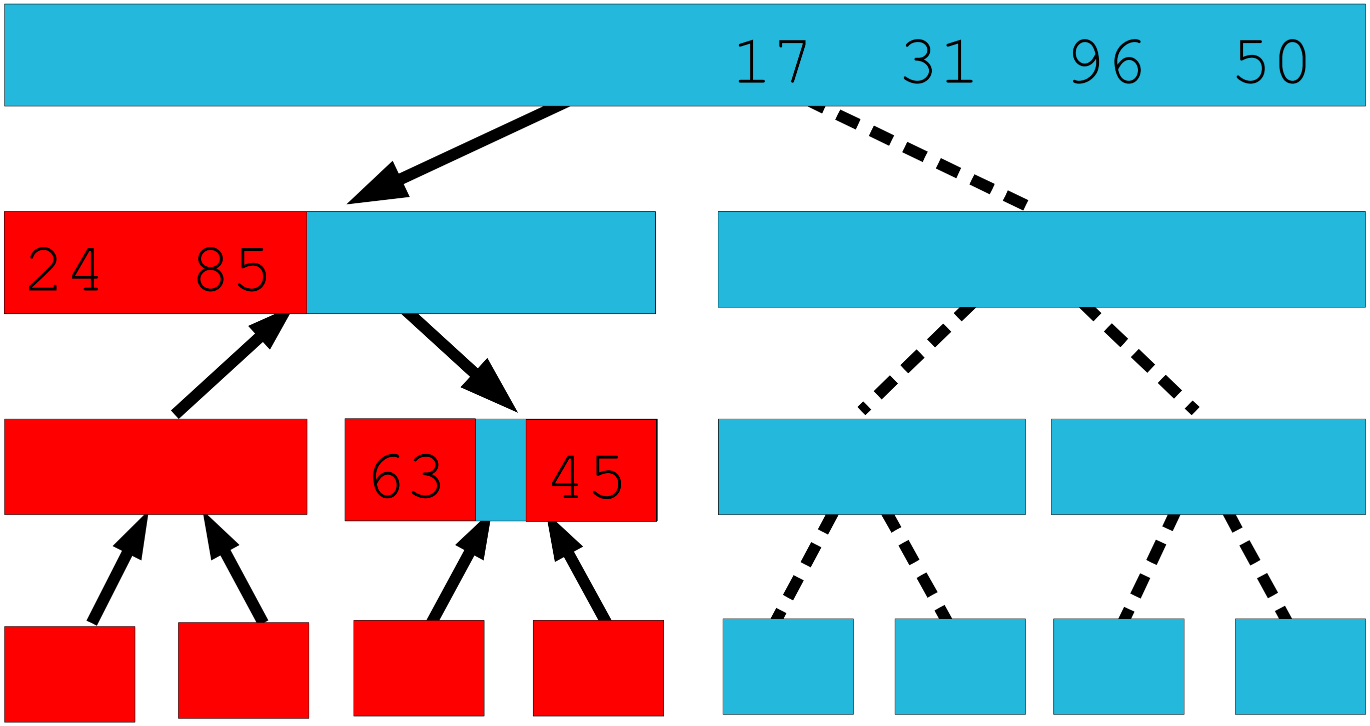
MergeSort Example/12



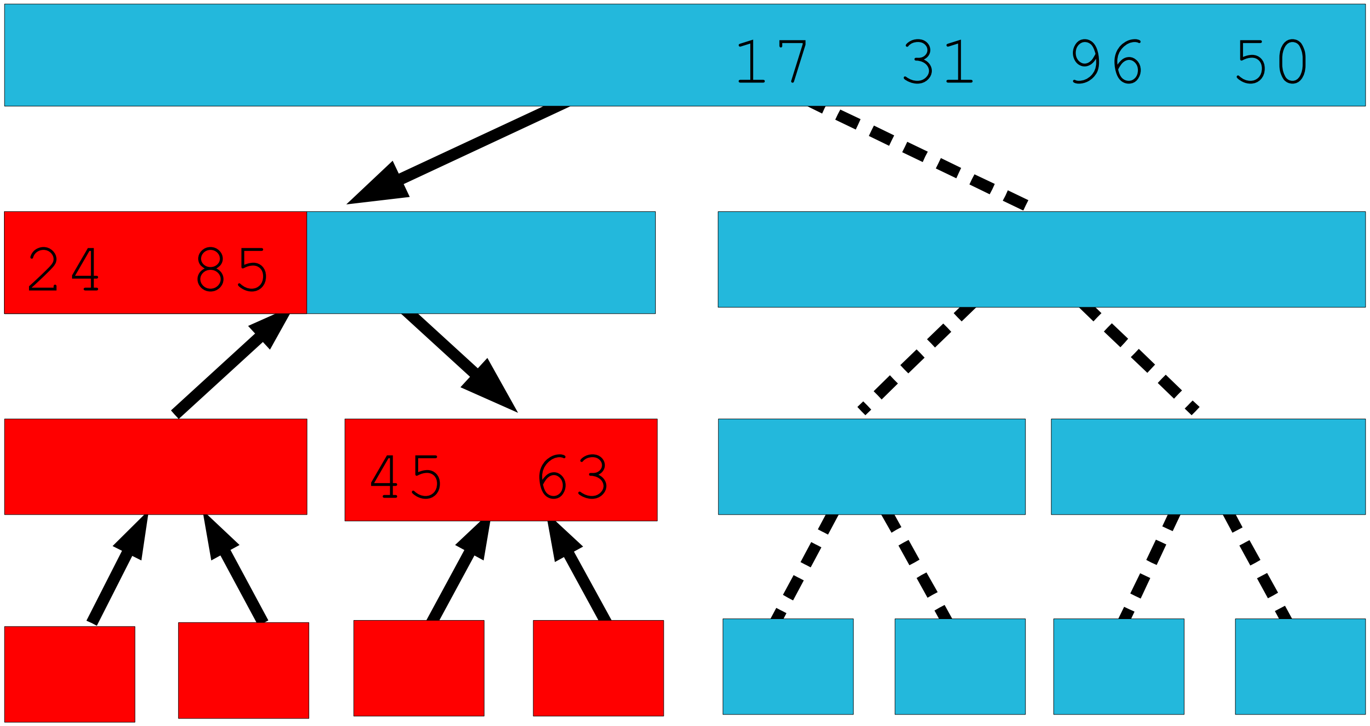
MergeSort Example/13



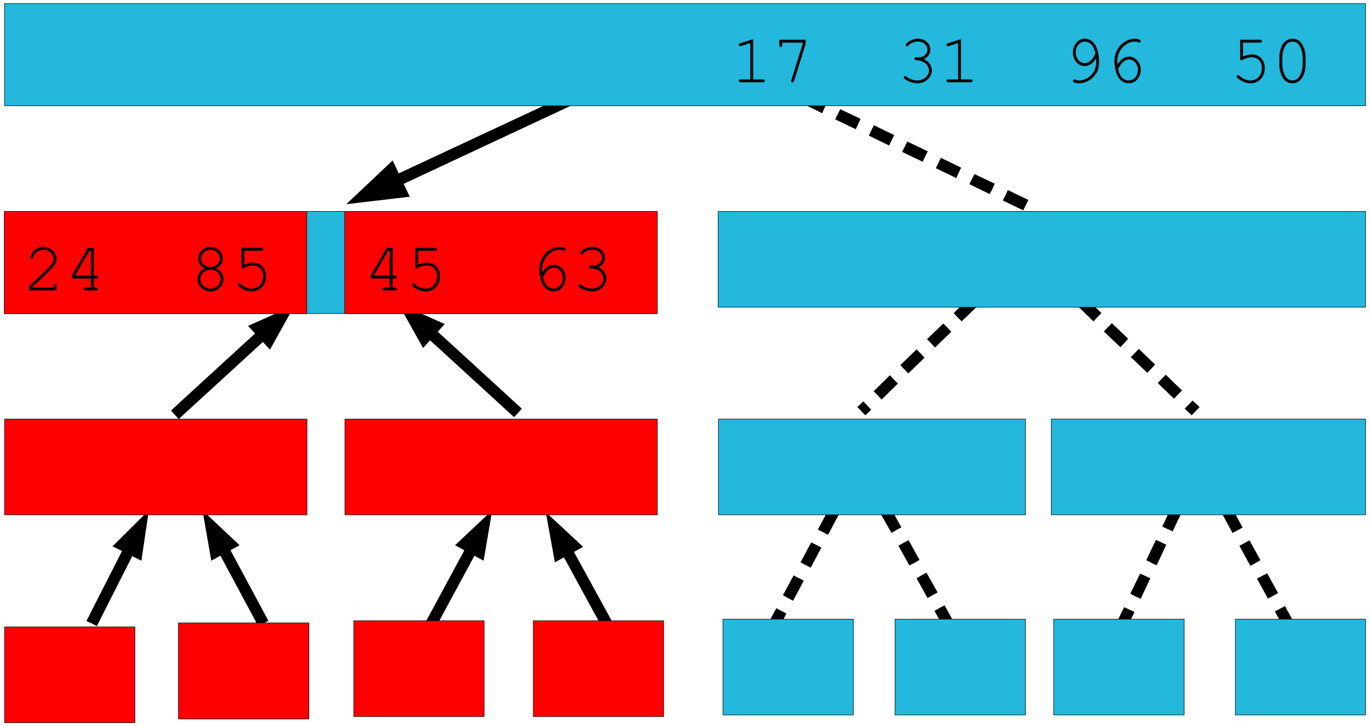
MergeSort Example/14



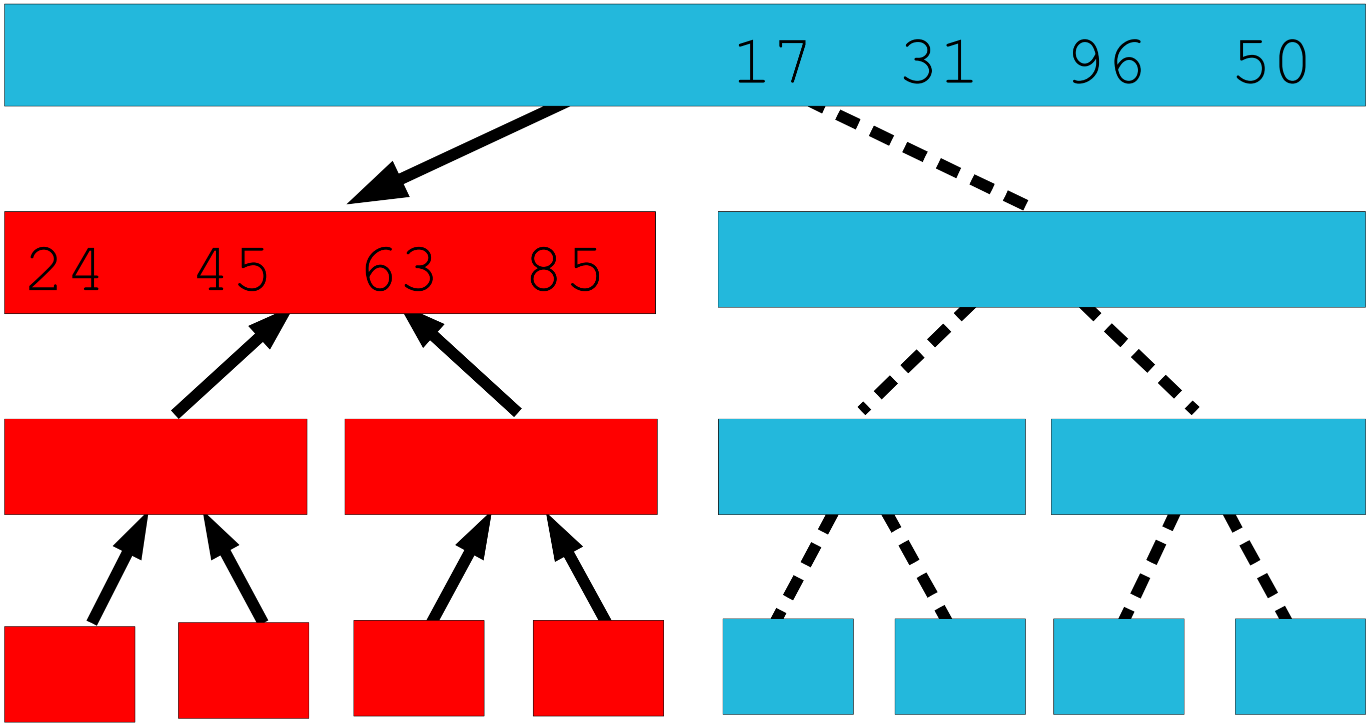
MergeSort Example/15



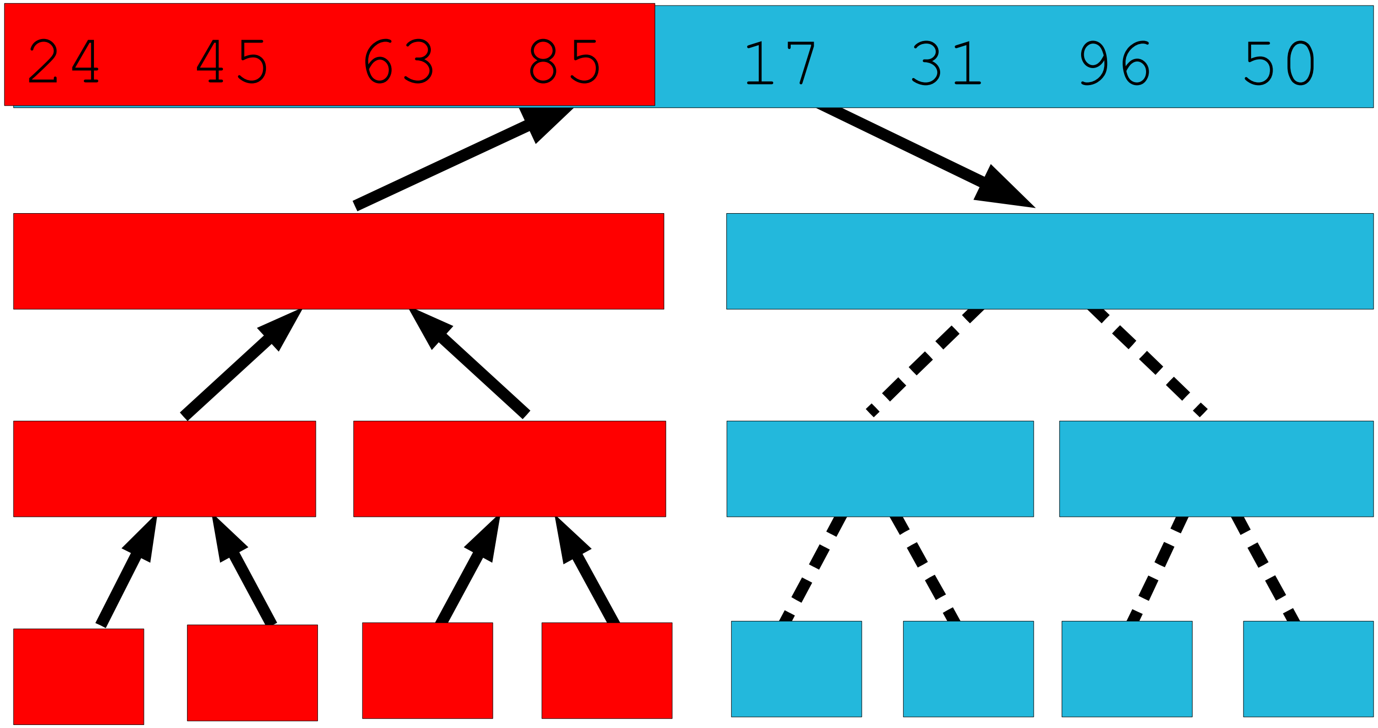
MergeSort Example/16



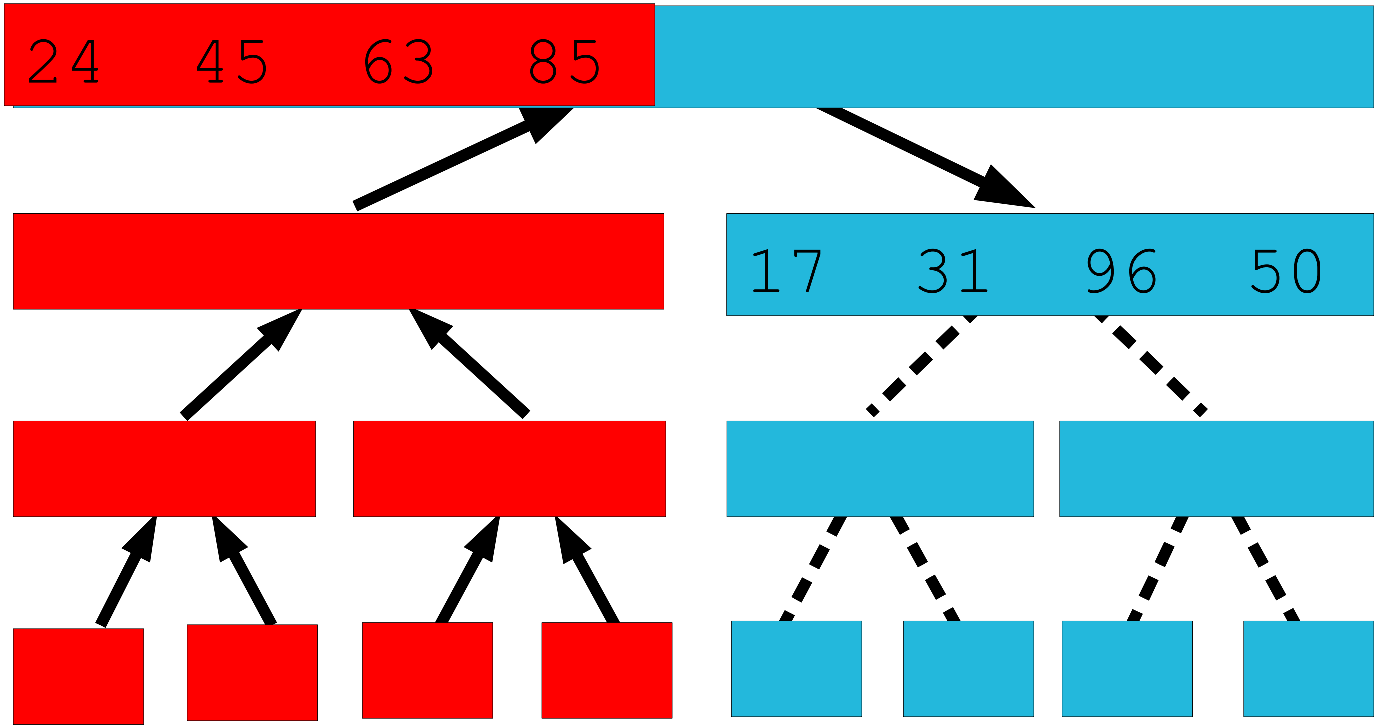
MergeSort Example/17



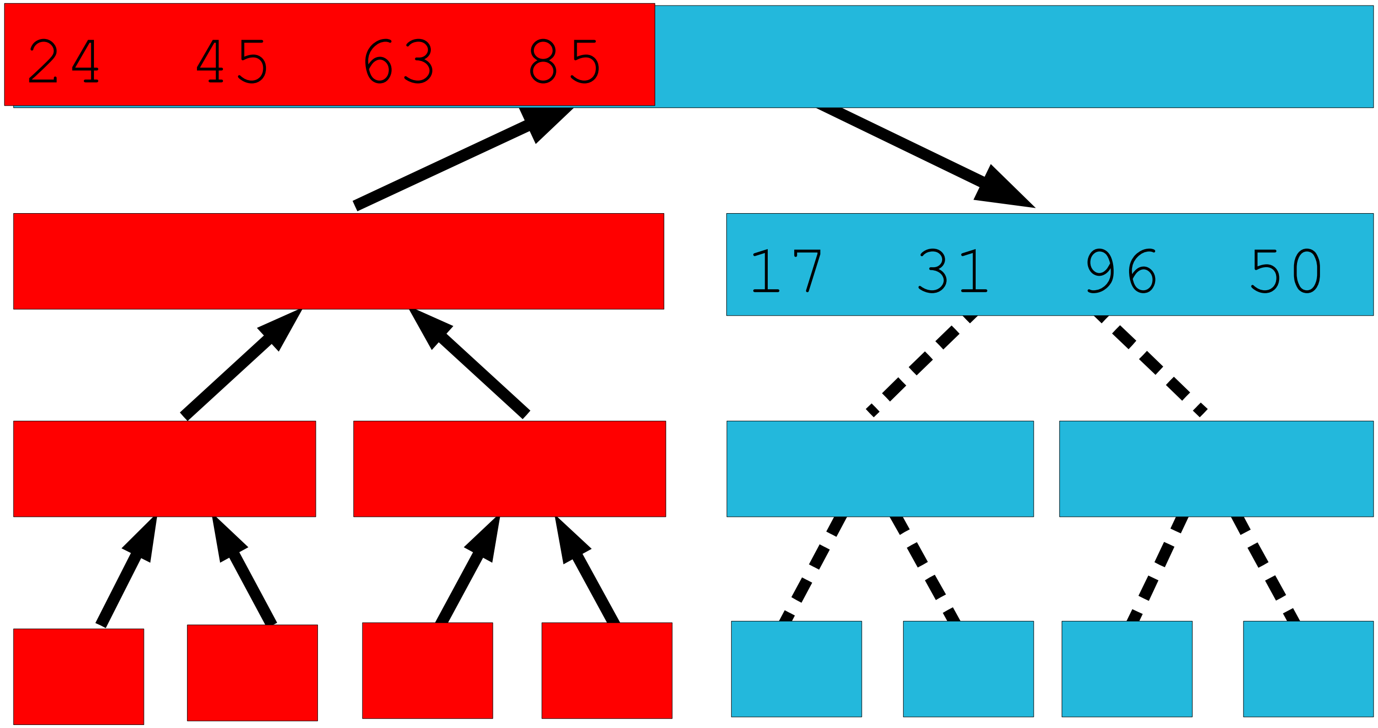
MergeSort Example/18



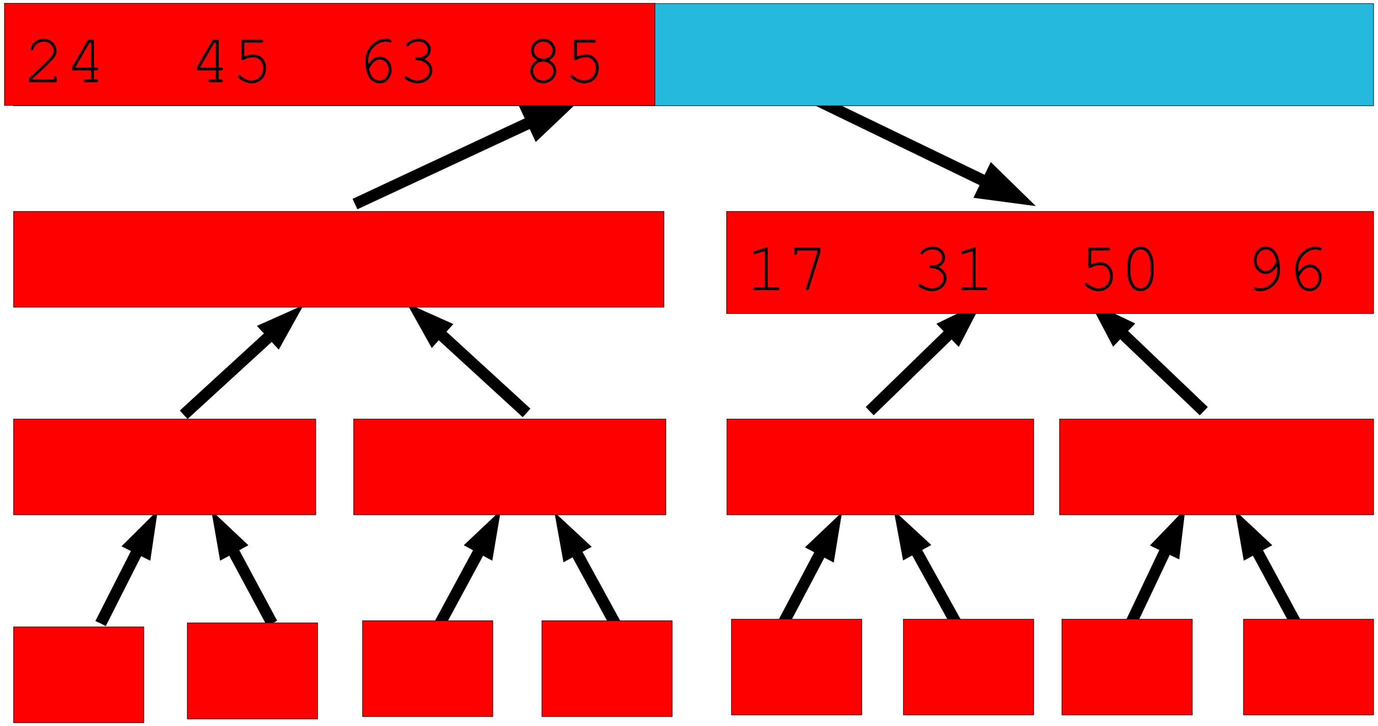
MergeSort Example/19



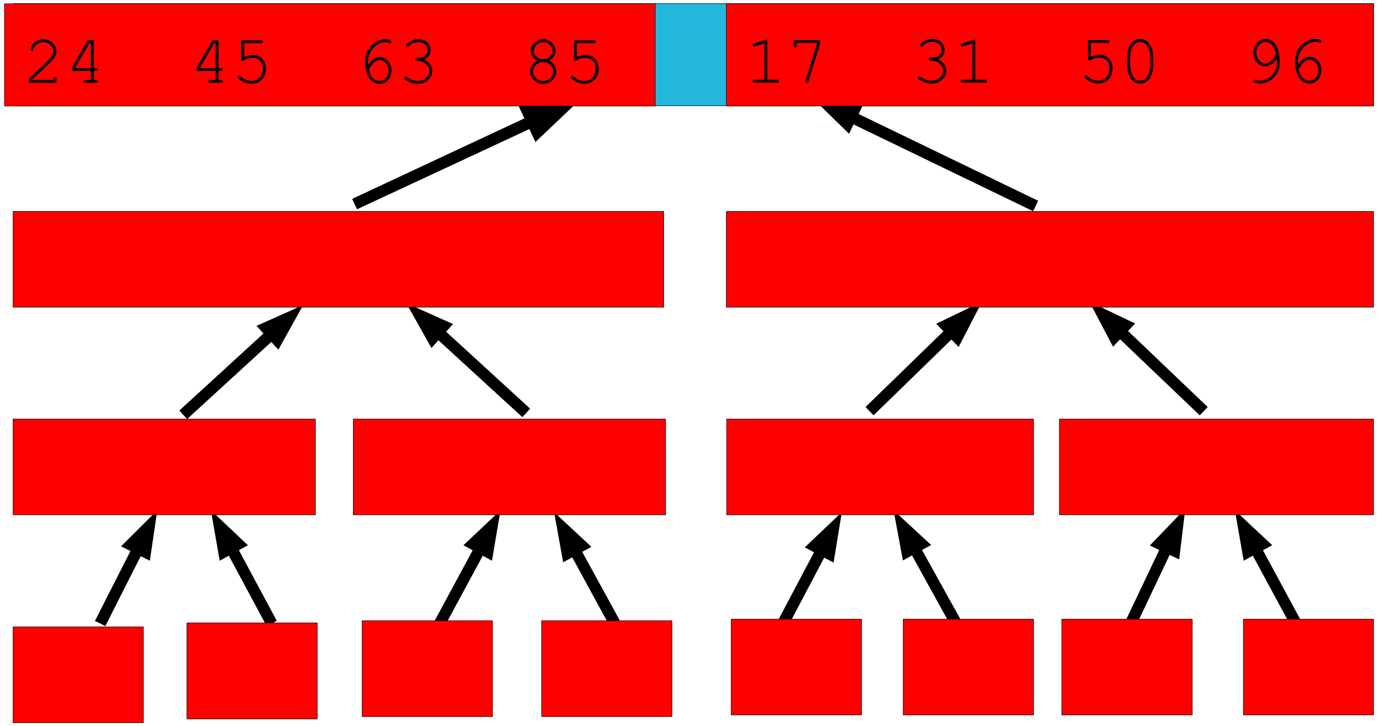
MergeSort Example/19



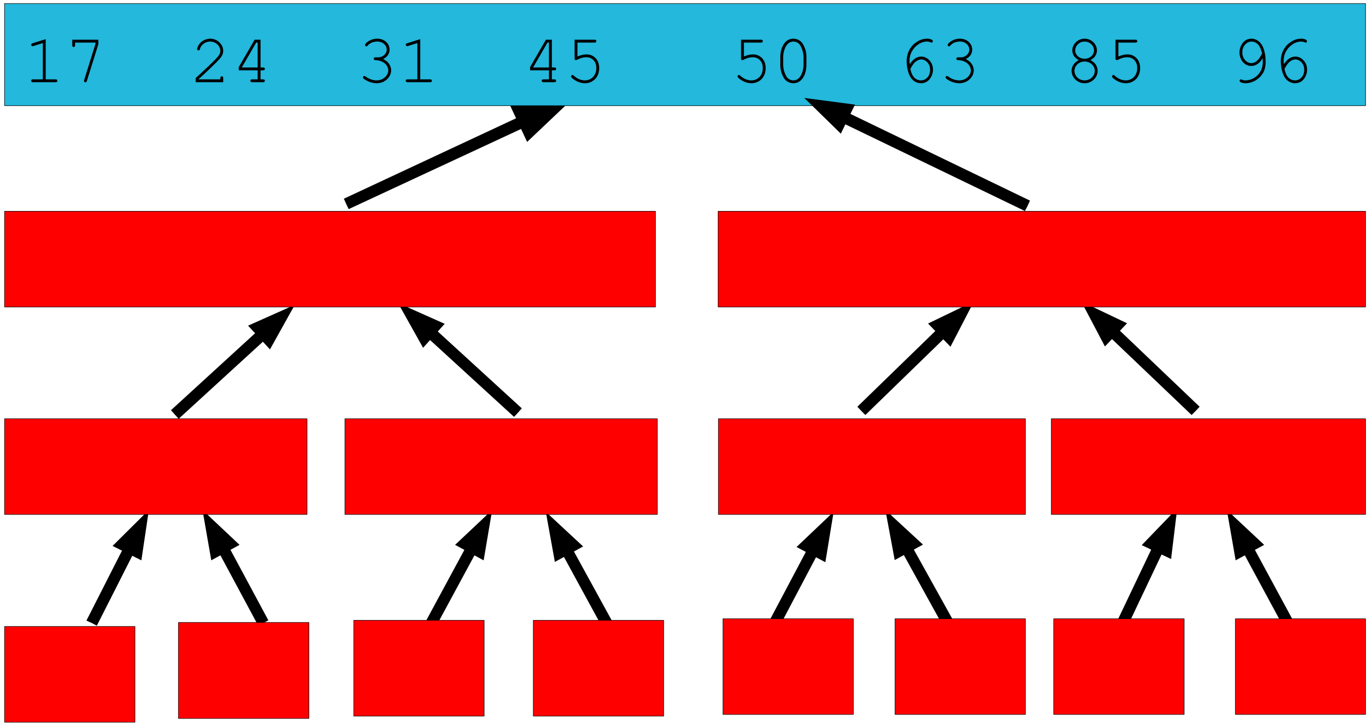
MergeSort Example/20



MergeSort Example/21

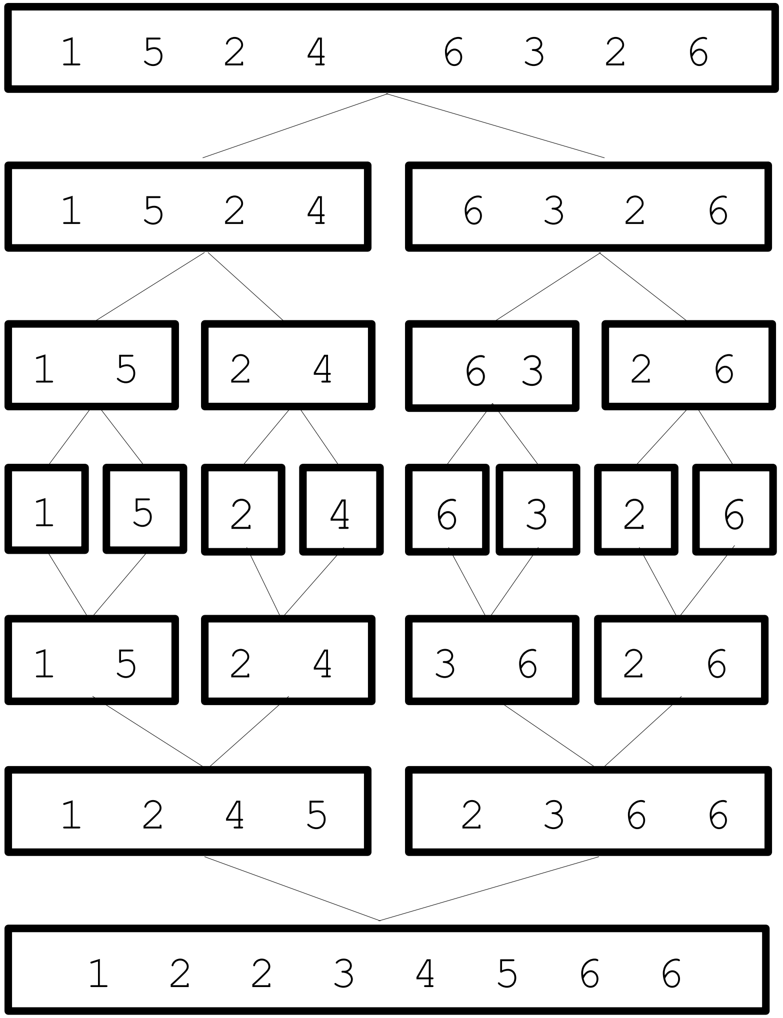


MergeSort Example/22



Merge Sort Summarized

- To sort *n* numbers
  - if n=1 done.
  - recursively sort 2 lists of $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements, elements, respectively.
  - merge 2 sorted lists of lengths n/2 in time ό(*n*).
- Strategy
  - break problem into similar (smaller) subproblems
  - recursively solve subproblems
  - combine solutions to answer



Running Time of MergeSort

The running time of a recursive procedure can be expressed as a recurrence:

$$T(n) = \left\{ \begin{array}{cc} \mathit{solving\,trivial\ problem} & \mathit{if}\ n = 1 \\ \mathit{NumPieces} * T(n / \mathit{SubProbFactor}) + \mathit{divide} + \mathit{combine} & \mathit{if}\ n > 1 \end{array} \right\}$$

$$T(n) = \left\{ \begin{array}{cc} \Theta(1) & \mathit{if}\, n = 1 \\ 2T(n/2) + \Theta(n) & \mathit{if}\, n > 1 \end{array} \right\}$$

Repeated Substitution Method

The running time of merge sort (assume $n=2^k$).

$$T(n)=\begin{cases}\Theta(1) & if\ n=1\\ 2\mathrm{T}(n/2)+\Theta(n) & if\ n>1\end{cases}$$

$T(n) = 2T(n/2) + n$ substitute

$= 2(2T(n/4) + n/2) + n$ expand

$= 2^2T(n/4) + 2n$ substitute

$= 2^2(2T(n/8) + n/4) + 2n$ expand

$= 2^3T(n/8) + 3n$ observe pattern

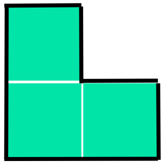
$T(n) = 2^iT(n/2^i) + i\,n$

$= 2^{\log n}T(n/n) + n \log n$

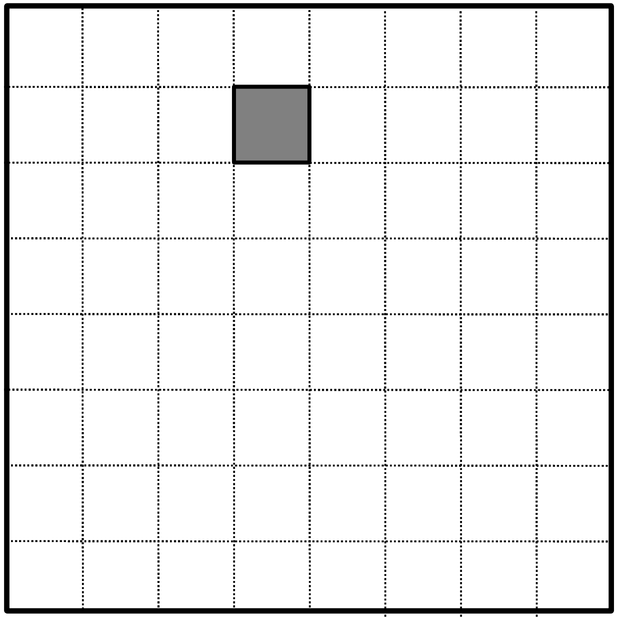
$= n + n \log n$

Tiling

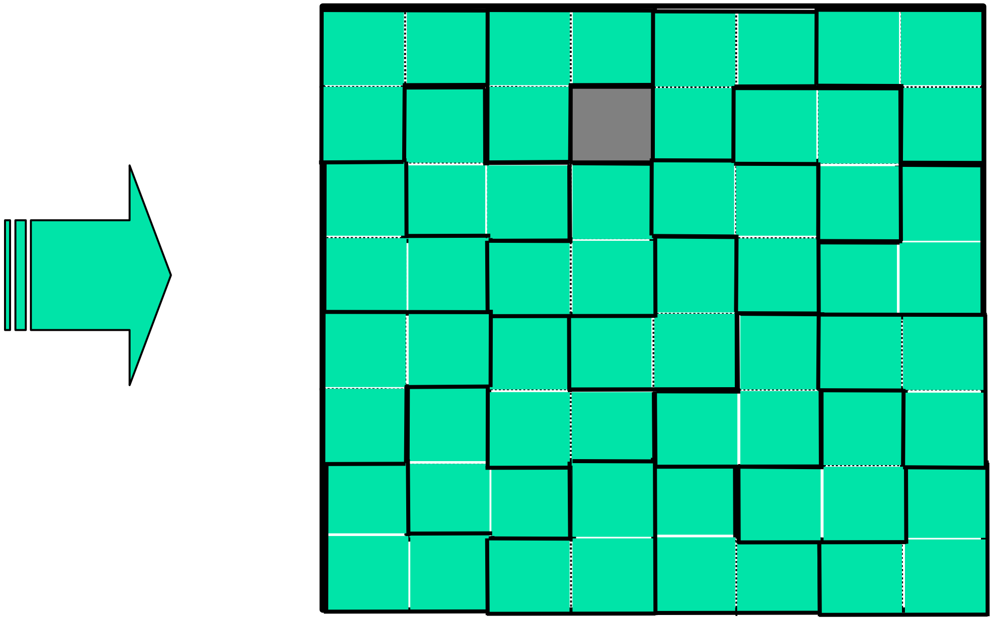
A tromino tile:



A 2^nx2^n board with a hole:

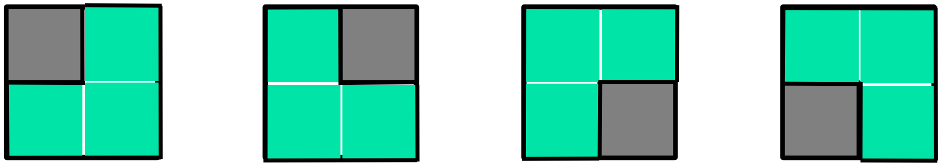


A tiling of the board with trominos:



Tiling: Trivial Case (n = 1)

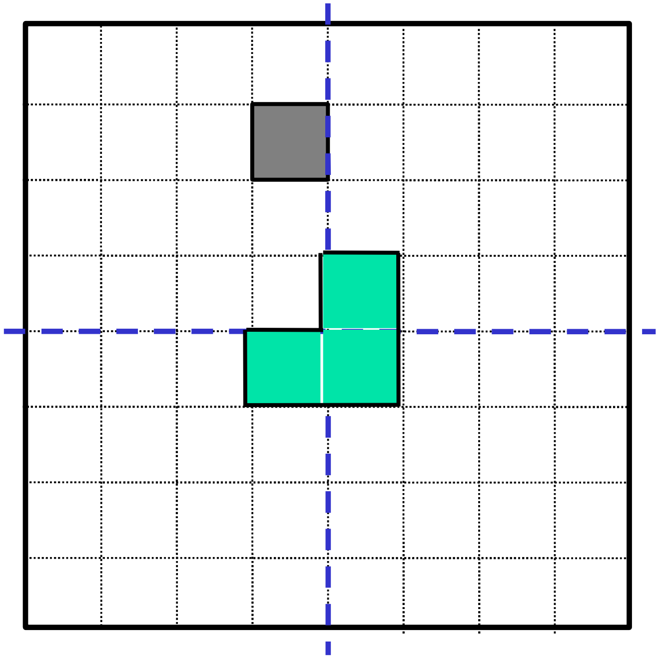
Trivial case (n = 1): tiling a 2x2 board with a hole:



Idea: reduce the size of the original problem, so that we eventually get to the 2x2 boards, which we know how to solve.

Tiling: Dividing the Problem/2

Idea: insert one tromino at the center to "cover" three holes in each of the three smaller boards



- Now we have four boards with holes of the size 2^{n-1}x2^{n-1}.
- Keep doing this division, until we get the 2x2 boards with holes – we know how to tile those.

Tiling: Algorithm

```
INPUT:   n – the board size ($2^n$x$2^n$ board),
         L – location of the hole.
OUTPUT: tiling of the board

Tile(n, L)
  if n = 1 then //Trivial case
    Tile with one tromino
    return
  Divide the board into four equal-sized boards
  Place one tromino at the center to cover 3 additional
      holes
  Let L1, L2, L3, L4 be the positions of the 4 holes
  Tile(n-1, L1)
  Tile(n-1, L2)
  Tile(n-1, L3)
  Tile(n-1, L4)
```

Tiling: Divide-and-Conquer

Tiling is a divide-and-conquer algorithm:

The problem is trivial if the board is 2x2, else:

Divide the board into four smaller boards
(introduce holes at the corners of the
three smaller boards
to make them look like original problems).

Conquer using the same algorithm recursively

Combine by placing a single tromino
in the center to cover the three new holes.

Karatsuba Multiplication

Multiplying two *n*-digit (or *n*-bit) numbers costs n^2 digit multiplications, using a straightforward procedure.

Observation:

$23*14 = (2\times10^1+3)*(1\times10^1+4) =$

$= (2*1)10^2 + (3*1 + 2*4)10^1 + (3*4)$

To save one multiplication we use a trick:

$(3*1 + 2*4) = (2+3)*(1+4) - (2*1) - (3*4)$

Original by S. Saltenis, Aalborg

Karatsuba Multiplication/2

To multiply *a* and *b*, which are *n*-digit numbers, we use a divide and conquer algorithm. We split them in half:

$$a = a_1 \times 10^{n/2} + a_0 \quad \text{and } b = b_1 \times 10^{n/2} + b_0$$

Then:

$$a * b = (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$

Use a trick to save a multiplication:

$$(a_1 * b_0 + a_0 * b_1) = (a_1 + a_0) * (b_1 + b_0) - (a_1 * b_1) - (a_0 * b_0)$$

Karatsuba Multiplication/3

The number of single-digit multiplications performed by the algorithm can be described by a recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) & \text{if } n > 1 \end{cases}$$

Recurrences

- Running times of algorithms with recursive calls can be described using recurrences.
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- For divide and conquer algorithms:

$$T(n) = \begin{cases} \text{solving trivial problem} & \text{if } n = 1 \\ \text{NumPieces} * T(n/\text{SubProbFactor}) + \text{divide} + \text{combine} & \text{if } n > 1 \end{cases}$$

- Example: Merge Sort

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

Solving Recurrences

- Repeated (backward) substitution method
  - Expanding the recurrence by substitution and noticing a pattern (this is not a strictly formal proof).
- Substitution method
  - guessing the solutions
  - verifying the solution by mathematical induction
- Recursion trees
- Master method
  - templates for different classes of recurrences

Repeated Substitution

Let’s find the running time of merge sort (assume $n=2^b$).

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + n \quad \text{substitute} \\ &= 2(2T(n/4) + n/2) + n \quad \text{expand} \\ &= 2^2T(n/4) + 2n \quad \text{substitute} \\ &= 2^2(2T(n/8) + n/4) + 2n \quad \text{expand} \\ &= 2^3T(n/8) + 3n \quad \text{observe pattern} \end{aligned}$$

Repeated Substitution/2

From $T(n) = 2^3 T(n/8) + 3n$

we get $T(n) = 2^i T(n/2^i) + i\, n$

An upper bound for *i* is *log n:*

$$T(n) = 2^{\log n} T(n/n) + n \log n$$

$$T(n) = n + n \log n$$

Repeated Substitution Method

The procedure is straightforward:

- Substitute, Expand, Substitute, Expand, …
- Observe a pattern and determine the expression after the *i*-th substitution.
- Find out what the highest value of *i* (number of iterations, e.g., *log n*) should be to get to the base case of the recurrence (e.g., *T(1)*).
- Insert the value of *T(1)* and the expression of *i* into your expression.

Analysis of Merge Sort

- Let’s find a more exact running time of merge sort (assume n=2^b).

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

$T(n) = 2T(n/2) + 2n + 3$ substitute

$= 2(2T(n/4) + n + 3) + 2n + 3$ expand

$= 2^2T(n/4) + 4n + 2*3 + 3$ substitute

$= 2^2(2T(n/8) + n/2 + 3) + 4n + 2*3 + 3$ expand

$= 2^3T(n/2^3) + 2*3n + (2^{2+}2^{1+}2^0)*3$ observe pattern