Data Structures and Algorithms

Chapter 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, Chapter 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 one 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.

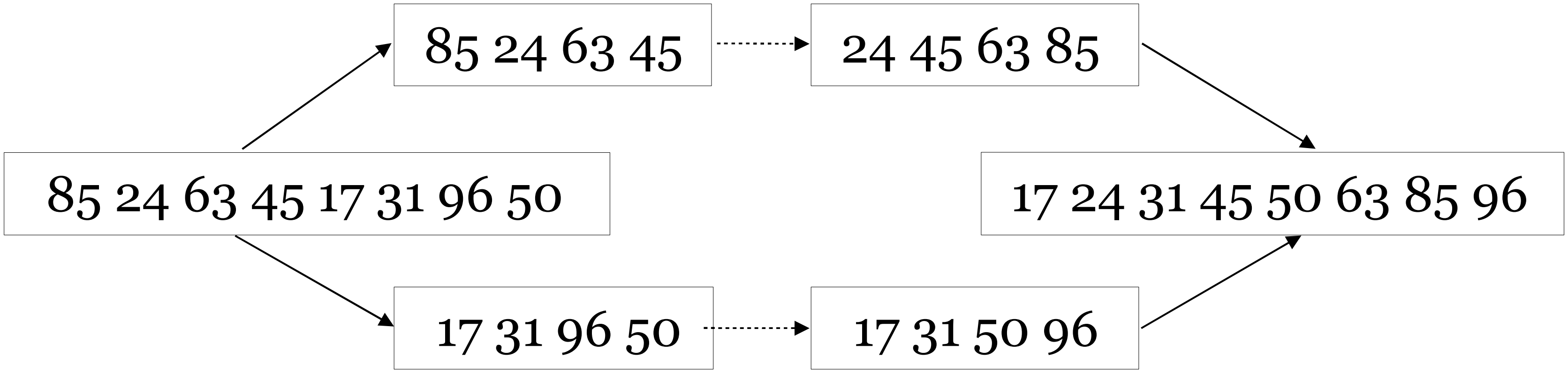
Picking a Decomposition

- 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 of the same kind as 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.



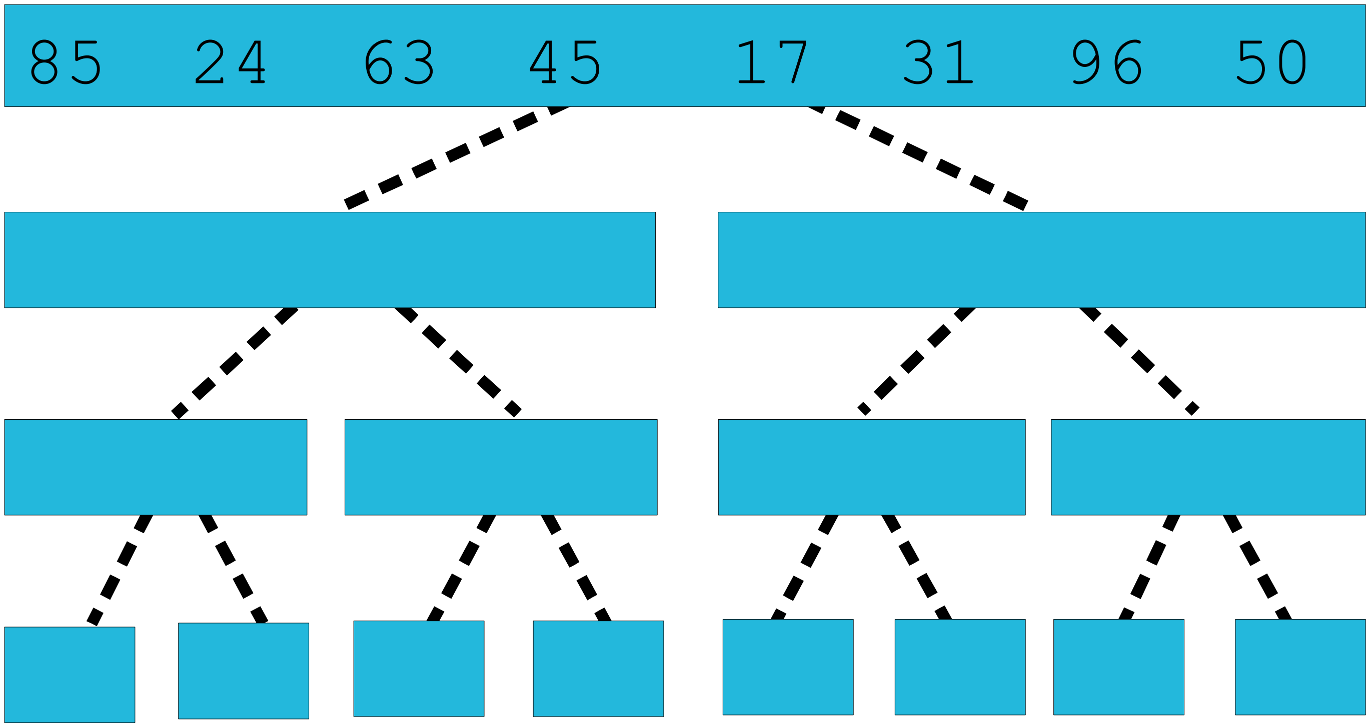
Merge Sort Algorithm

Divide: If segment S has at least two elements,
divide S into segments 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

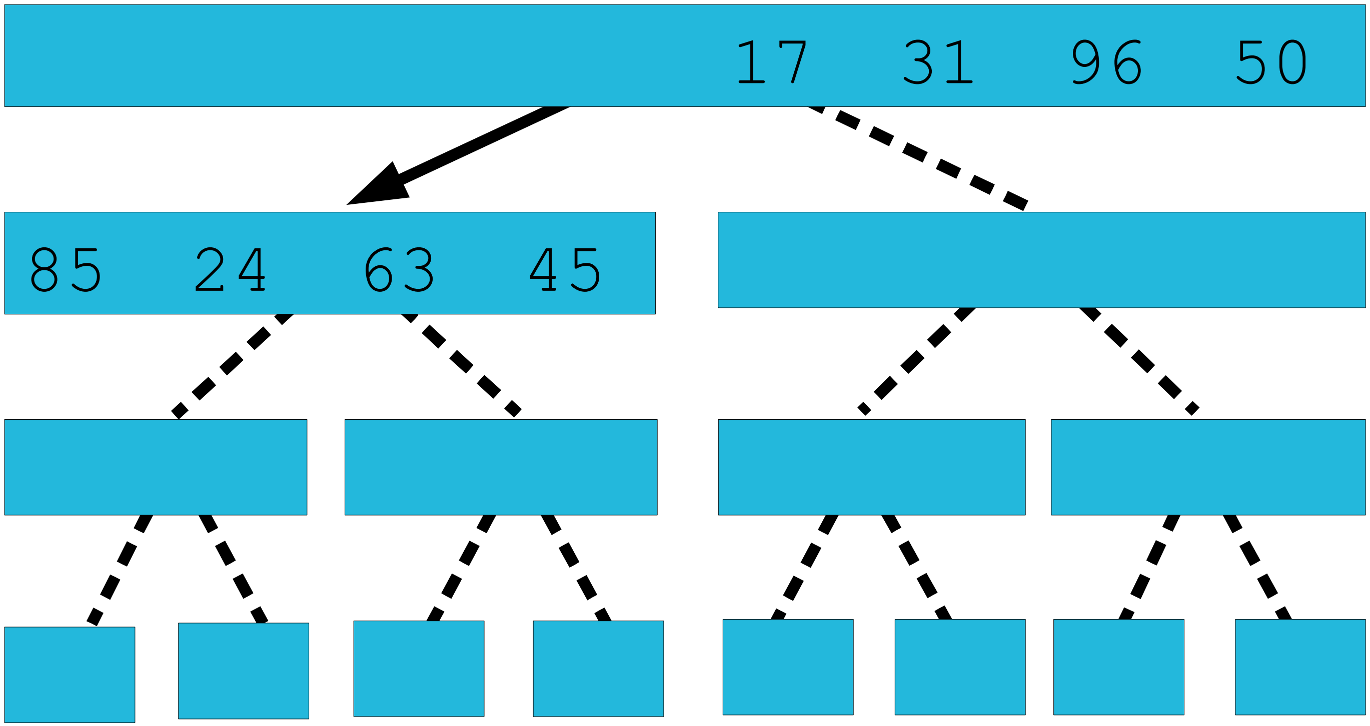
Conquer: Sort segments S_1 and S_2
using merge sort

Combine: Merge the sorted segments S_1 and S_2,
into one sorted auxiliary array,
and copy the auxiliary array back into segment S

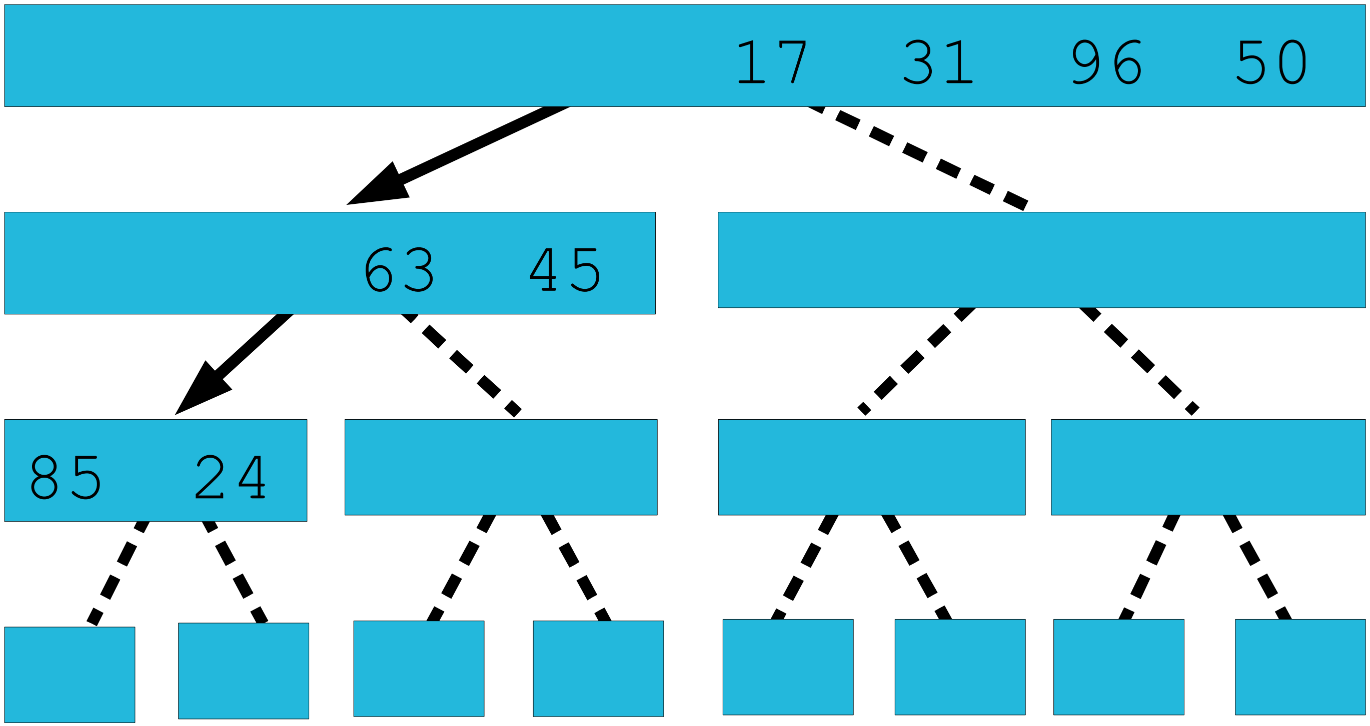
MergeSort Example/1



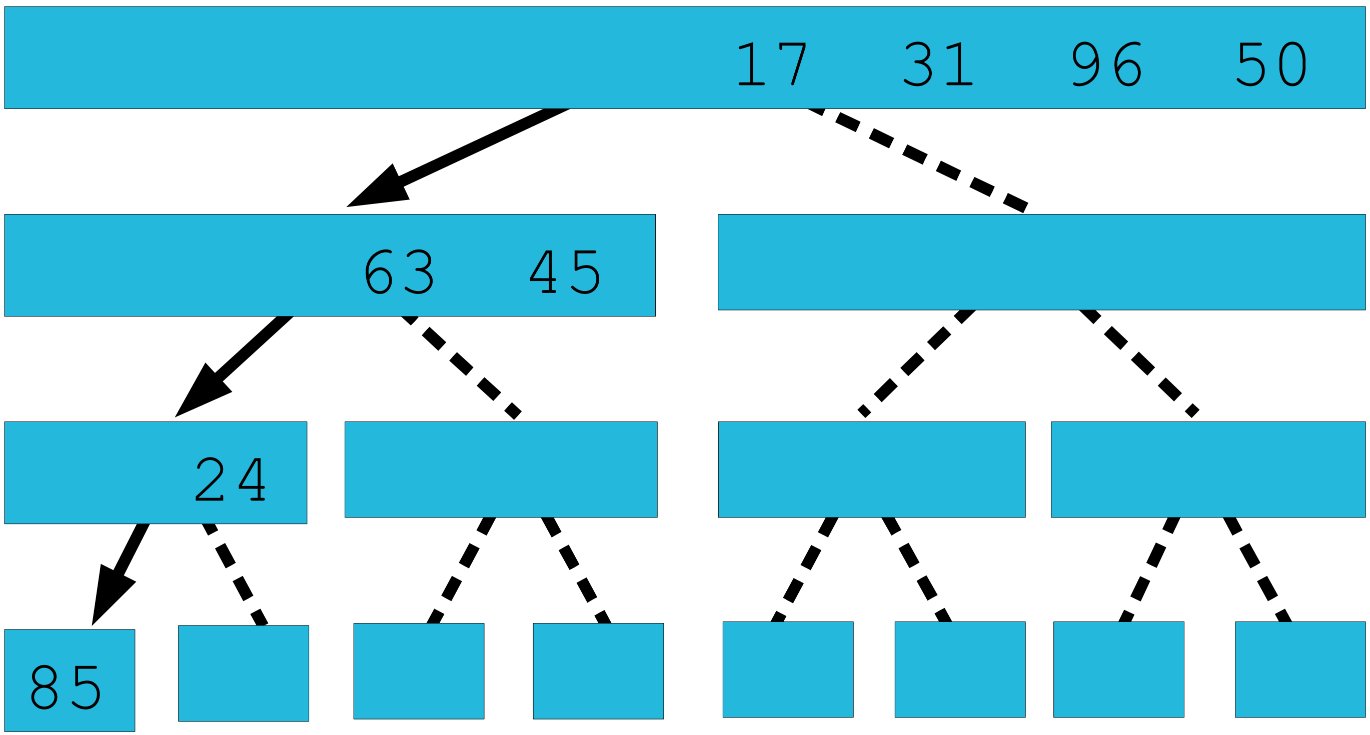
MergeSort Example/2



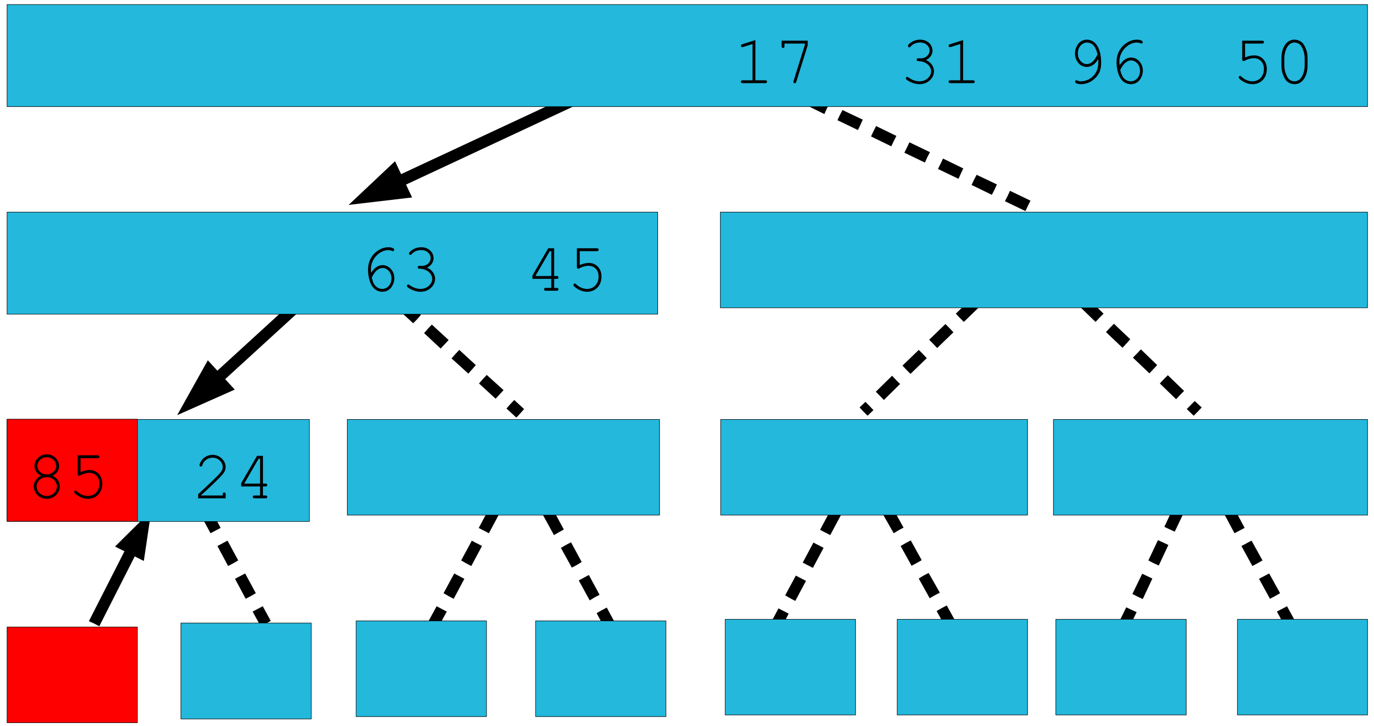
MergeSort Example/3



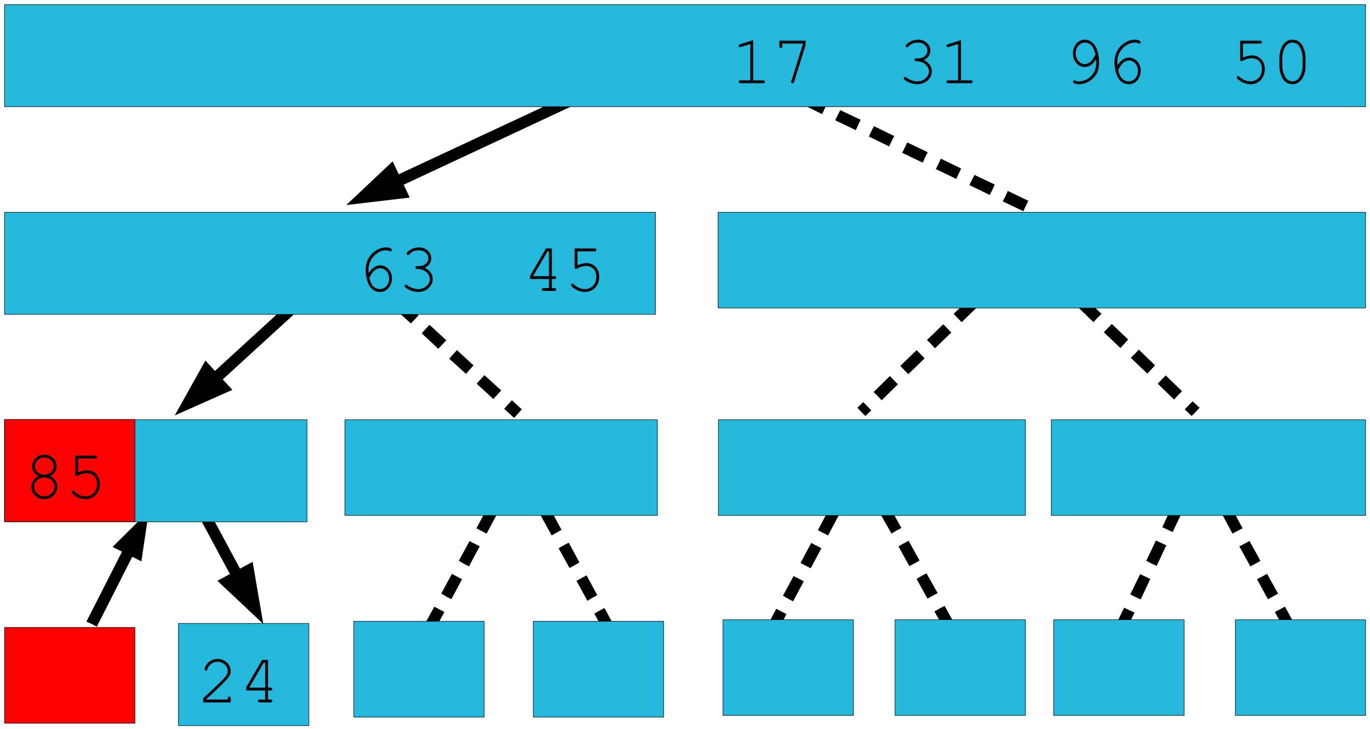
MergeSort Example/4



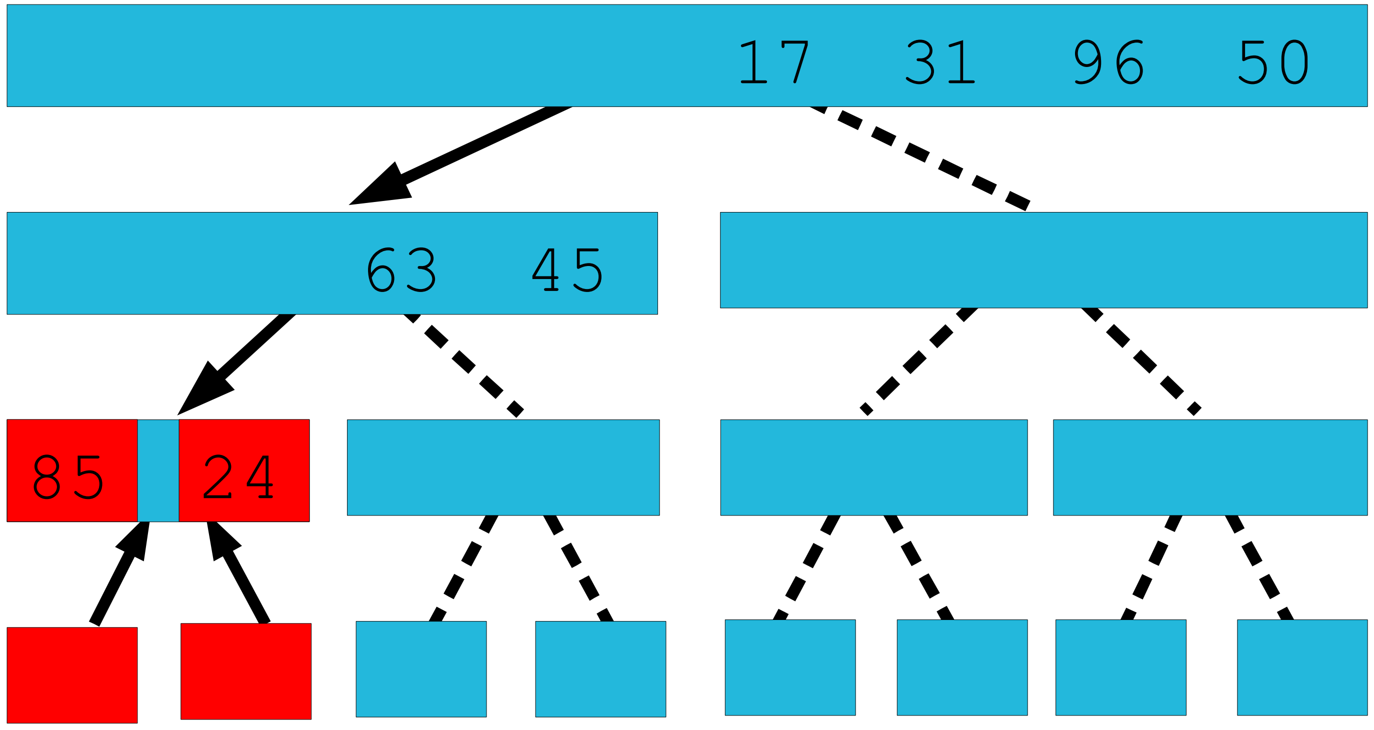
MergeSort Example/5



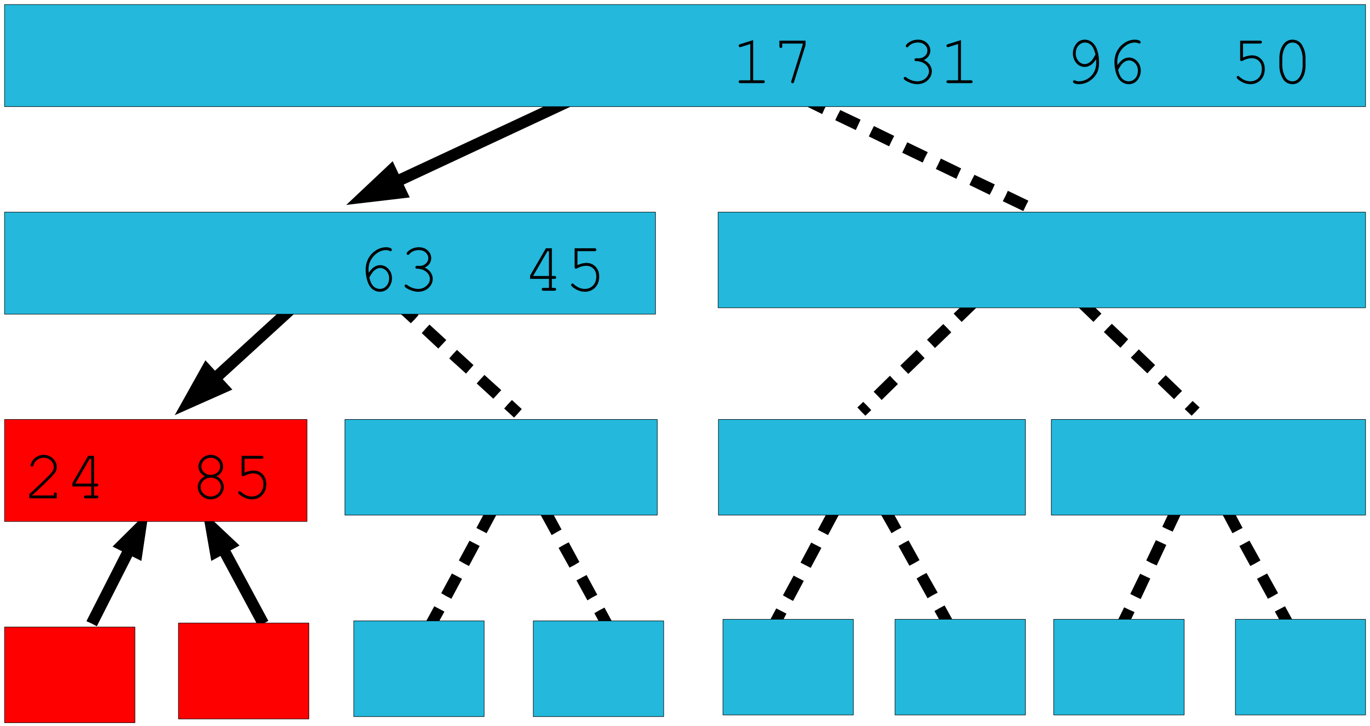
MergeSort Example/6



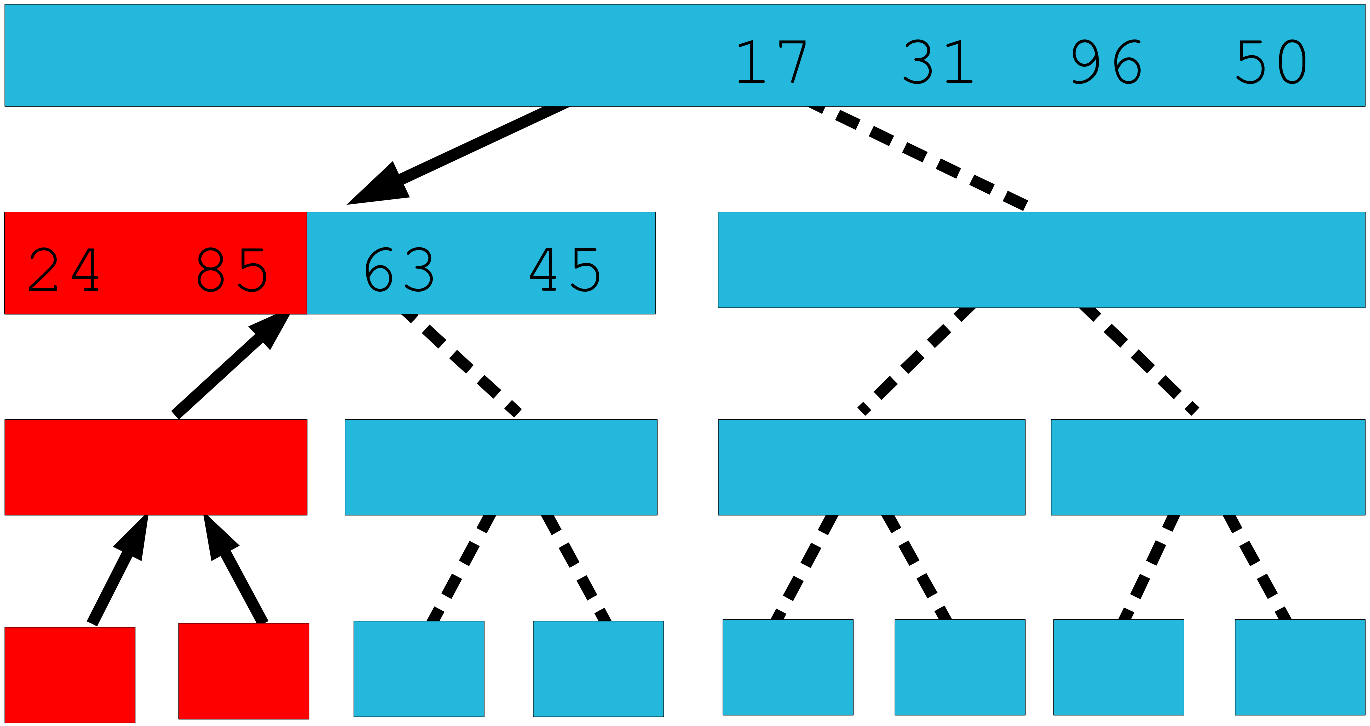
MergeSort Example/7



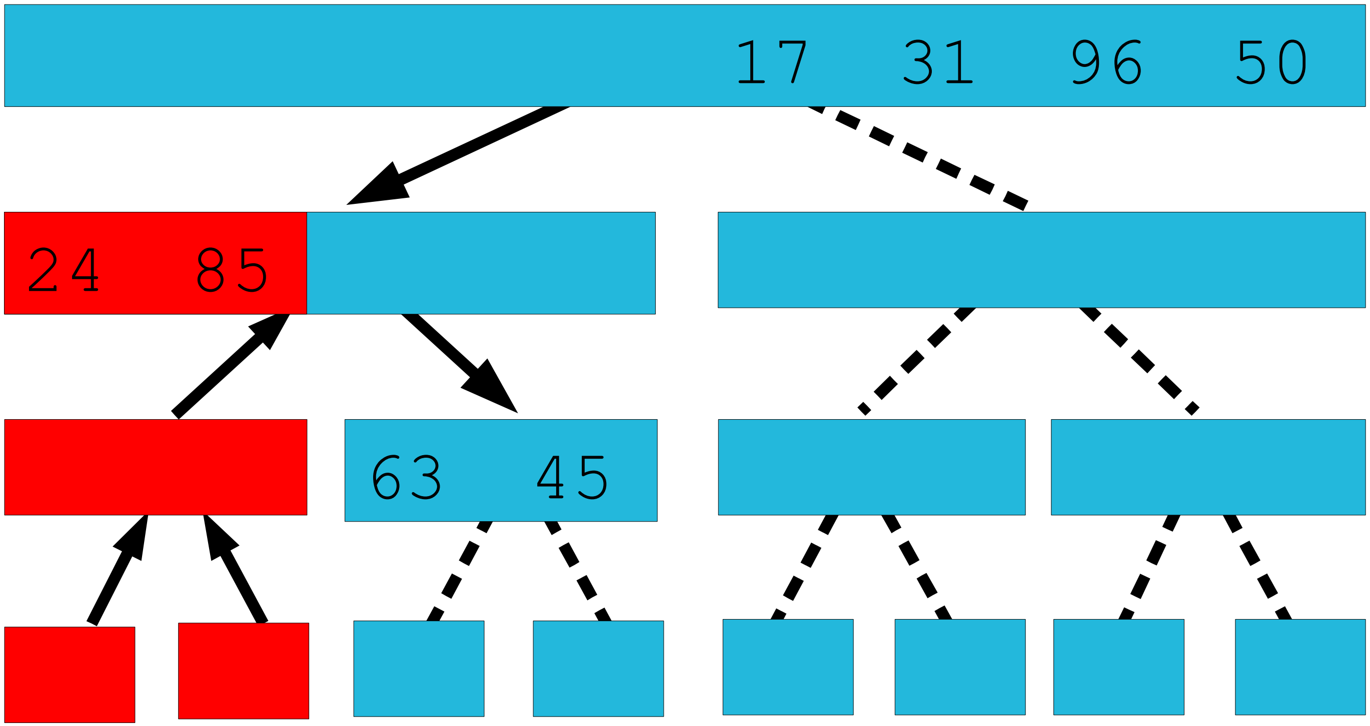
MergeSort Example/8



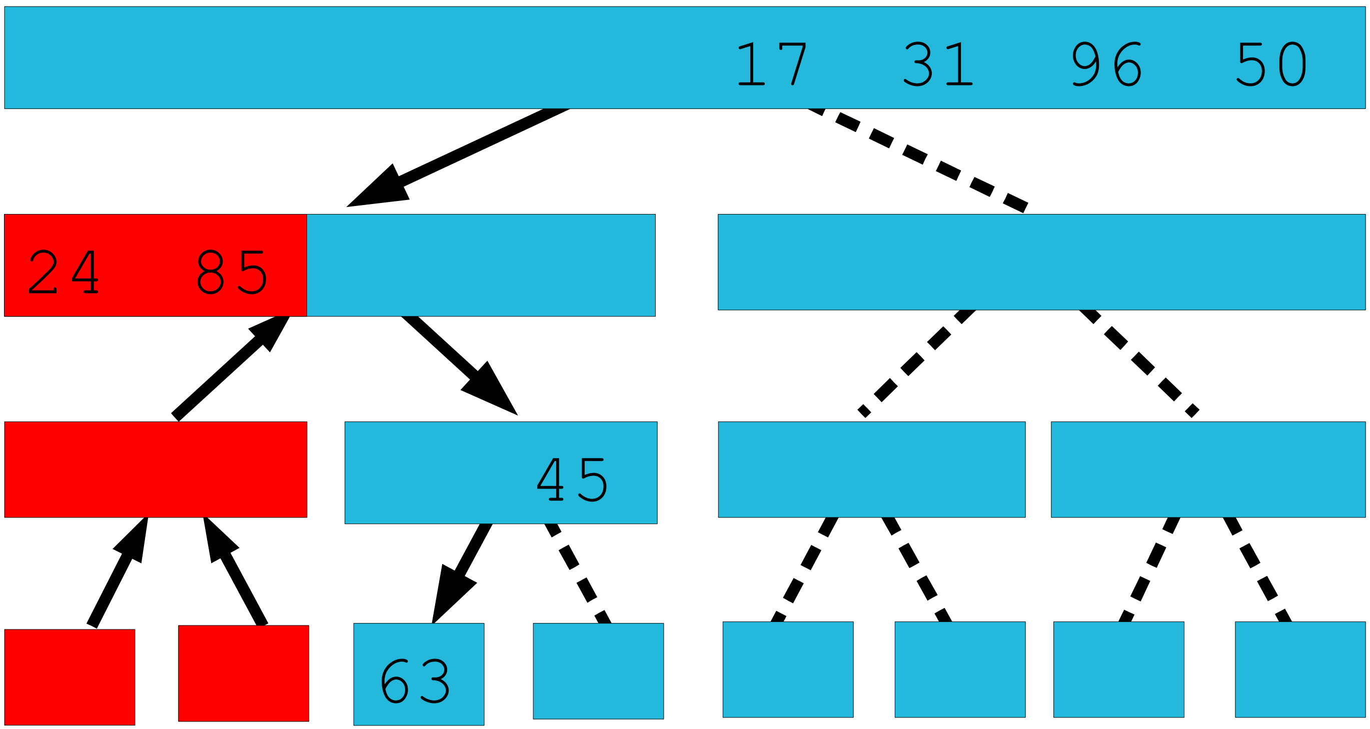
MergeSort Example/9



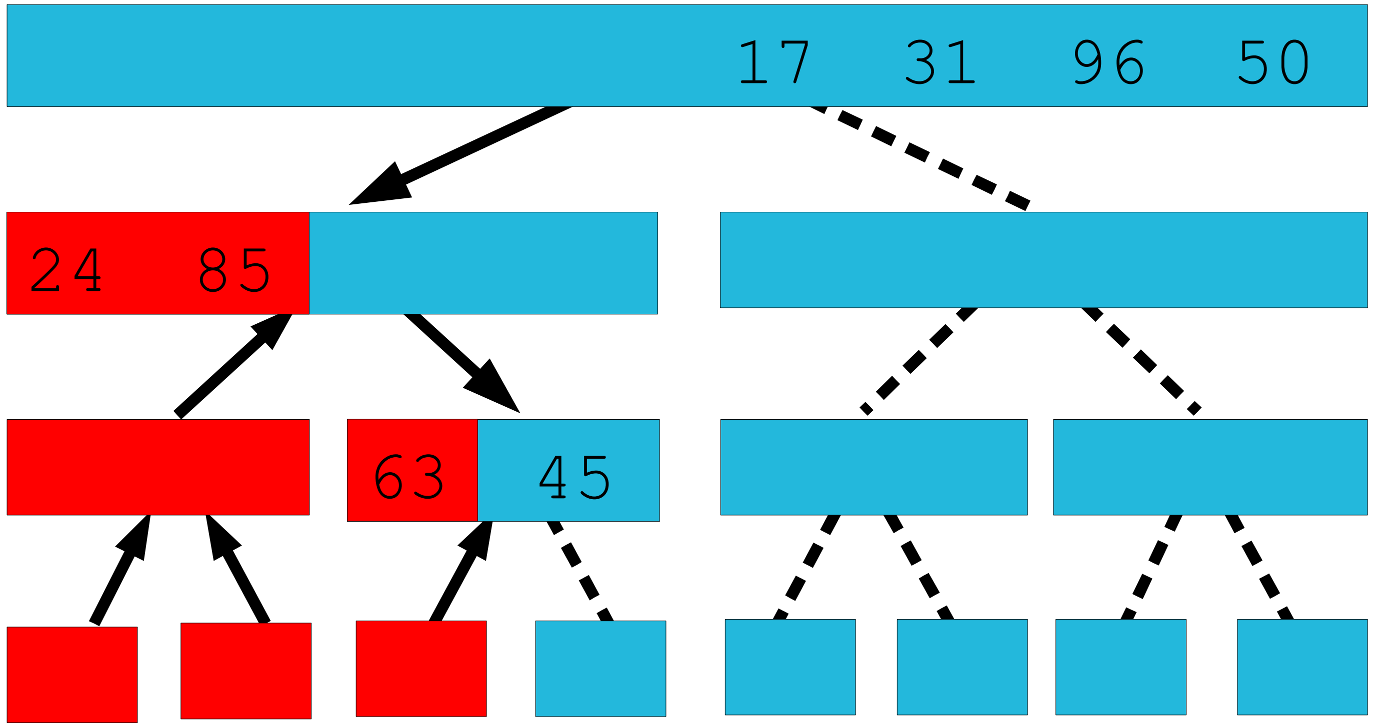
MergeSort Example/10



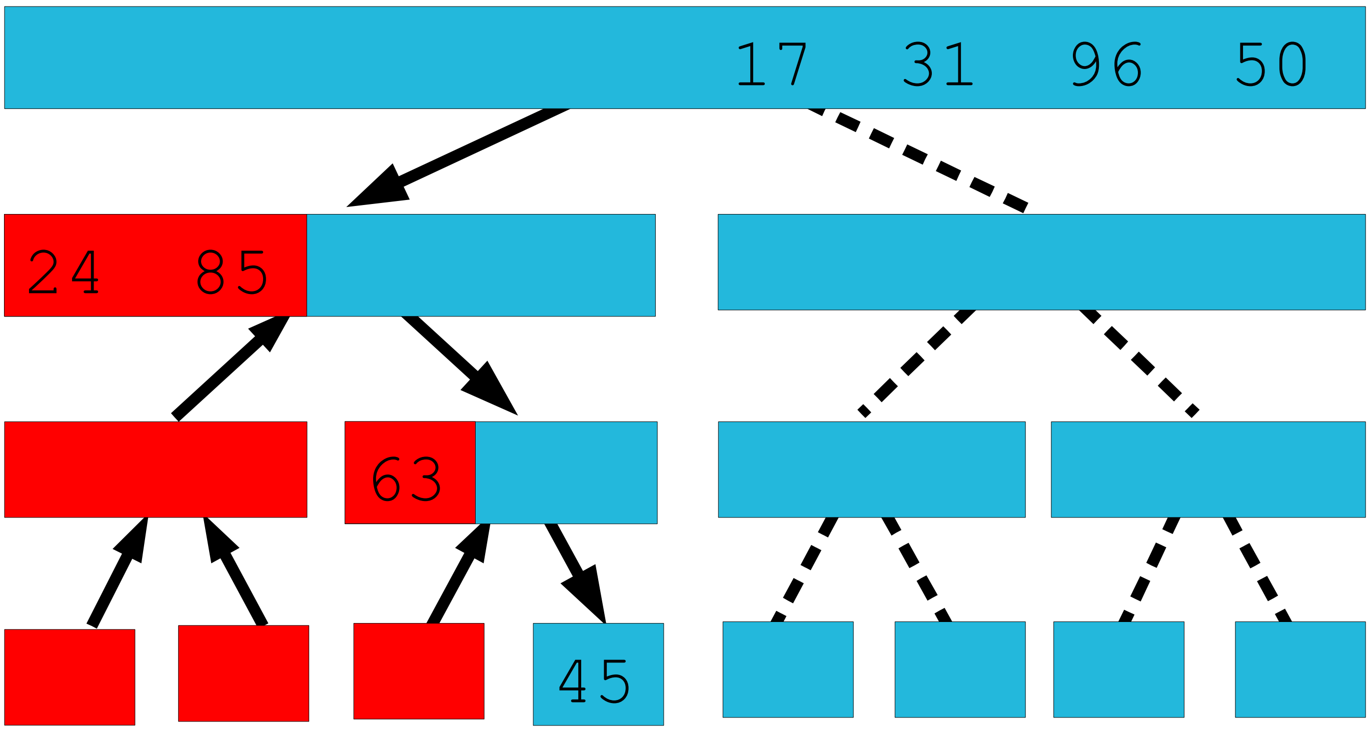
MergeSort Example/11



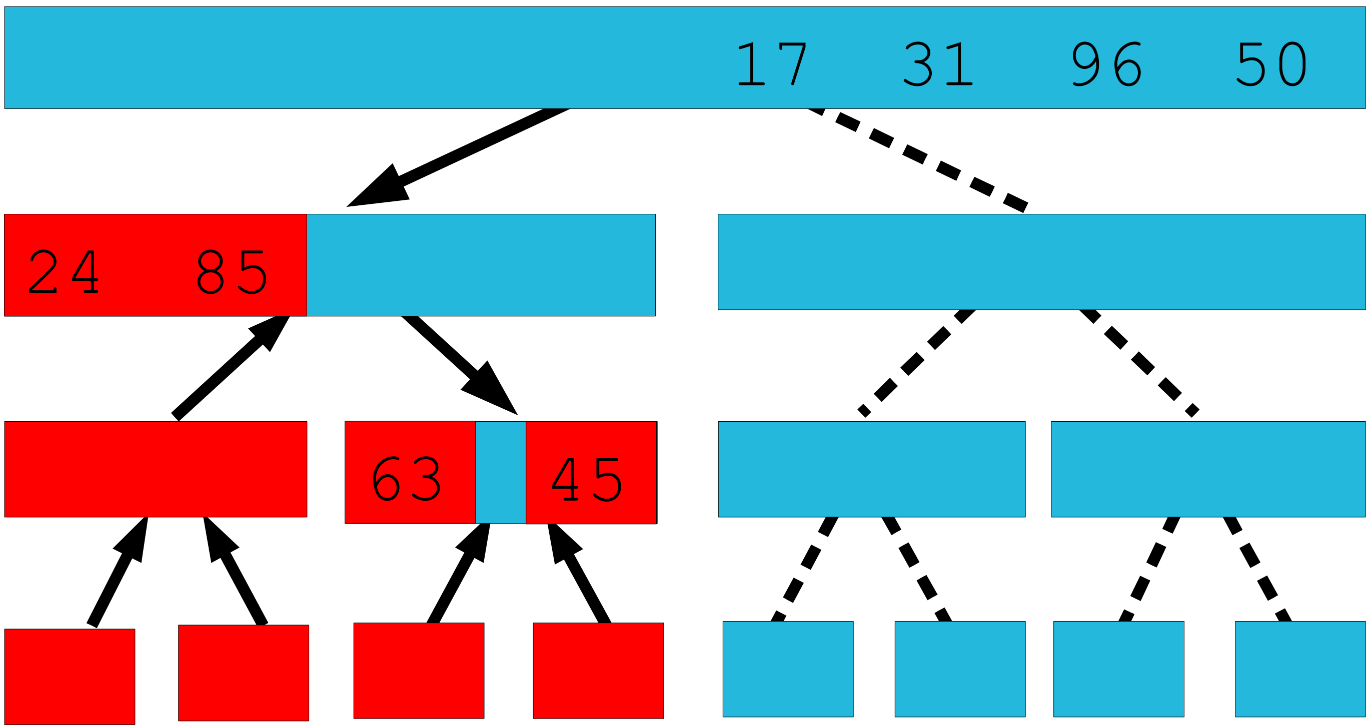
MergeSort Example/12



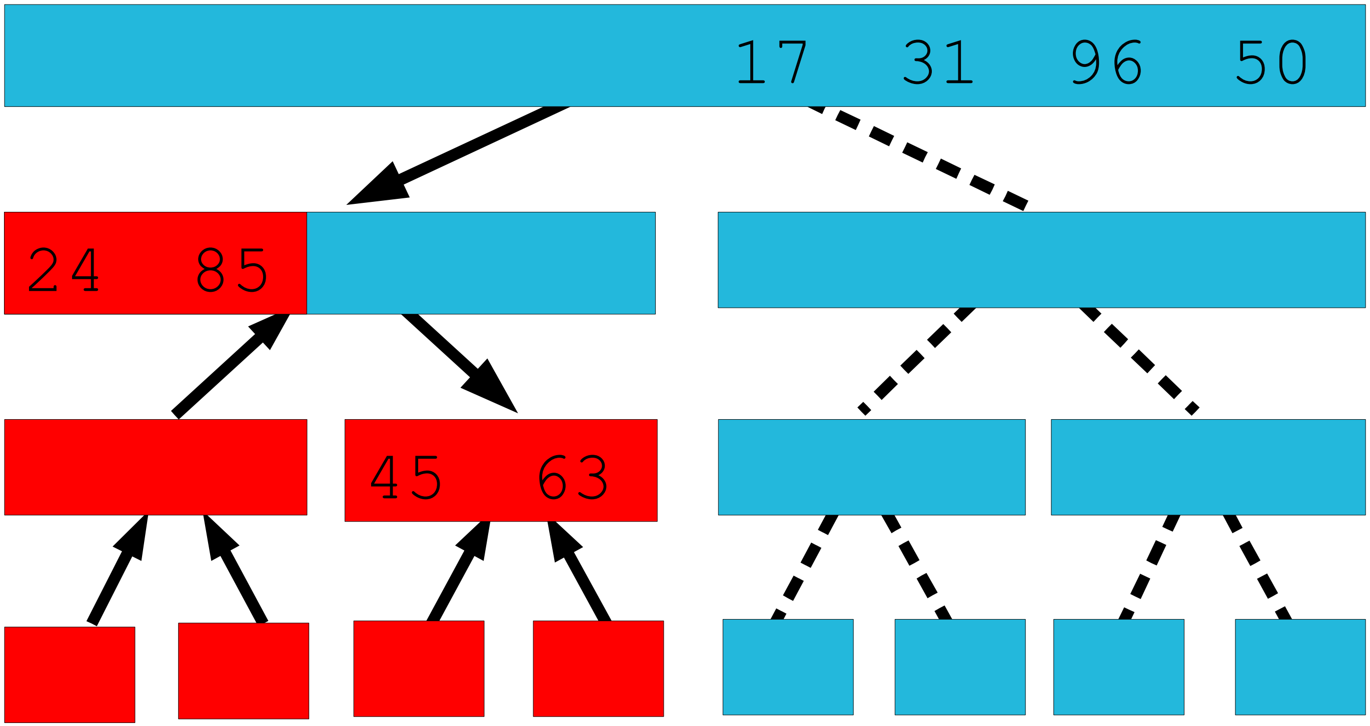
MergeSort Example/13



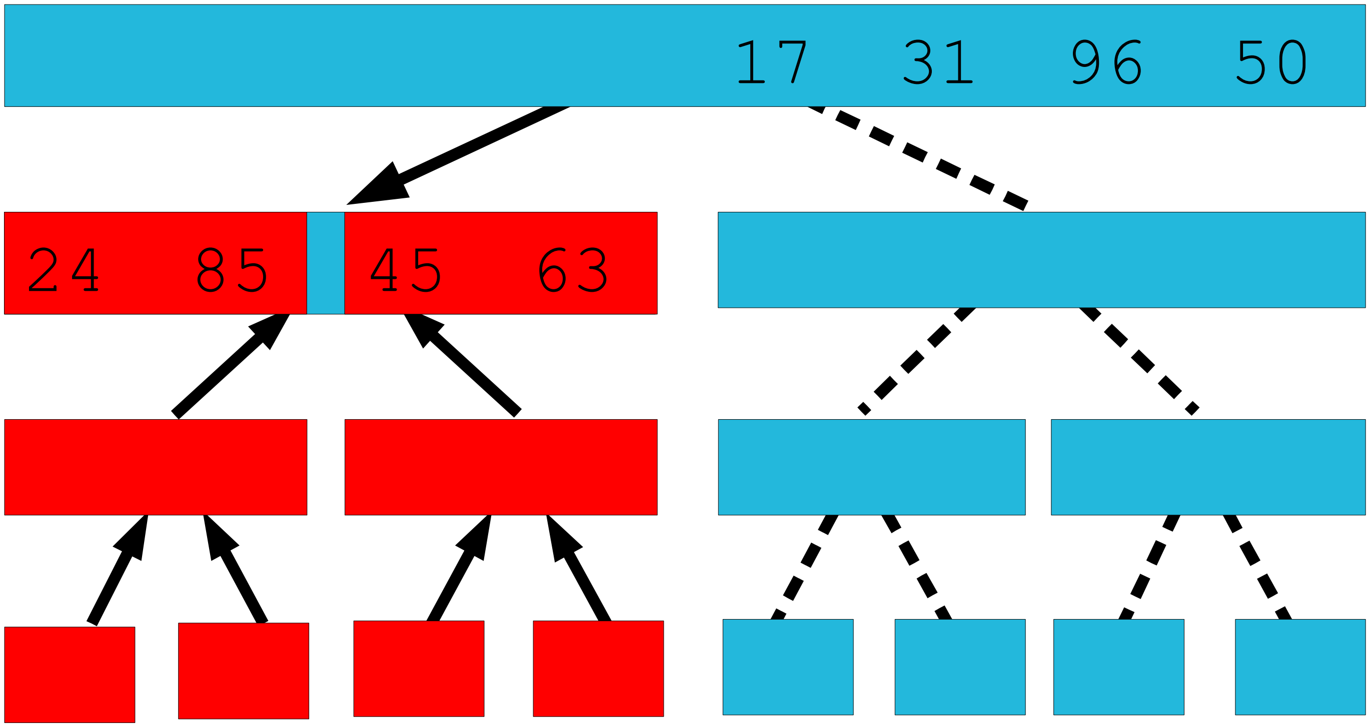
MergeSort Example/14



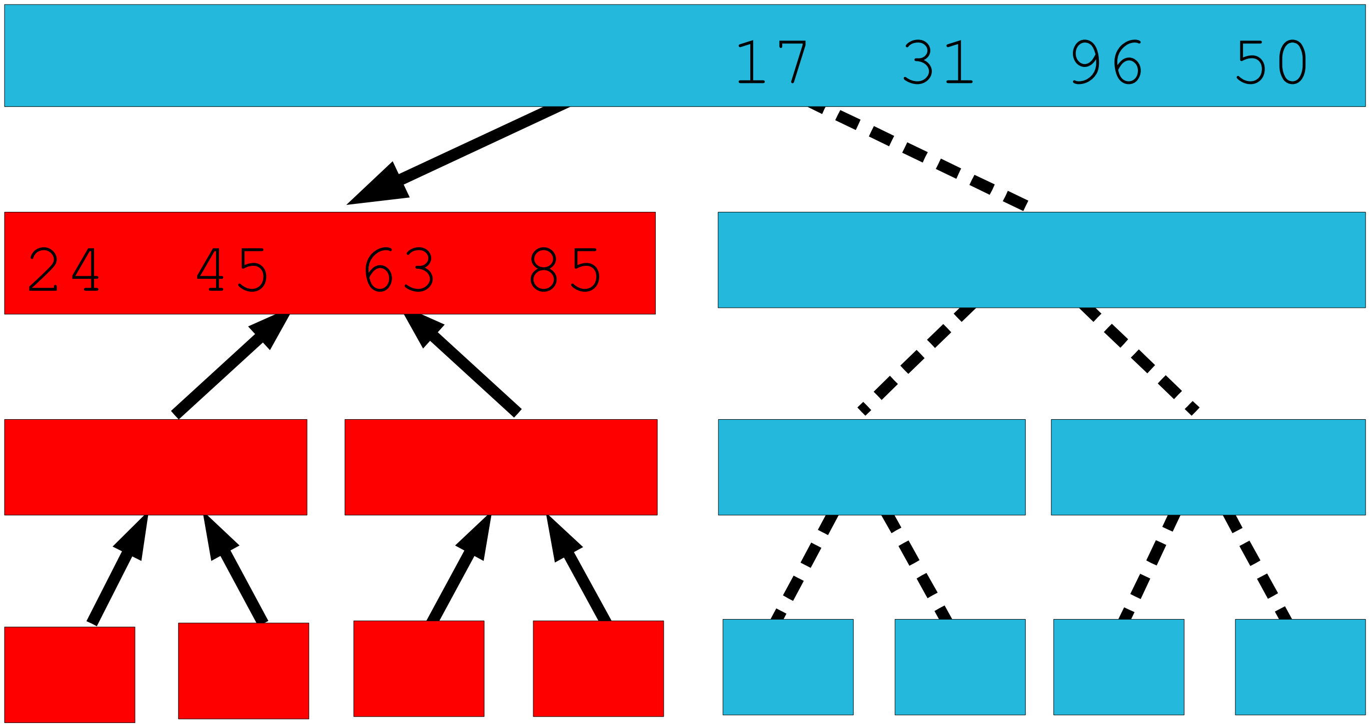
MergeSort Example/15



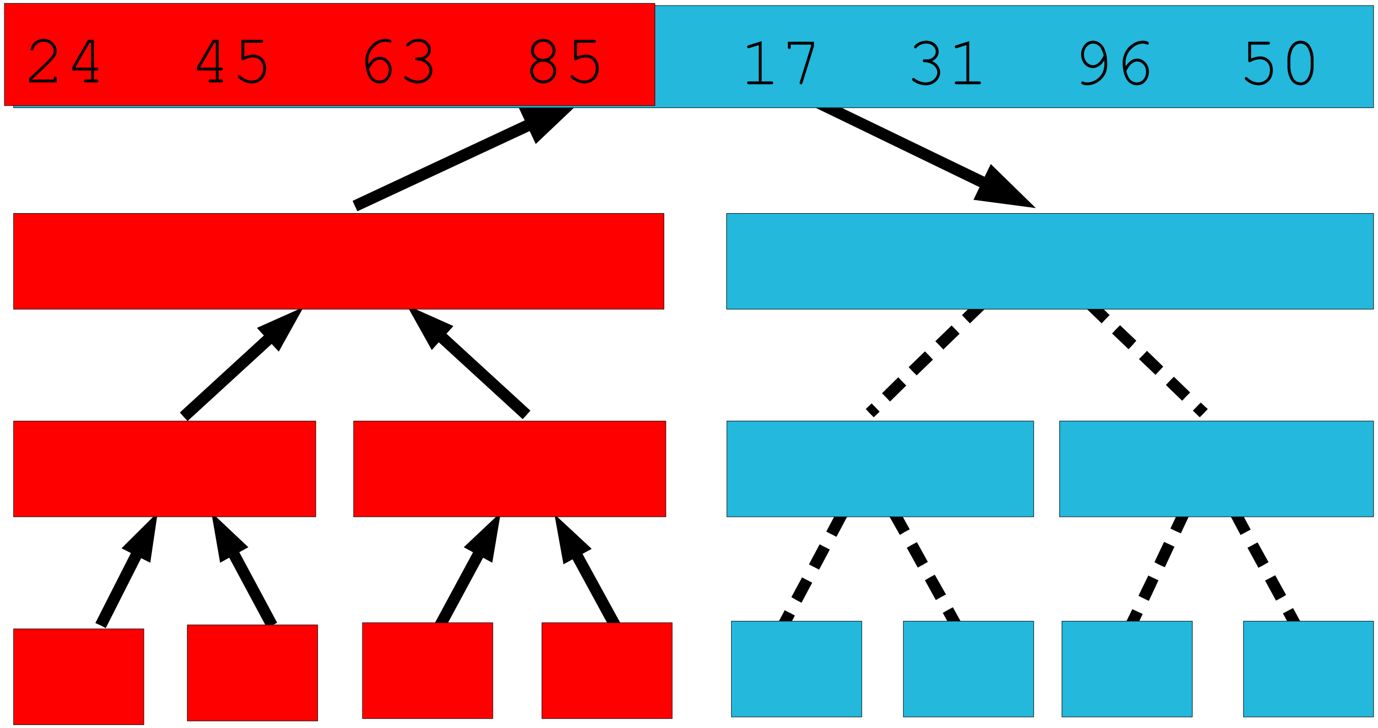
MergeSort Example/16



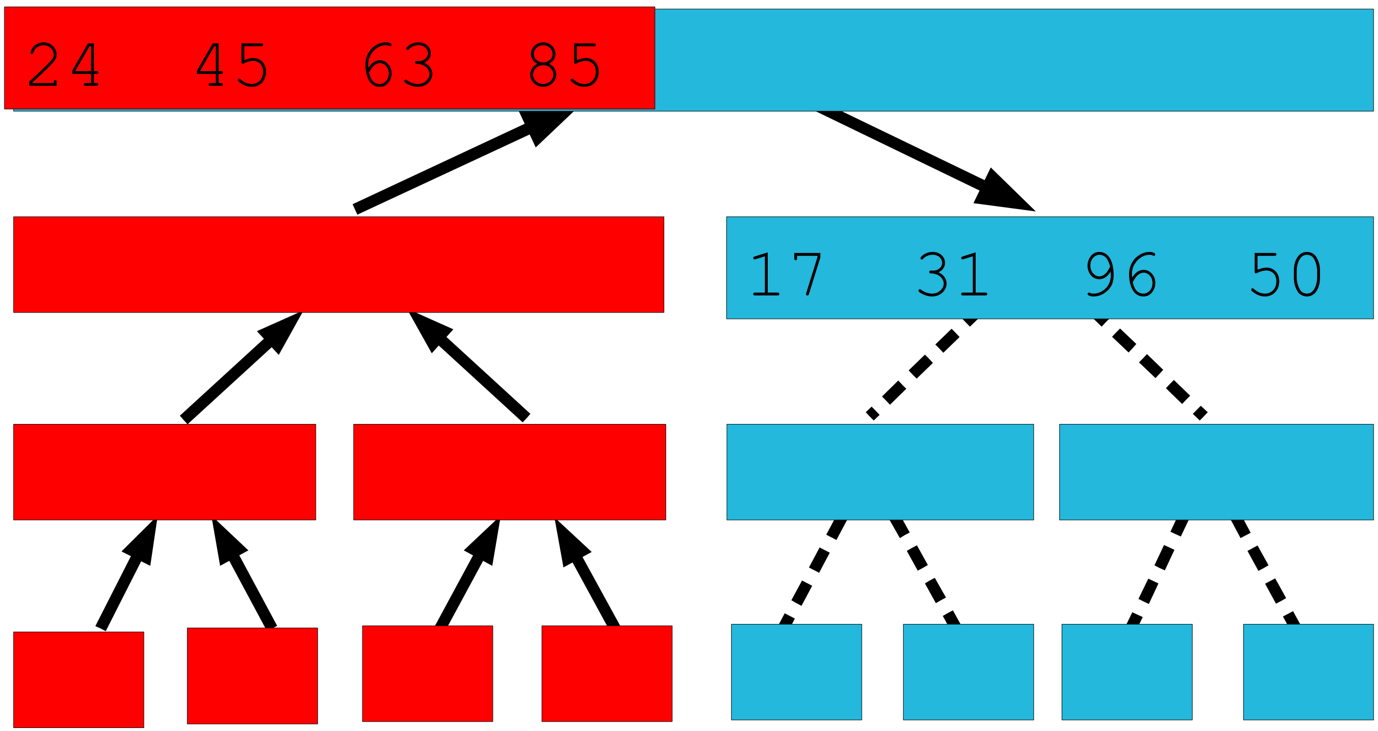
MergeSort Example/17



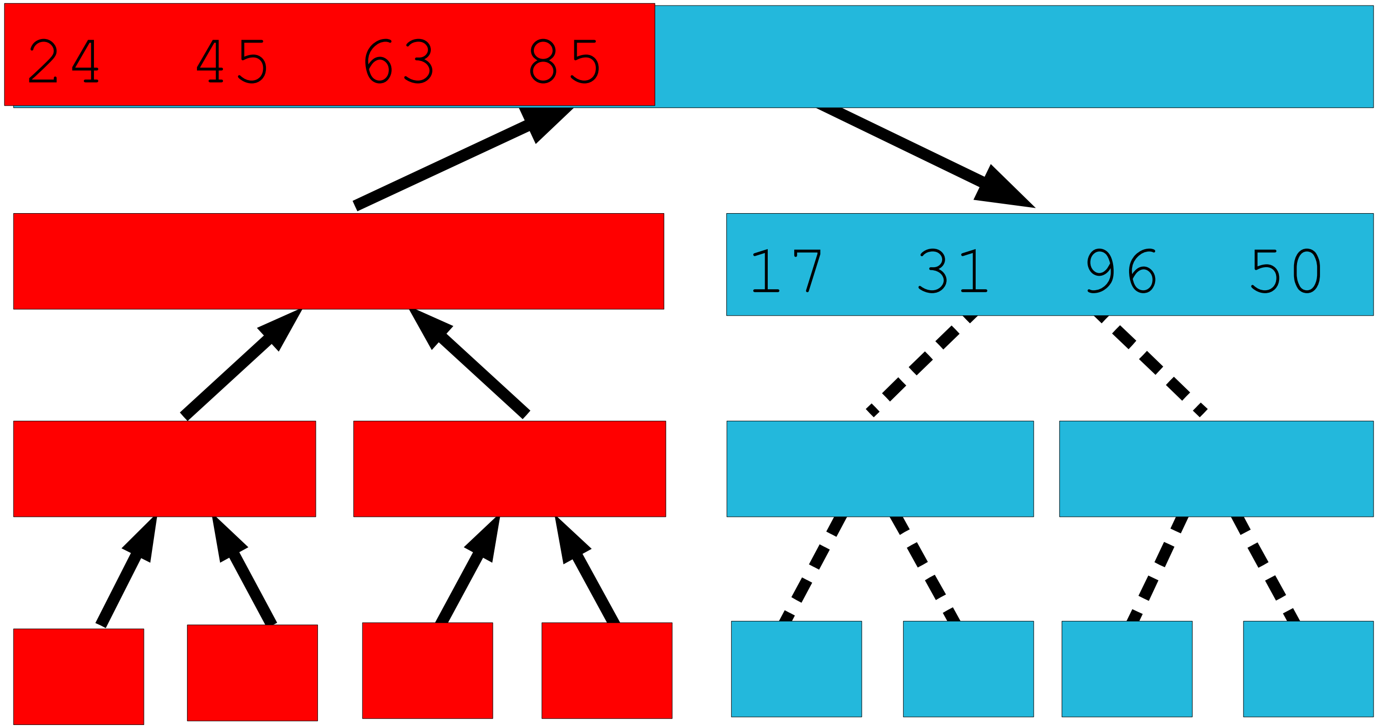
MergeSort Example/18



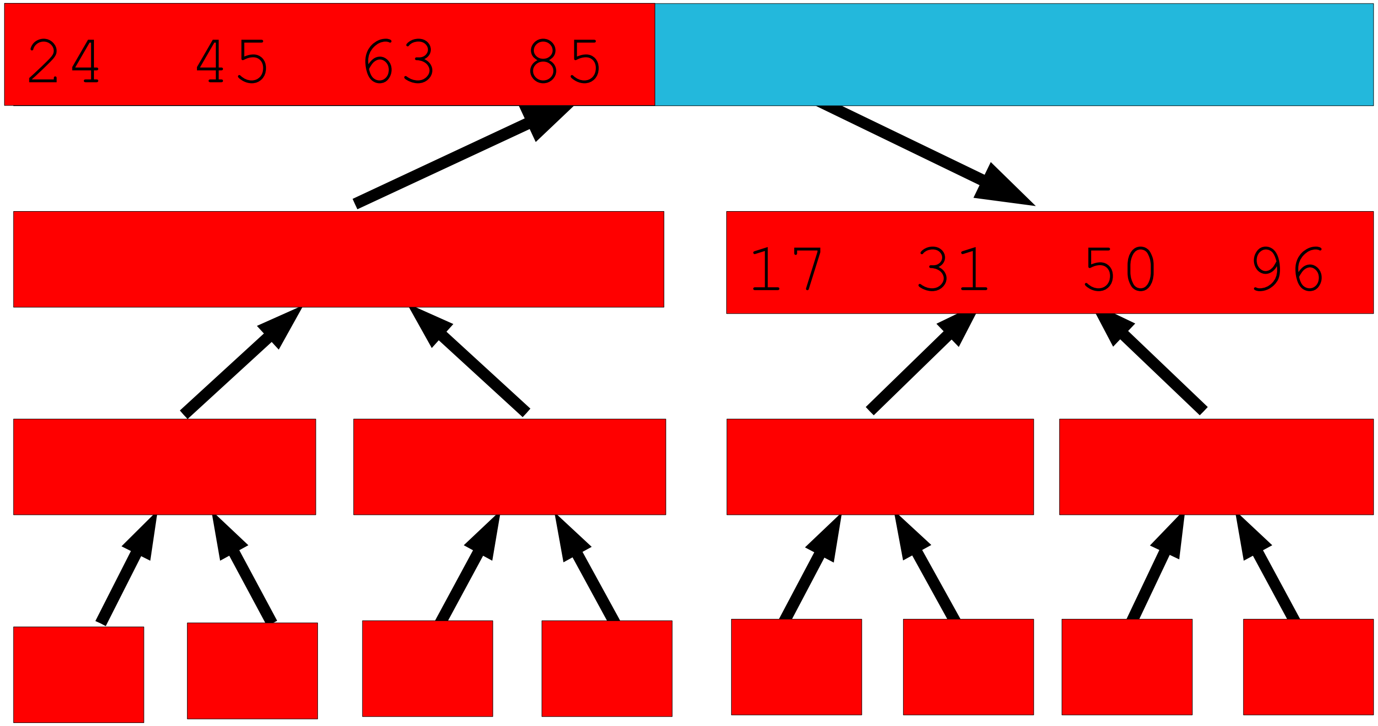
MergeSort Example/19



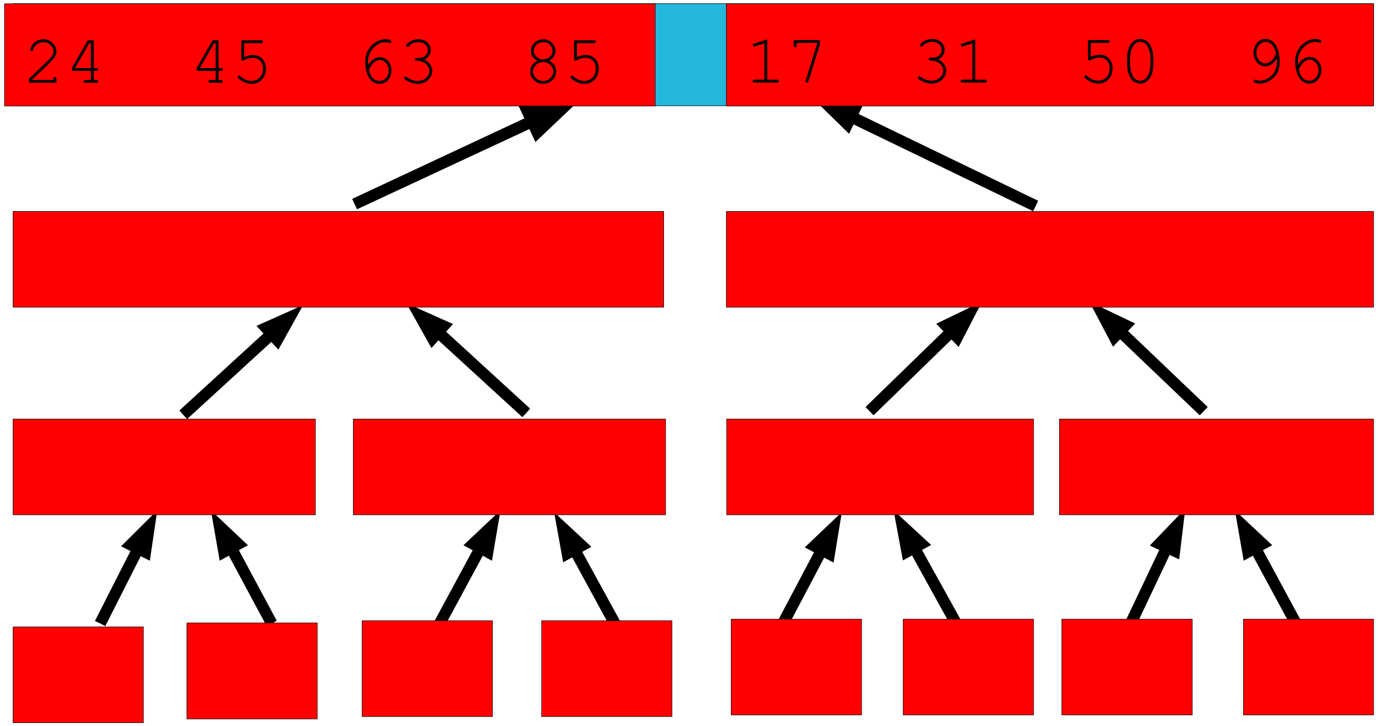
MergeSort Example/19



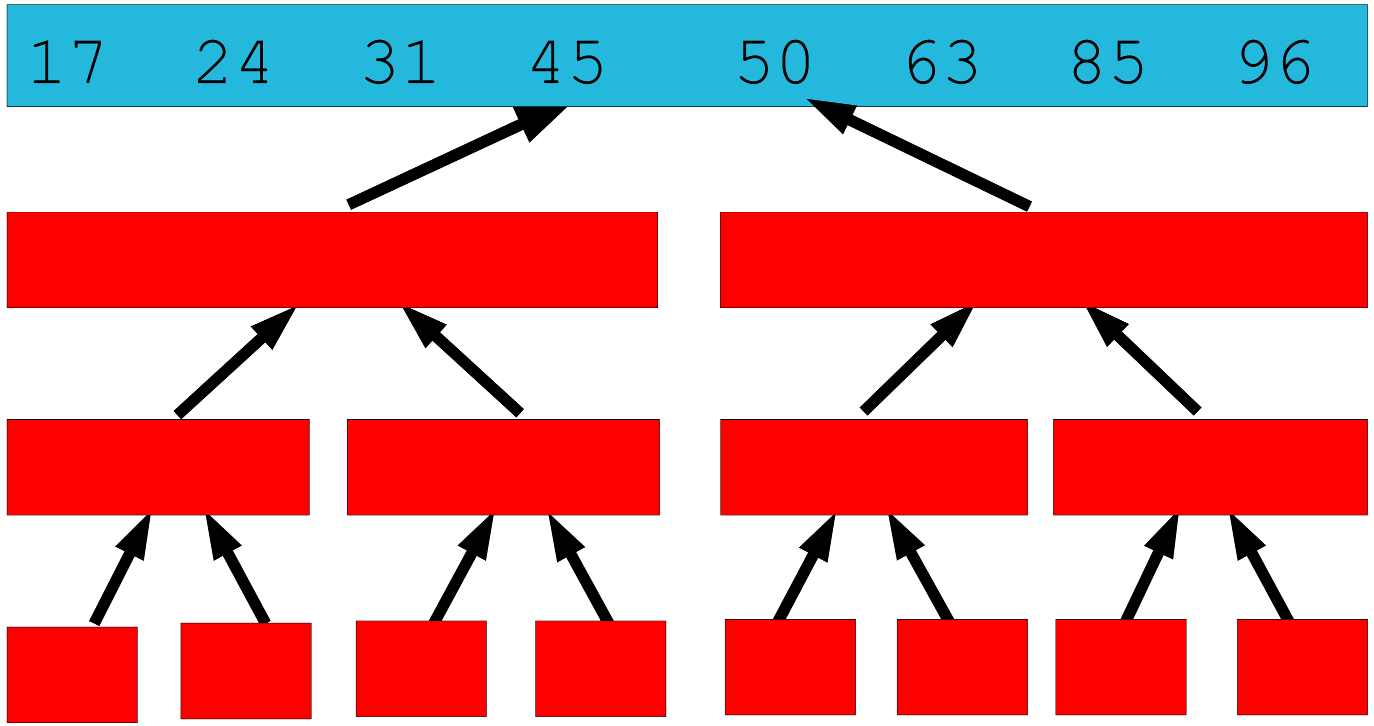
MergeSort Example/20



MergeSort Example/21



MergeSort Example/22



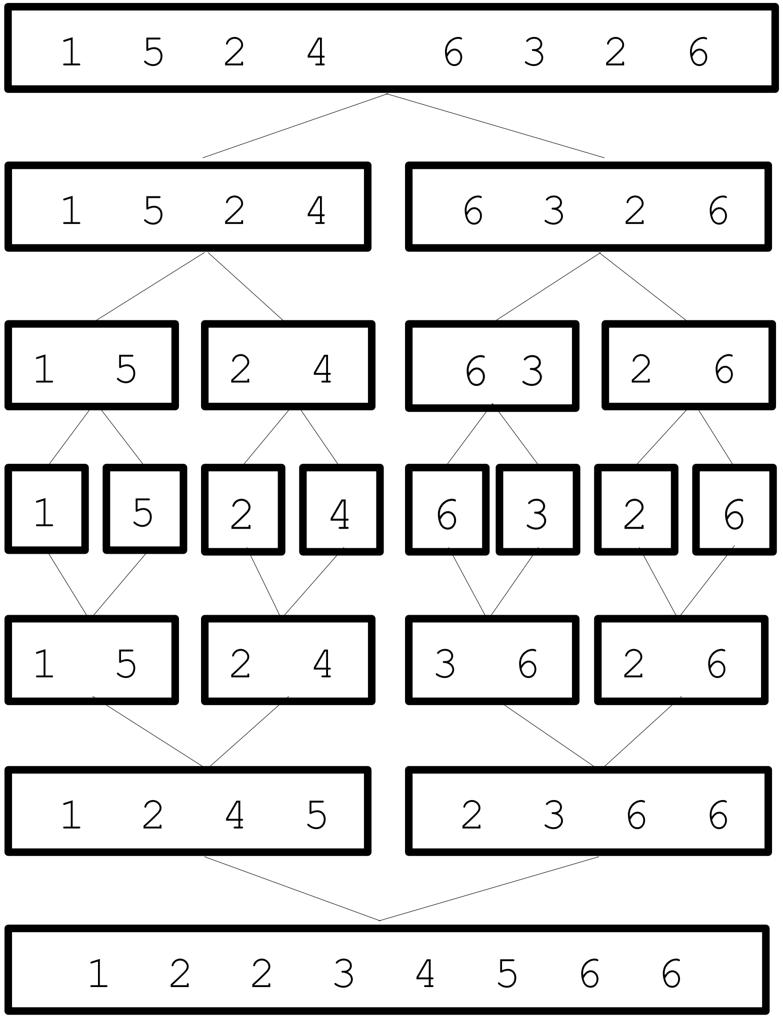
Merge Sort: Algorithm

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

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

Merge Sort Summarized

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



Running Time of Merge Sort

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

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

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

Repeated Substitution Method

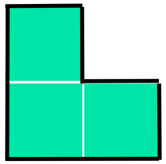
The running time of Merge Sort (assume $n=2^k$).

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

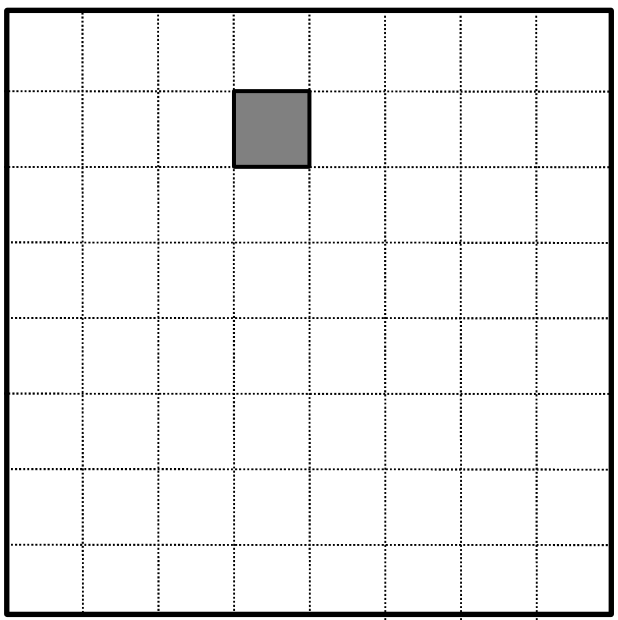
$T(n) = 2T(n/2) + n$		substitute
	$= 2(2T(n/4) + n/2) + n$	expand
	$= 2^2 T(n/4) + 2n$	substitute
	$= 2^2(2T(n/8) + n/4) + 2n$	expand
	$= 2^3 T(n/8) + 3n$	observe pattern
$T(n) = 2^k T(n/2^k) + k\, n$		
	$= 2^{\log n} T(n/n) + n \log n$	
	$= n + n \log n$	

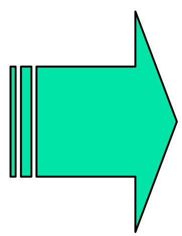
Tiling

A tromino tile:

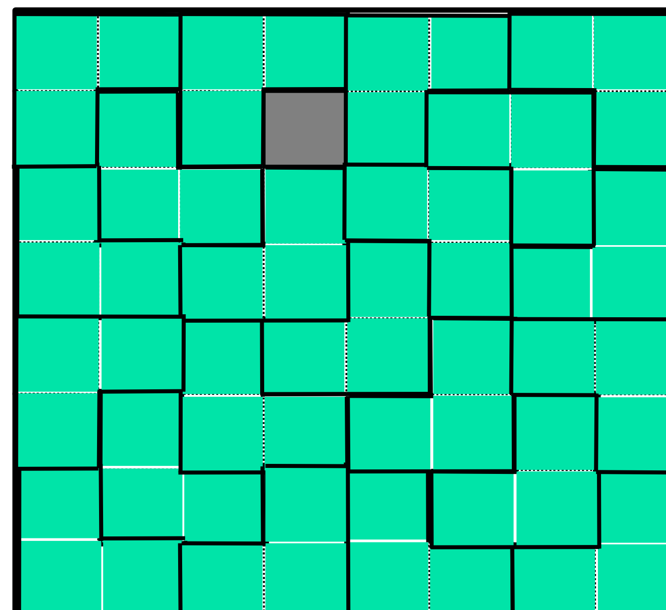


A 2^kx2^k board with a hole:



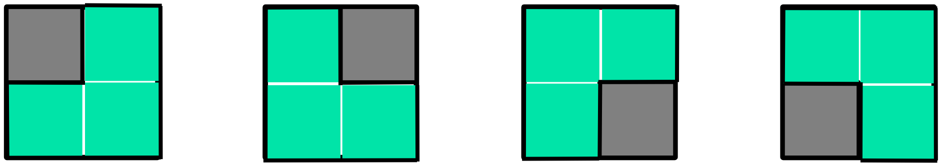


A tiling of the board with trominos:



Tiling: Trivial Case (k = 1)

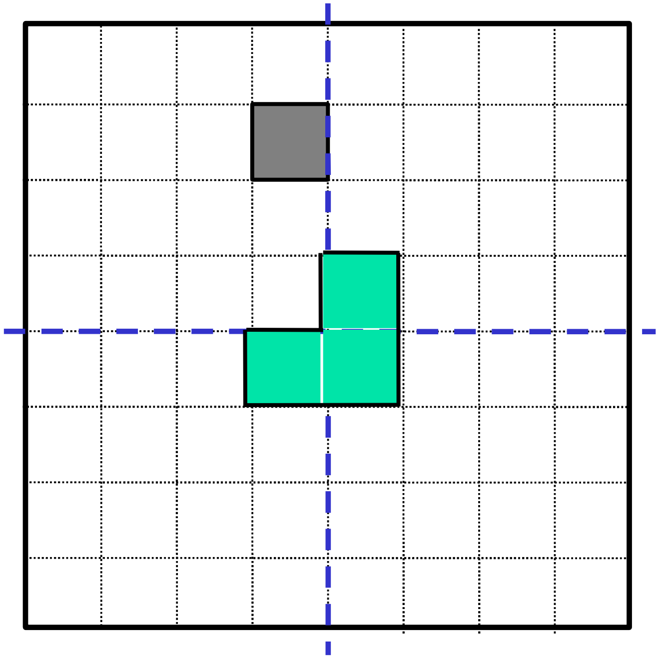
Trivial case (k = 1): tiling a 2x2 board with a hole:



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 $2^{k-1}x2^{k-1}$.
- Keep doing this division, until we get the 2x2 boards with holes – we know how to tile those.

Tiling: Algorithm

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

Tile(k, L)
  if k = 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(k-1, L1)
  Tile(k-1, L2)
  Tile(k-1, L3)
  Tile(k-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:

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

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 in Java

```
public static BigInteger karatsuba(BigInteger x, BigInteger y) {
//
// Copyright © 2000-2011, Robert Sedgewick and Kevin Wayne.
//

// length of number
int N = Math.max(x.bitLength(), y.bitLength());

// number of bits divided by 2, rounded up
N = (N / 2) + (N % 2);
```

Karatsuba Multiplication in Java/2

```
// x = a1 2^N + a0, y = b1 2^N + b0
BigInteger a1 = x.shiftRight(N);
BigInteger a0 = x.subtract(a1.shiftLeft(N));
BigInteger b1 = y.shiftRight(N);
BigInteger b0 = y.subtract(b1.shiftLeft(N));

// compute sub-expressions
BigInteger a0b0 = karatsuba(a0, b0);
BigInteger a1b1 = karatsuba(a1, b1);
BigInteger a0PLUSa1MULTb0PLUSb1 = karatsuba(a0.add(a1), b0.add(b1));

Return a0b0.add(a0PLUSa1MULTb0PLUSb1
                    .subtract(a0b0).subtract(a1b1)
                    .shiftLeft(N))
              .add(a1b1.shiftLeft(2 * N));
}
```

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 (Example)

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

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

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

Repeated Substitution (Example)/2

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

we get $T(n) = 2^k T(n/2^k) + kn$

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

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

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

Repeated Substitution (Example)/2

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

we get $T(n) = 2^k T(n/2^k) + kn$

If $n = 2^k$, then $k = log\ n$:

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

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

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 Sort Merge

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

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

$$\begin{aligned}
T(n) &= 2\color{red}{T(n/2)} + 2n + 3 \quad \text{substitute} \\
&= 2(\color{red}{2T(n/4) + n + 3}) + 2n + 3 \quad \text{expand} \\
&= 2^2\color{blue}{T(n/4)} + 4n + 2*3 + 3 \quad \text{substitute} \\
&= 2^2(\color{blue}{2T(n/8) + n/2 + 3}) + 4n + 2*3 + 3 \quad \text{expand} \\
&= 2^3T(n/2^3) + 2*3n + (2^2+2^1+2^0)*3 \quad \text{observe pattern}
\end{aligned}$$

Analysis of Sort Merge/2

$$T(n) \quad = 2^i T(n/2^i) + 2in + 3$$

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

$$= 2^{\log n} T(n/2^{\log n}) + 2\,n \log n + 3^*(2^{\log n} - 1)$$

$$= 5n + 2\,n \;\; i\, n - 3$$

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

Substitution Method

The substitution method to solve recurrences entails two steps:

- Guess the solution.
- Use induction to prove the solution.

Example:

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

Substitution Method/2

1) *Guess* $T(n) = O(n^3)$, *i.e.,* $T(n)$ *is of the form* cn^3

2) *Prove* $T(n) \leq cn^3$ *by induction*

$T(n)$	$= 4T(n/2) + n$	recurrence
	$\leq 4c(n/2)^3 + n$	induction hypothesis
	$= 0.5cn^3 + n$	simplify
	$= cn^3 - (0.5cn^3 - n)$	rearrange
	$\leq cn^3$ if $c>=2$ and $n>=1$	

Thus $T(n) = O(n^3)$

Substitution Method/3

Tighter bound for $T(n) = 4T(n/2) + n$:

Try to show $T(n) = O(n^2)$

Prove $T(n) \leq cn^2$ by induction

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4c(n/2)^2 + n \\
&= cn^2 + n \\
&\text{NOT} \leq cn^2
\end{aligned}
$$

=> contradiction

Substitution Method/4

- What is the problem? Rewriting

  $$T(n) = O(n^2) = cn^2 + (something\ positive)$$

  as $T(n) \leq cn^2$

  does not work with the inductive proof.
- Solution: Strengthen the hypothesis for the inductive proof:
  - $T(n) \leq$ (*answer you want*) - (*something* > 0)

Substitution Method/5

Fixed proof: strengthen the inductive hypothesis by subtracting lower-order terms:

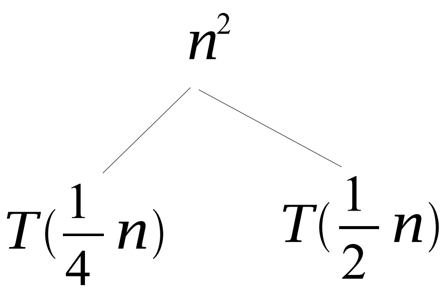
Prove $T(n) \leq cn^2 - dn$ by induction

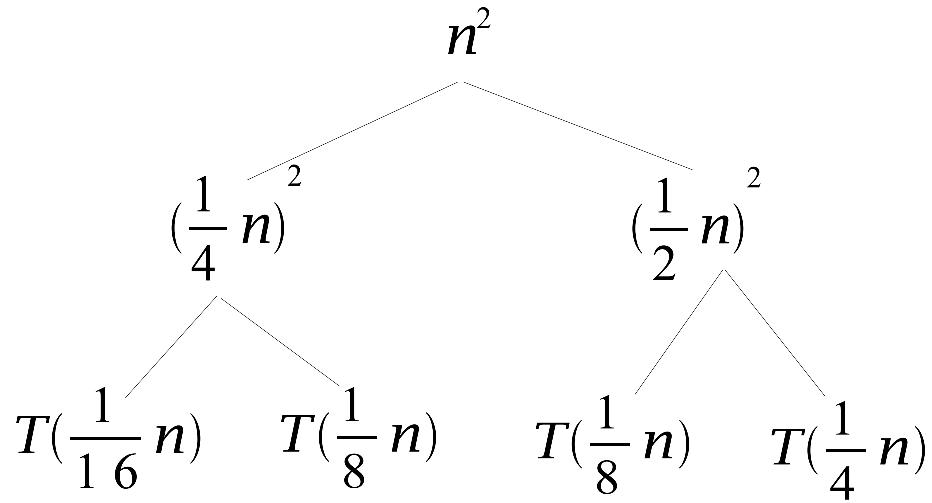
$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c(n/2)^2 - d(n/2)) + n \\
&= cn^2 - 2dn + n \\
&= cn^2 - dn - (dn - n) \\
&\leq cn^2 - dn \text{ if } d \geq 1
\end{aligned}
$$

Recursion Tree

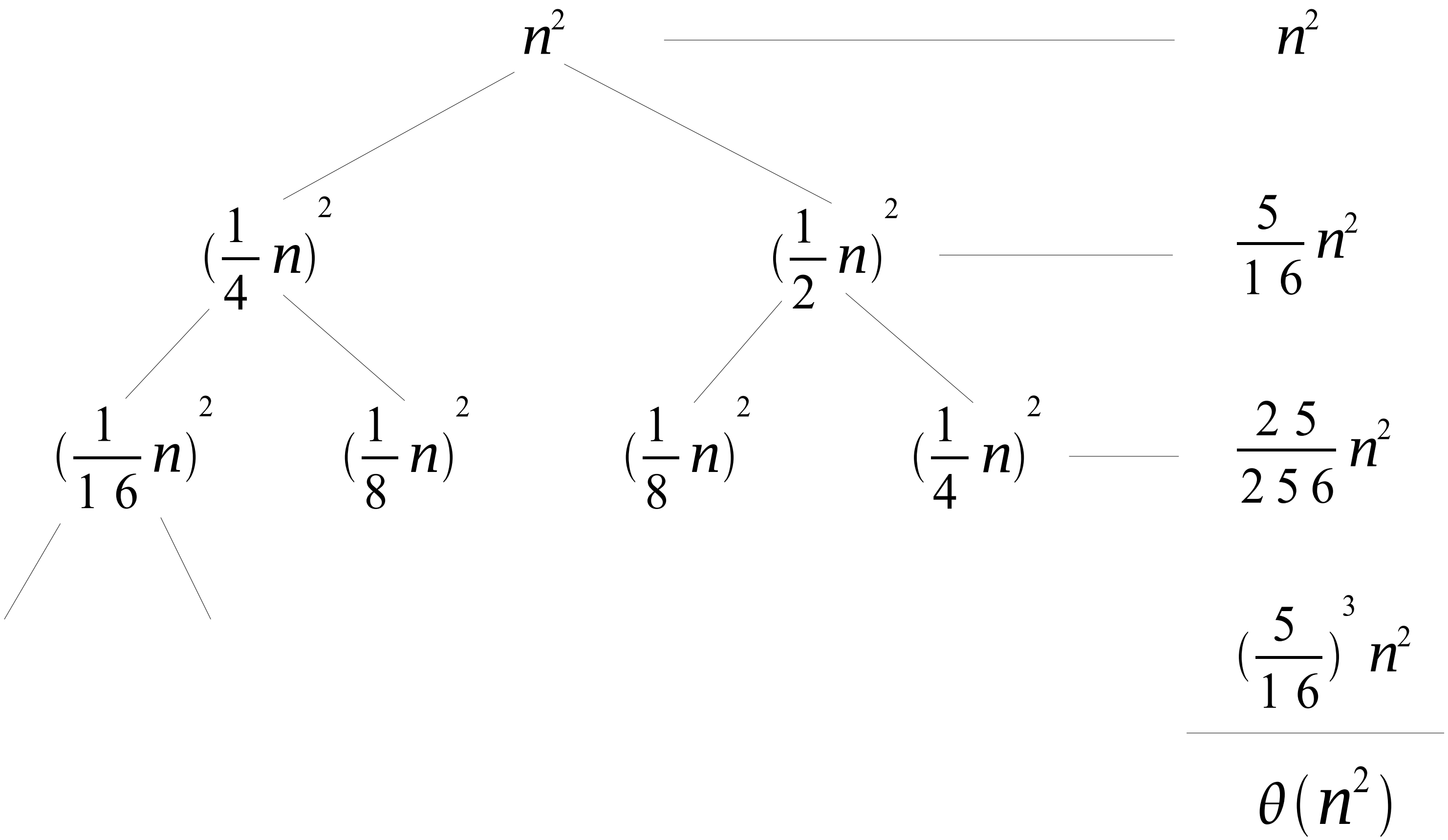
A recursion tree is a convenient way to visualize what happens when a recurrence is iterated.

– Good for ”guessing” asymptotic solutions to recurrences





Recursion Tree/2



Master Method

- The idea is to solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- *Assumptions:* $a \geq 1$ and $b > 1$, and $f(n)$ is asymptotically positive.
- Abstractly speaking, $T(n)$ is the runtime for an algorithm and we know that
  - a subproblem of size n/b are solved recursively, each in time $T(n/b)$.
  - $f(n)$ is the cost of dividing the problem and combining the results.

In merge-sort $T(n) = 2T(n/2) + \Theta(n)$.

Master Method/2

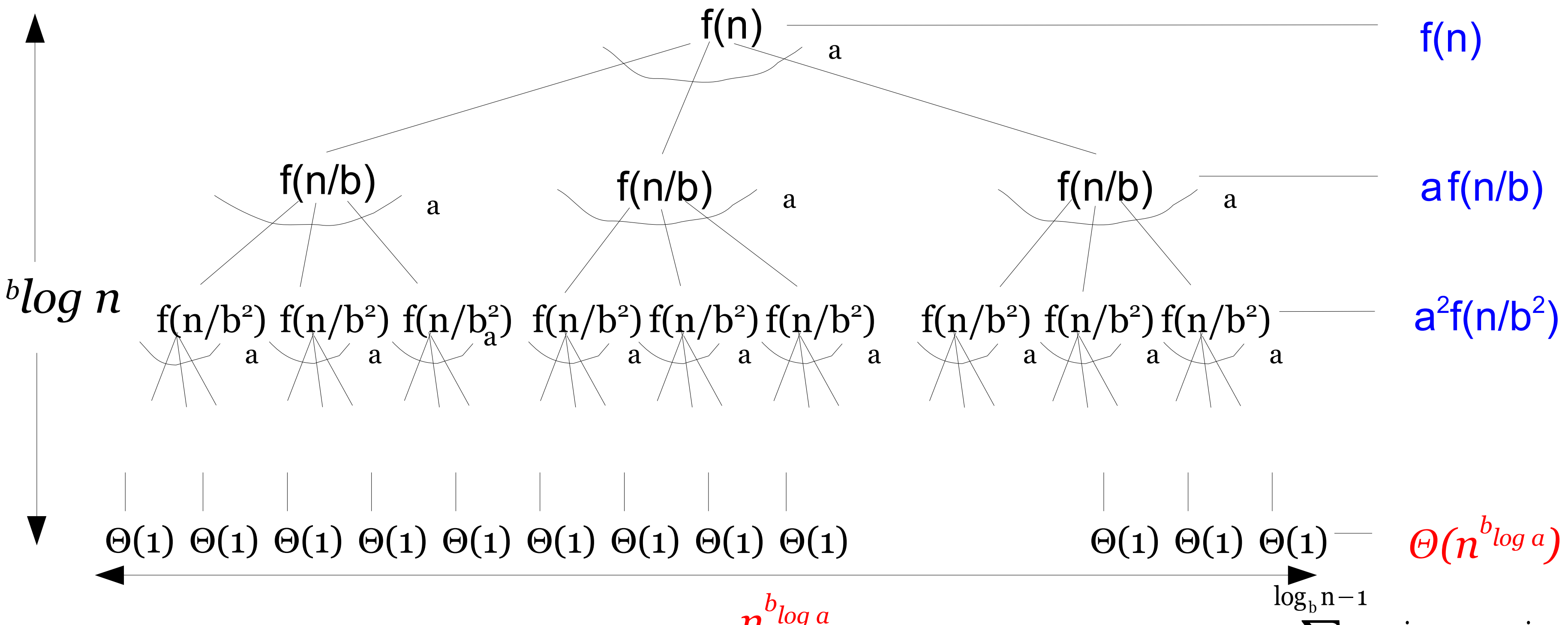
- Iterating the recurrence (expanding the tree) yields

$$T(n) = f(n) + aT(n/b)$$
$$= f(n) + af(n/b) + a^2T(n/b^2)$$
$$= f(n) + af(n/b) + a^2f(n/b^2) + \ldots$$
$$+ a^{{}^b\log n-1}f(n/a^{{}^b\log n-1}) + a^{{}^b\log n}T(1)$$

$$T(n) = \sum_{j=0}^{{}^b\log n-1} a^j f(n/b^j) + \Theta(n^{{}^b\log a})$$

- The first term is a division/recombination cost (totaled across all levels of the tree).
- The second term is the cost of doing all subproblems of size 1 (total of all work pushed to leaves).

Master Method/3



$$\textit{Total: } \Theta(n^{{}^{b}\log a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

Note: split into a parts, ${}^{b}\log n$ levels, $a^{{}^{b}\log n} = n^{{}^{b}\log a}$ leaves.

Master Method, Intuition

- Three common cases:
  - Running time dominated by cost at leaves.
  - Running time evenly distributed throughout the tree.
  - Running time dominated by cost at the root.
- To solve the recurrence, we need to identify the dominant term.
- In each case compare $f(n)$ with $O(n^{\log_b a})$.

Master Method, Case 1

$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$

– $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor n^{ε}).

The work at the leaf level dominates

$$T(n) = \Theta(n^{\log_b a})$$

Cost of all the leaves

Master Method, Case 2

$f(n) = \Theta(n^{\log_b a})$

- $f(n)$ and $n^{\log_b a}$ are asymptotically the same

The work is distributed equally throughout the tree

$$T(n) = \Theta(n^{\log_b a} \log n)$$

(level cost) × (number of levels)

Master Method, Case 3

- $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$
  - Inverse of Case 1
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$
  - Also need a “regularity” condition

$$\exists c < 1 \text{ and } n_0 > 0 \text{ such that } a f(n/b) \le c f(n) \quad \forall n > n_0$$

The work at the root dominates

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

division/recombination cost

Master Theorem Summarized

Given: recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a-\varepsilon})$
   $\Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$
   $\Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ and
   $a\, f(n/b) \leq \alpha\, f(n)$ for some $\alpha < 1$, $n>n_0$
   $\Rightarrow T(n) = \Theta(f(n))$

Strategy

1. Extract *a*, *b*, and *f(n)* from a given recurrence
2. Determine $n^{^b\log a}$
3. Compare *f(n)* and $n^{^b\log a}$ asymptotically
4. Determine appropriate MT case and apply it

Merge sort: $T(n) = 2T(n/2) + \Theta(n)$

$a=2, b=2, f(n) = \Theta(n)$

$n^{^2\log 2} = n$

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

=> Case 2: $T(n) = \Theta(n^{^b\log a} \log n) = \Theta(n \log n)$

Examples of Master Method

```
BinarySearch(A, l, r, q):
   m := (l+r)/2
   if A[m]=q then return m
   else if A[m]>q then
        BinarySearch(A, l, m-1, q)
   else BinarySearch(A, m+1, r, q)
```

$T(n) = T(n/2) + 1$

$a=1, b=2, f(n) = 1$

$n^{{}^{2}\log 1} = 1$

$1 = \Theta(1)$

=> Case 2: $T(n) = \Theta(\log n)$

Examples of Master Method/2

$T(n) = 9T(n/3) + n$

$a=9, b=3, f(n) = n$

$n^{3\log 9} = n^2$

$n = O(n^{3\log 9 - \varepsilon})$ with $\varepsilon = 1$

=> Case 1: $T(n) = \Theta(n^2)$

Examples of Master Method/3

$T(n) = 3T(n/4) + n \log n$

$a=3, b=4, f(n) = n \log n$

$n^{4\log 3} = n^{0.792}$

$n \log n = \Omega(n^{4\log 3 + \varepsilon})$ with $\varepsilon = 0.208$

=> Case 3:

regularity condition: $a\, f(n/b) <= c\, f(n)$

$a\, f(n/b) = 3(n/4)\log(n/4) <=$

$(3/4)n \log n = c\, f(n)$ with $c=3/4$

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

BinarySearchRec1

Find a number in a sorted array:

- trivial if the array contains one element
- else divide into two equal halves and solve each half
- combine the results

```
INPUT:  A[1..n] – sorted array of integers, q – integer
OUTPUT: index j s.t. A[j] = q, NIL if ∀j(1≤j≤n): A[j] ≠ q

BinarySearchRec1(A, l, r, q):
   if l = r then
     if A[l] = q then return l else return NIL
   m := ⌊(l+r)/2⌋
   ret := BinarySearchRec1(A, l, m, q)
   if ret = NIL then return BinarySearchRec1(A, m+1, r, q)
   else return ret
```

$T(n)$ of BinarySearchRec1

Example: BinarySearchRec1

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

Solving the recurrence yields

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

BinarySearchRec2

$T(n) = \Theta(n)$ – not better than brute force!

Better way to conquer:

– Solve only one half!

```
INPUT:  A[1..n] – sorted array of integers, q – integer
OUTPUT: j s.t. A[j] = q, NIL if ∀j(1≤j≤n): A[j] ≠ q

BinarySearchRec2(A, l, r, q):
   if l = r then
      if A[l] = q then return l
      else return NIL
   m := ⌊(l+r)/2⌋
   if A[m] ≤ q then return BinarySearchRec2(A, l, m, q)
   else return BinarySearchRec2(A, m+1, r, q)
```

T(*n*) of BinarySearchRec2

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

Solving the recurrence yields

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

Example: Finding Min and Max

Given an unsorted array, find a minimum and a maximum element in the array

```
INPUT: A[l..r] – an unsorted array of integers, l≤r.
OUTPUT: (min,max) s.t. ∀j(l≤j≤r): A[j]≥min and A[j]≤max

MinMax(A, l, r):
   if l = r then return (A[l], A[r]) // Trivial case
   m := ⌊(l+r)/2⌋                     // Divide
   (minl,maxl) := MinMax(A, l, m)     // Conquer
   (minr,maxr) := MinMax(A, m+1, r)   // Conquer
   if minl < minr then min = minl else min = minr // Combine
   if maxl > maxr then max = maxl else max = maxr // Combine
   return (min,max)
```

Summary

- The Divide and Conquer principle
- Merge sort
- Tiling
- Computing powers
- Karatsuba multiplication
- Recurrences
  - repeated substitutions
  - substitution
  - Master method
- Example recurrences: Binary search

Next Chapter

- Sorting
  - HeapSort
  - QuickSort