Data Structures and Algorithms

Roberto Sebastiani
roberto.sebastiani@disi.unitn.it
http://www.disi.unitn.it/~rseba

- Week 03 -
B.S. In Applied Computer Science
Free University of Bozen/Bolzano
academic year 2010-2011

Acknowledgements & Copyright Notice

These slides are built on top of slides developed by Michael Boehlen**.** Moreover, some material (text, figures, examples) displayed in these slides is courtesy of **Kurt Ranalter**. Some exampes displayed in these slides are taken from [**Cormen, Leiserson, Rivest and Stein**, ``Introduction to Algorithms", MIT Press], and their copyright is detained by the authors. All the other material is copyrighted by **Roberto Sebastiani**. Every commercial use of this material is strictly forbidden by the copyright laws without the authorization of the authors. No copy of these slides can be displayed in public or be publicly distributed without containing this copyright notice.

Data Structures and Algorithms

Week 3

1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. Recurrences

Data Structures and Algorithms

Week 3

1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. 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 decomposition eventually yields the base case.
  - The decomposition must contribute to solving the original problem.

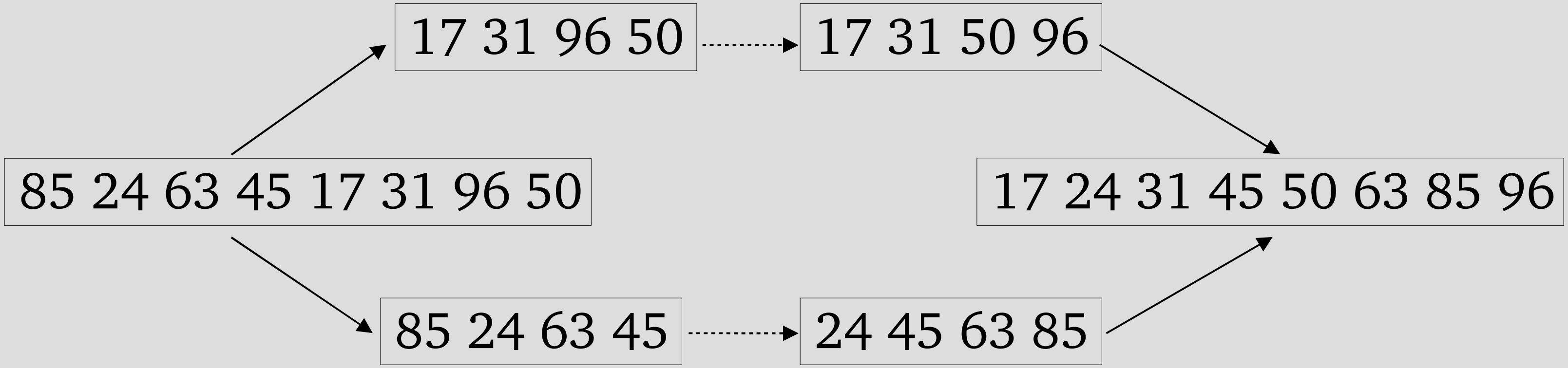
Data Structures and Algorithms

Week 3

1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. Recurrences

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

85 24 63 45

17 31 50 96

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 into the resulting sequence. Repeat this, until both sequences are empty. Copy the resulting sequence into A[l..r].

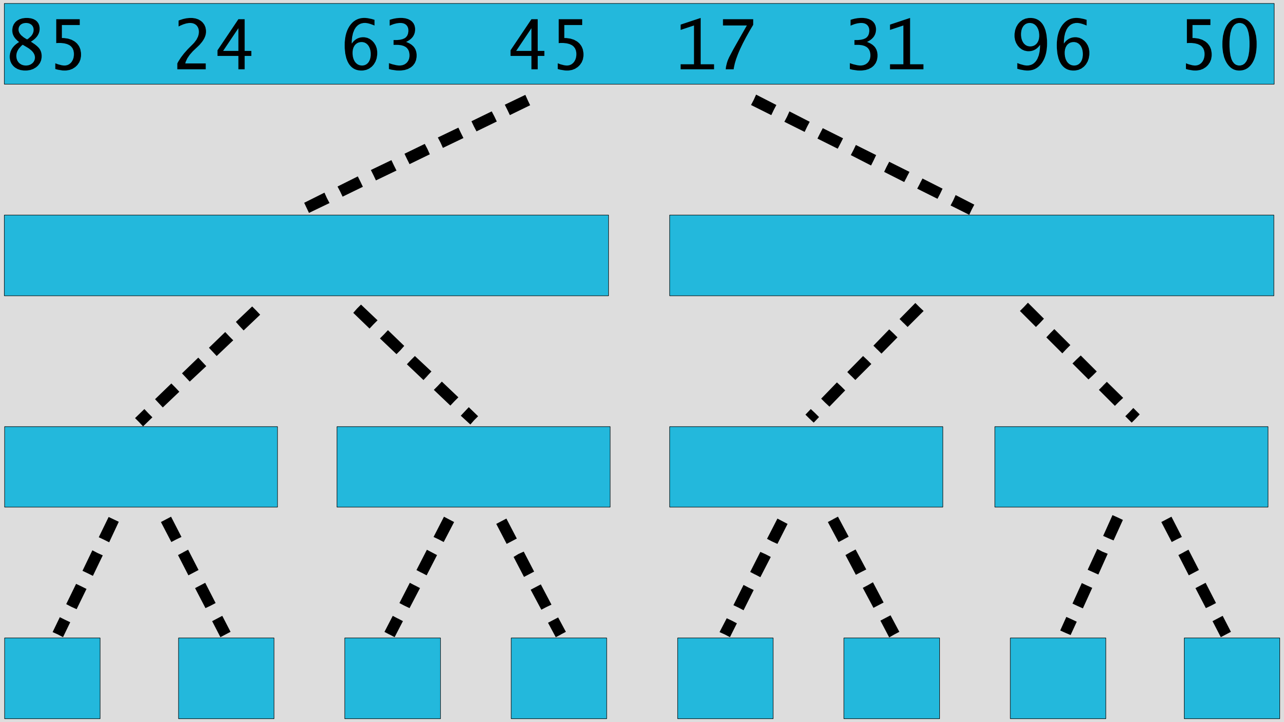
Merge

INPUT: A[1..n1], B[1..n2] sorted arrays of integers

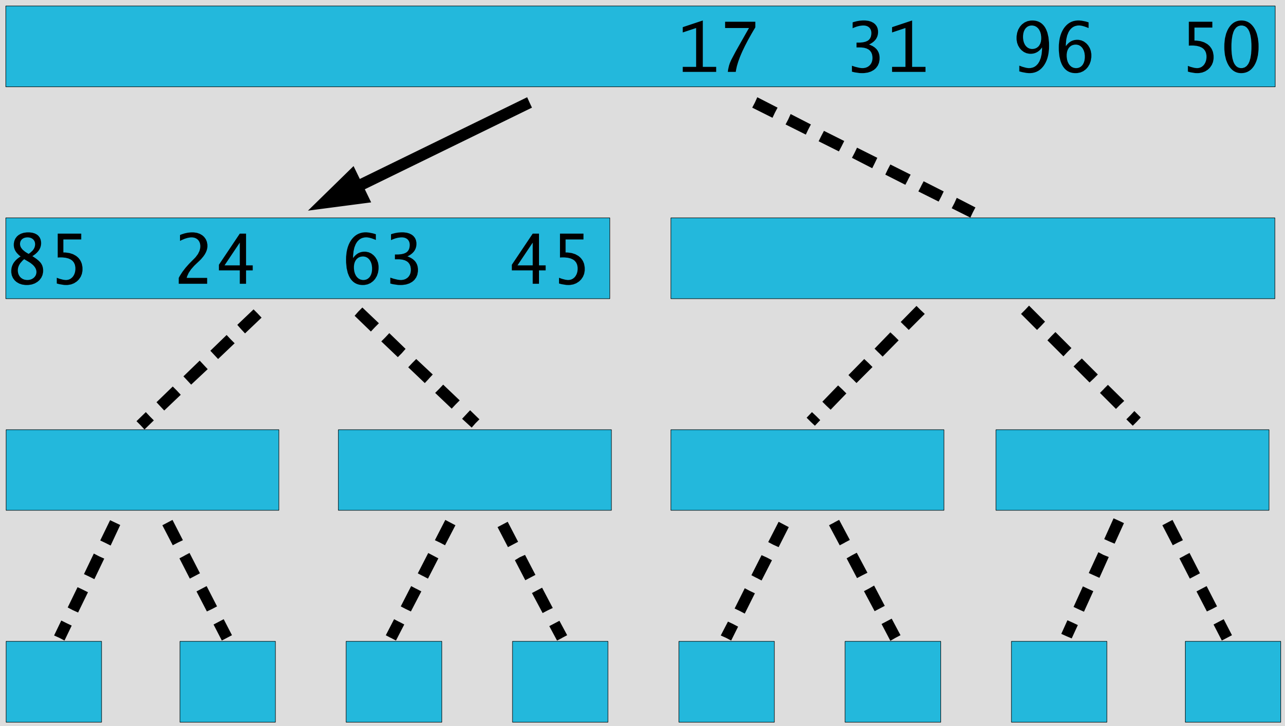
OUTPUT: permutation C of A.B s.t. C[1]≤C[2]≤...≤C[n1+n2]

```
i=1;j=1;
for k:= 1 to n1+n2 do
   If j>n2 or (i<=n1 and A[i]<=B[j])
   Then C[k]=A[i];i=i+1;
   Else C[k]=B[j];j=j+1;
Return C;
```

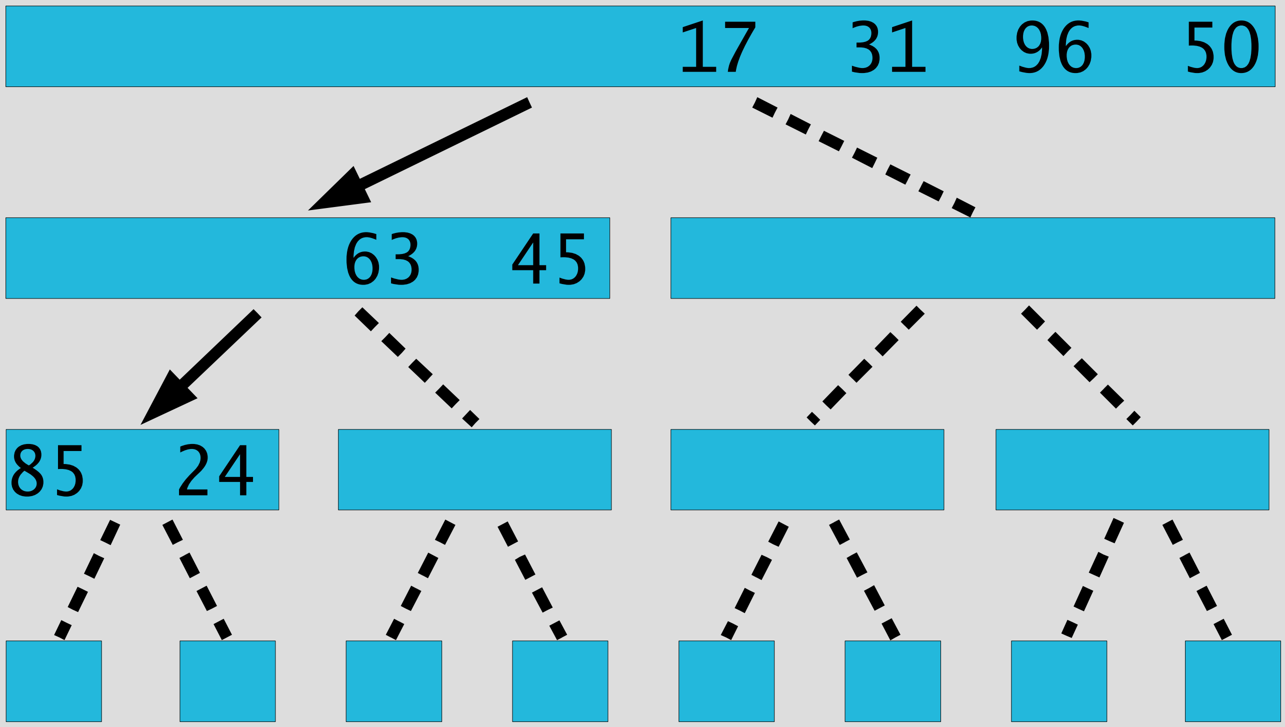
MergeSort Example/1



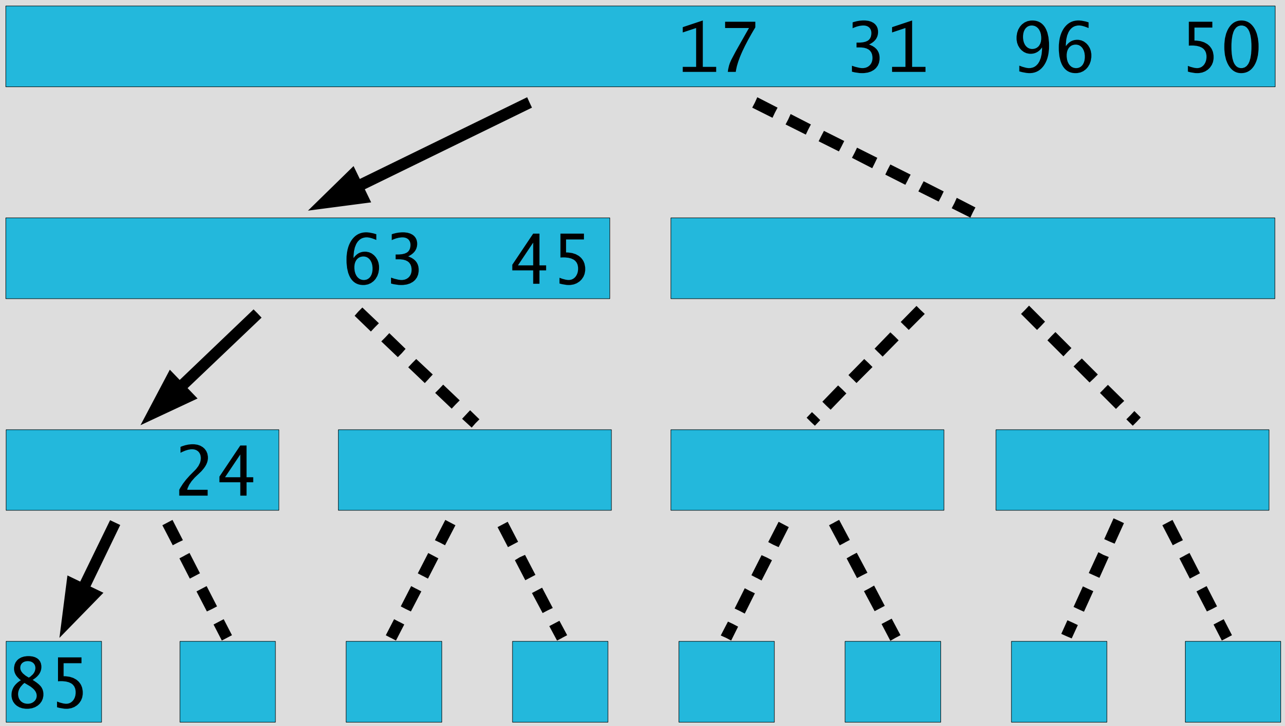
MergeSort Example/2



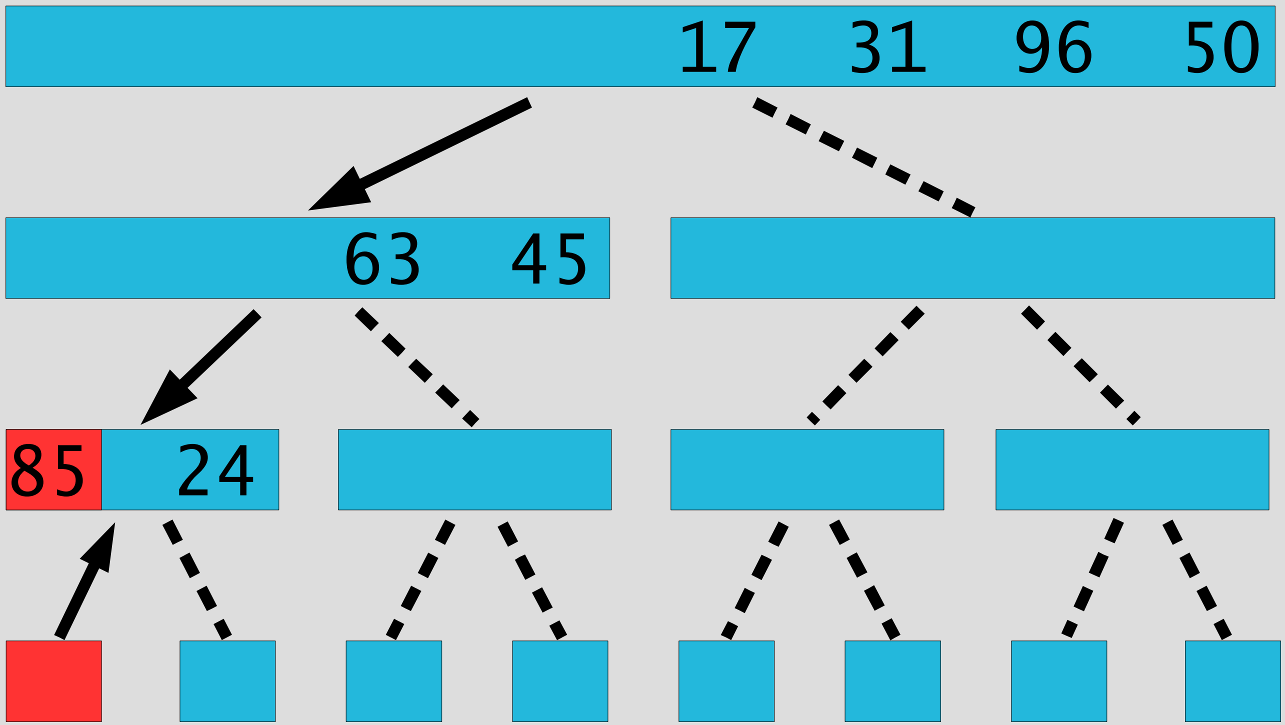
MergeSort Example/3



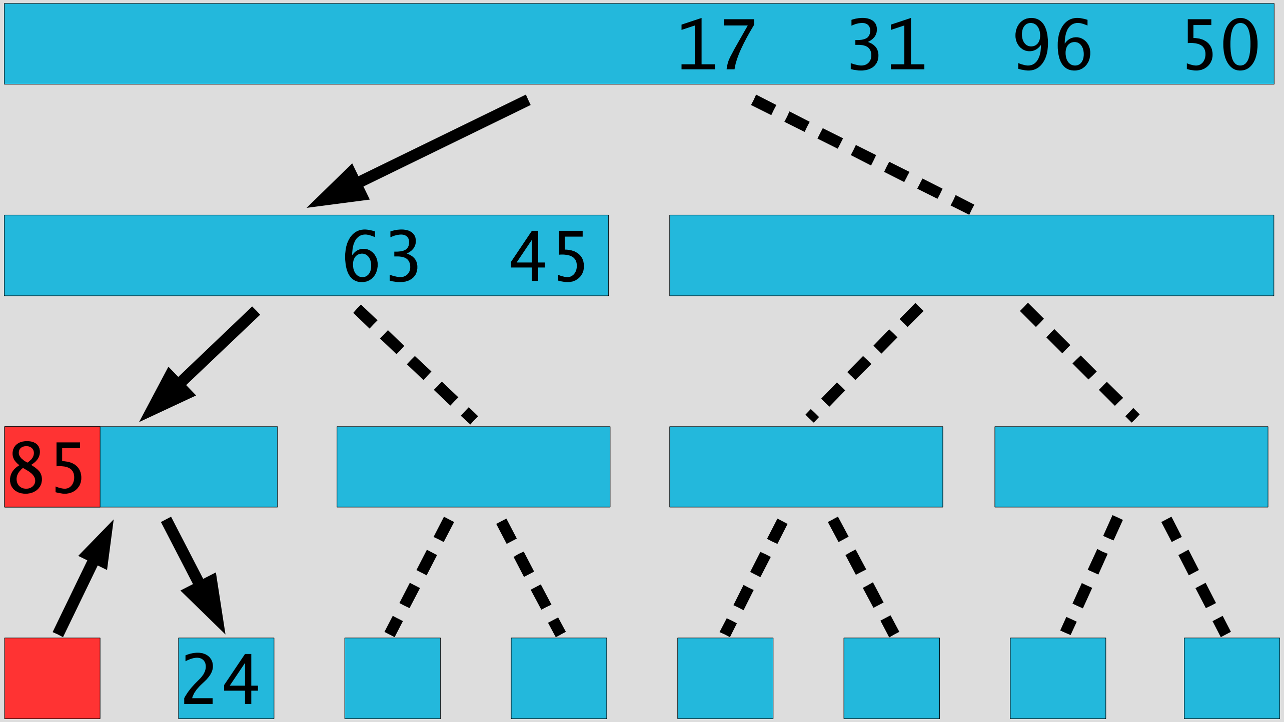
MergeSort Example/4



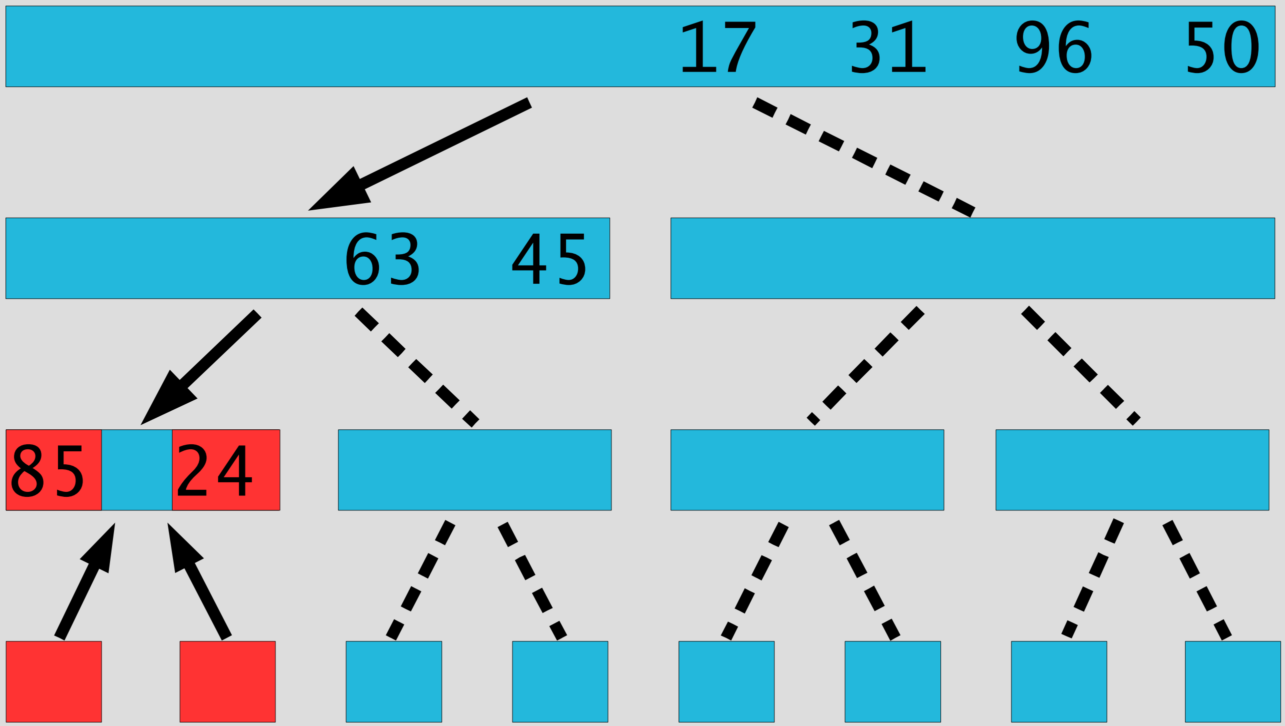
MergeSort Example/5



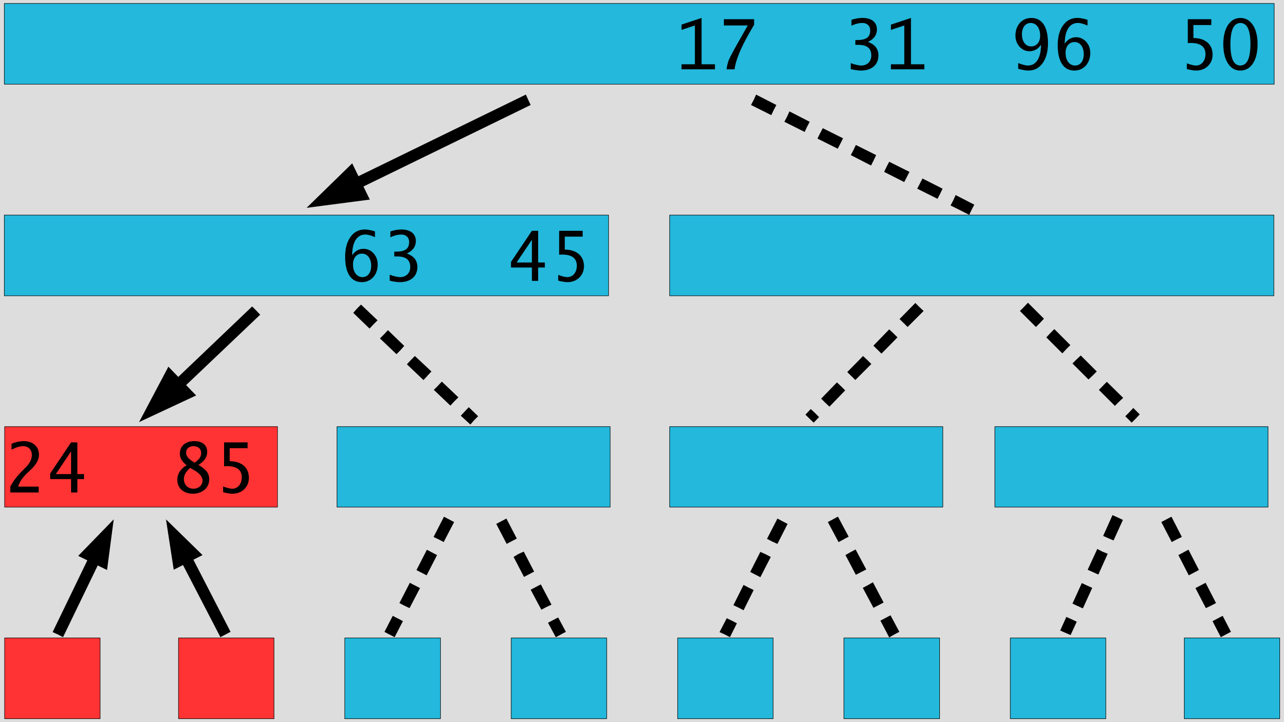
MergeSort Example/6



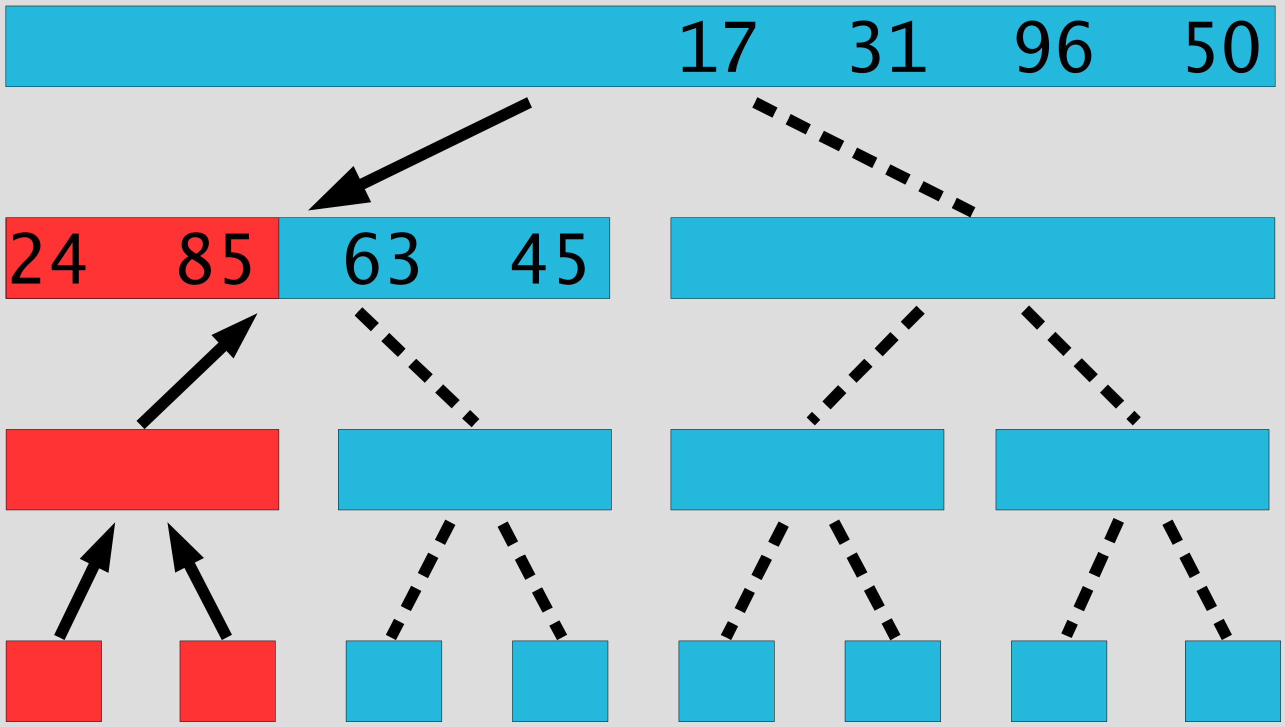
MergeSort Example/7



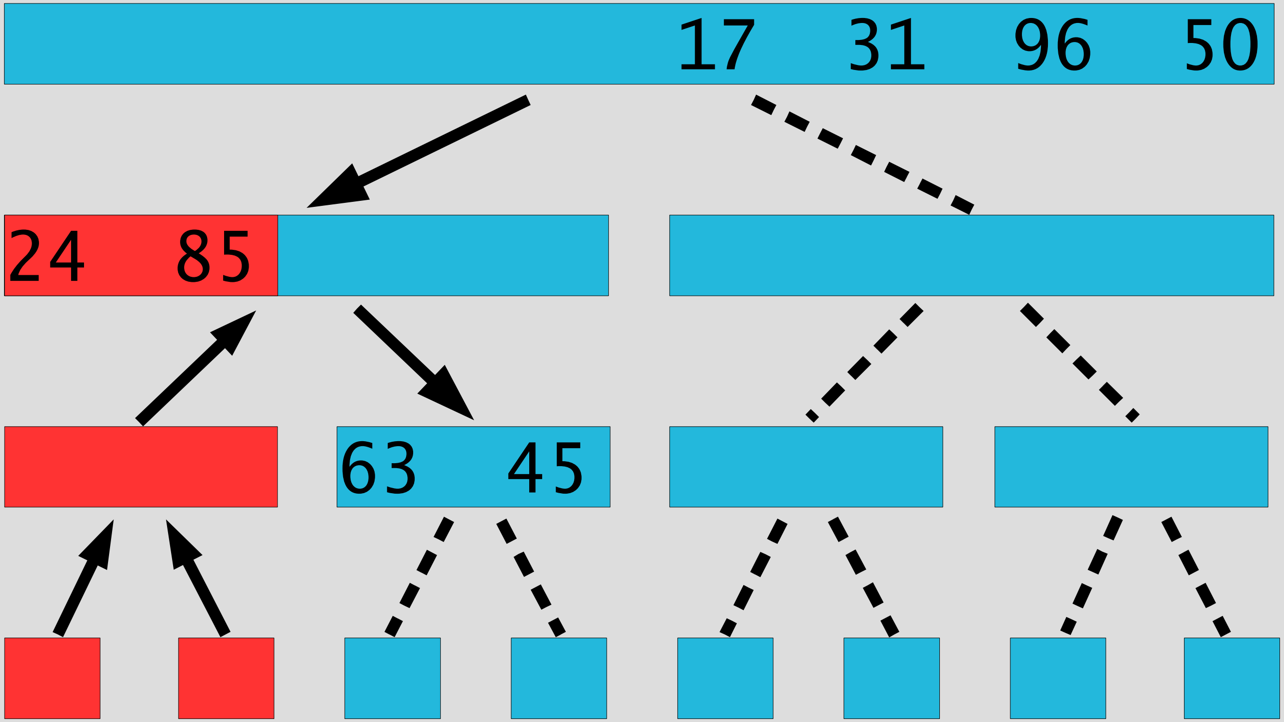
MergeSort Example/8



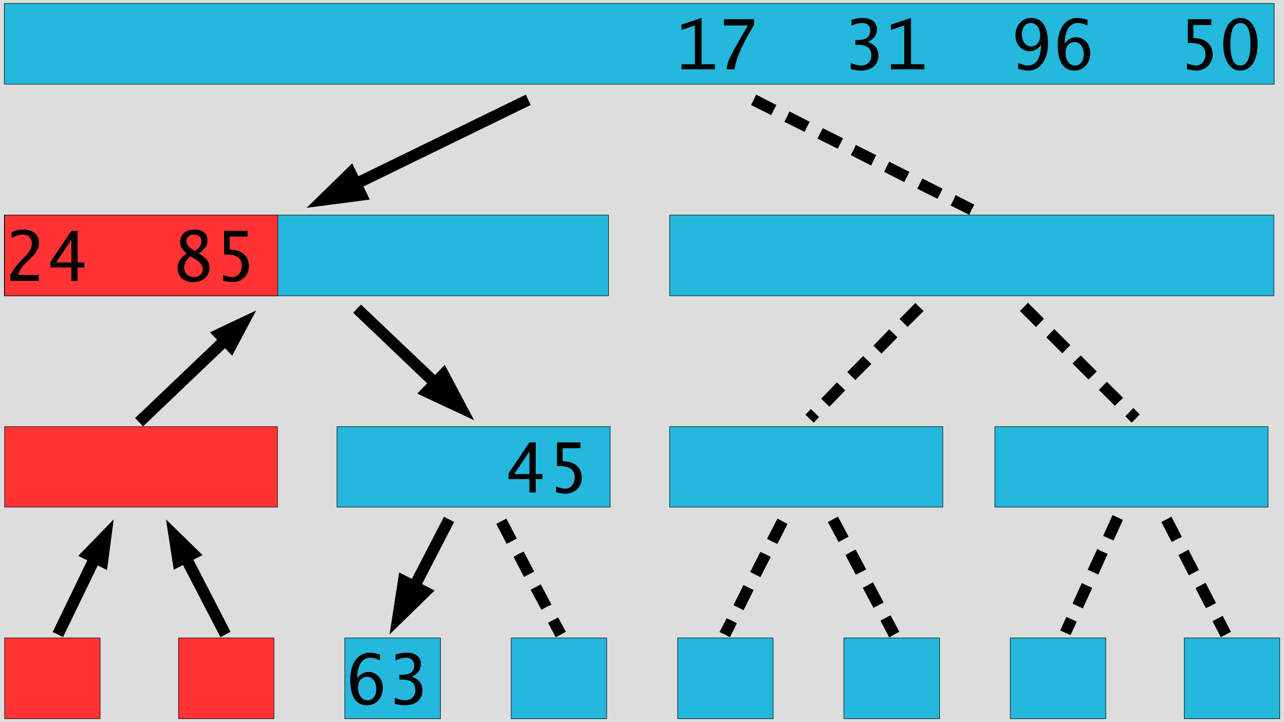
MergeSort Example/9



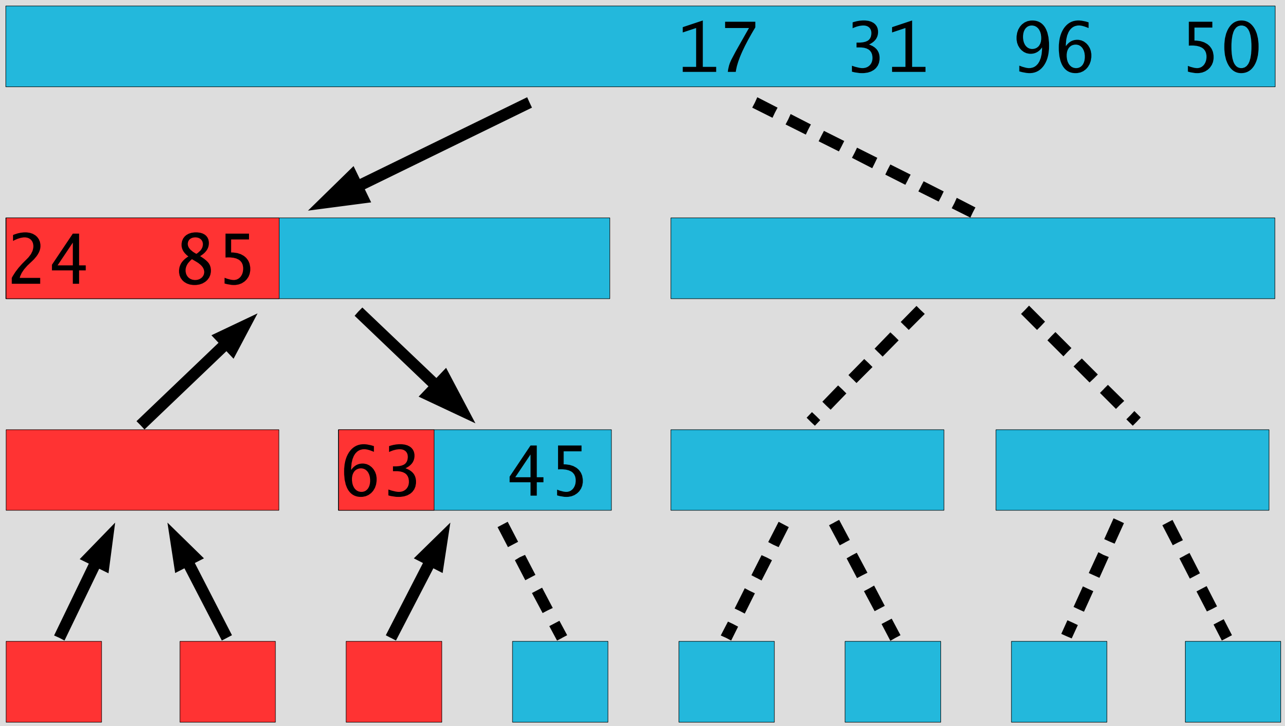
MergeSort Example/10



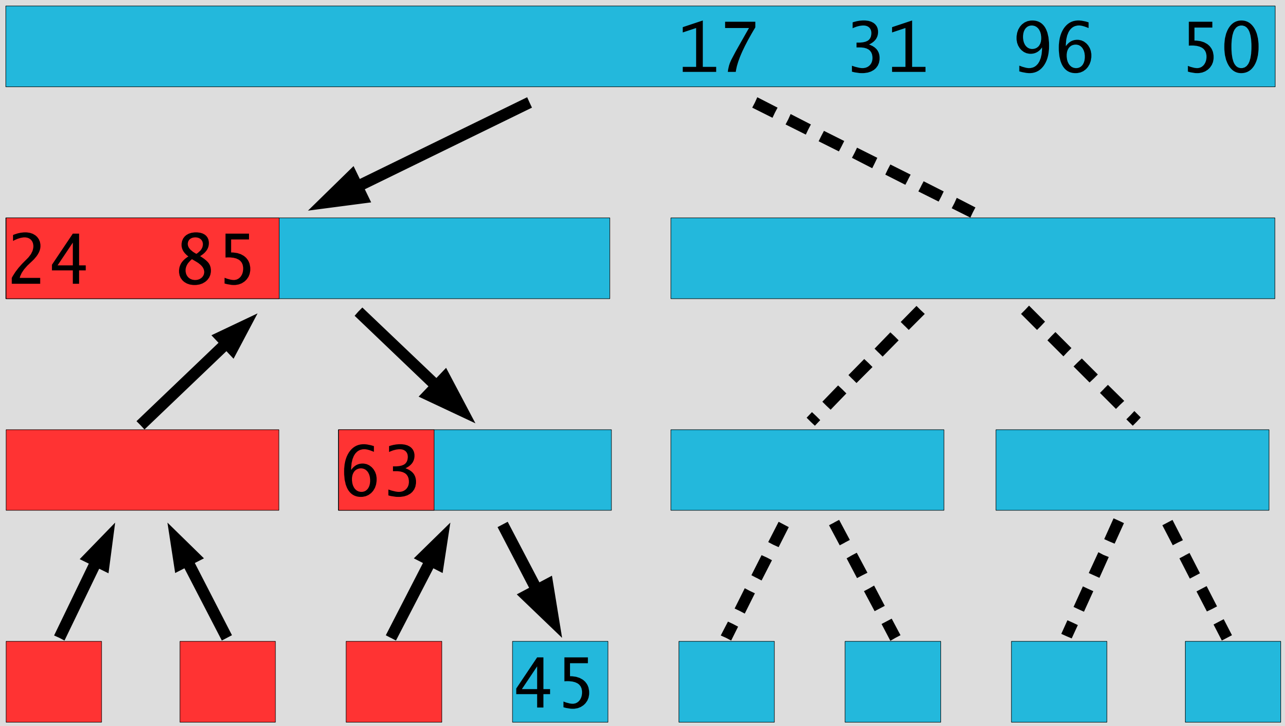
MergeSort Example/11



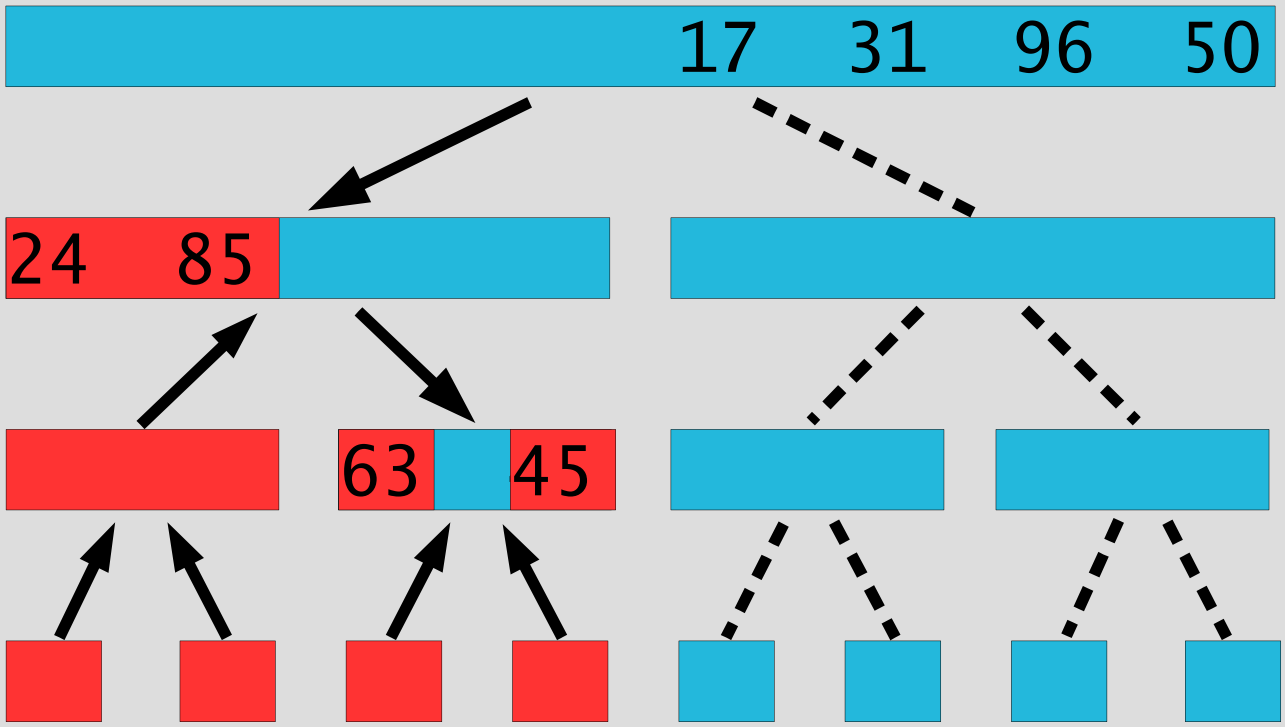
MergeSort Example/12



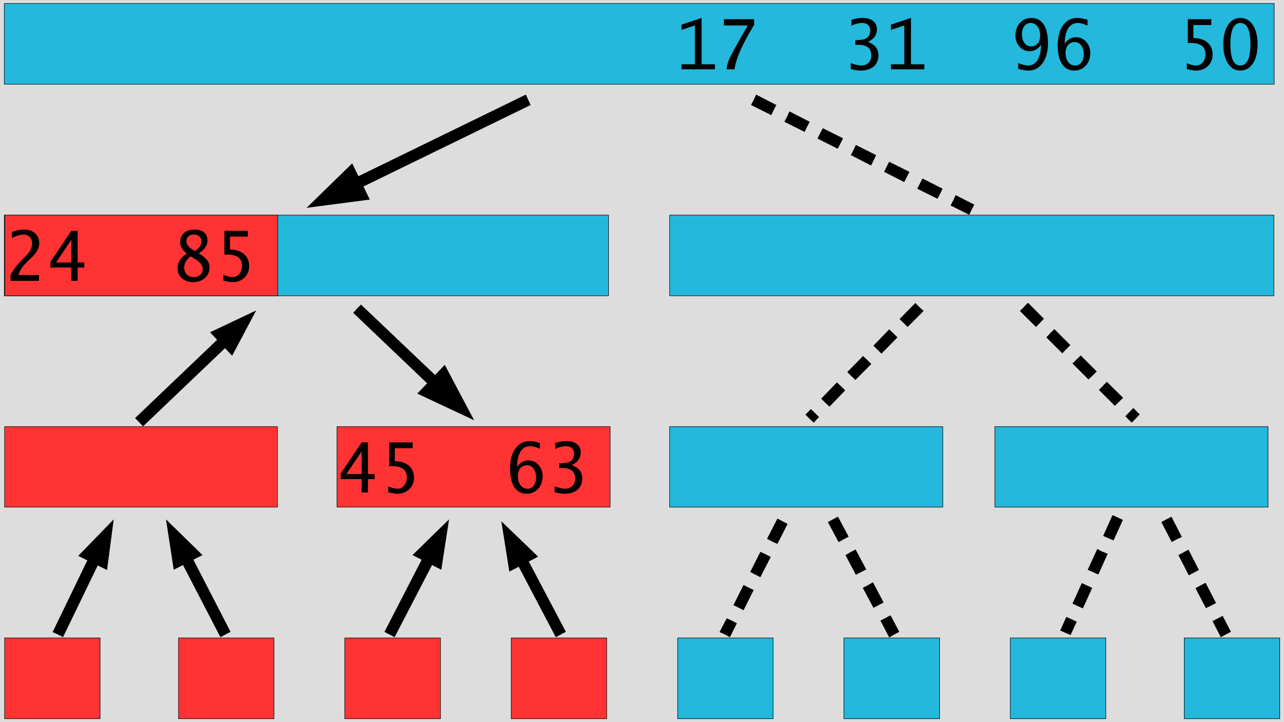
MergeSort Example/13



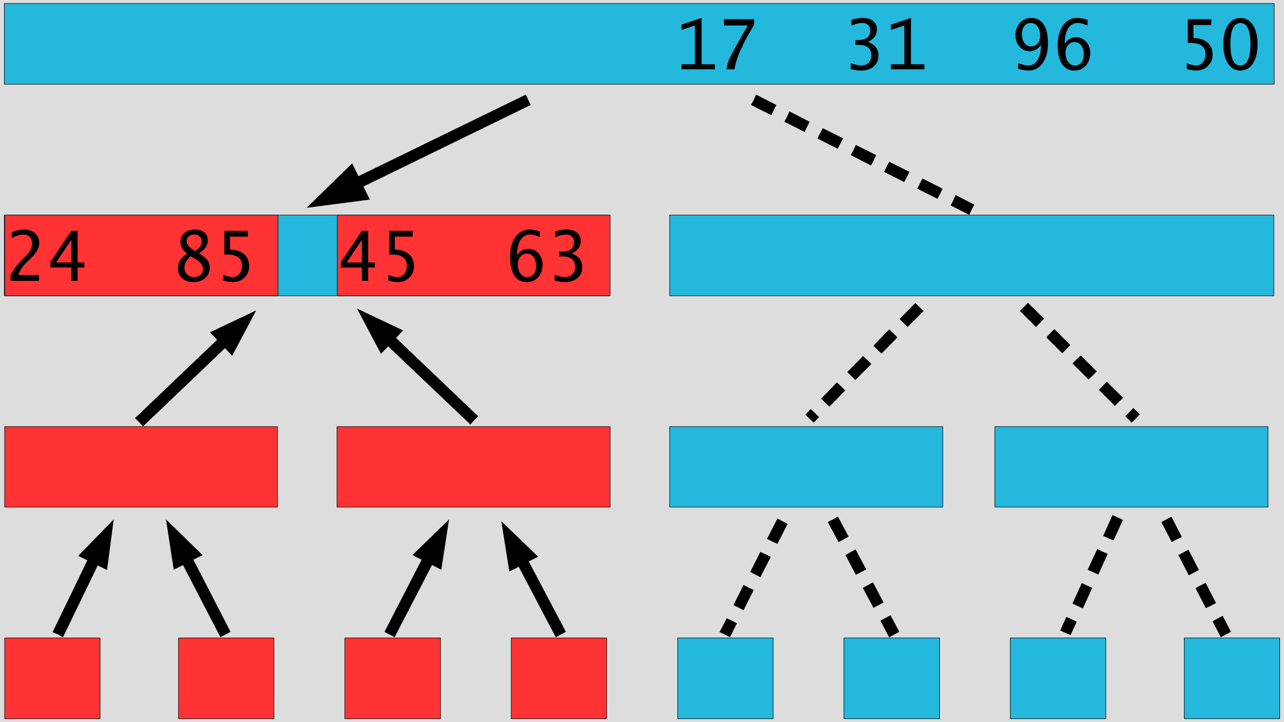
MergeSort Example/14



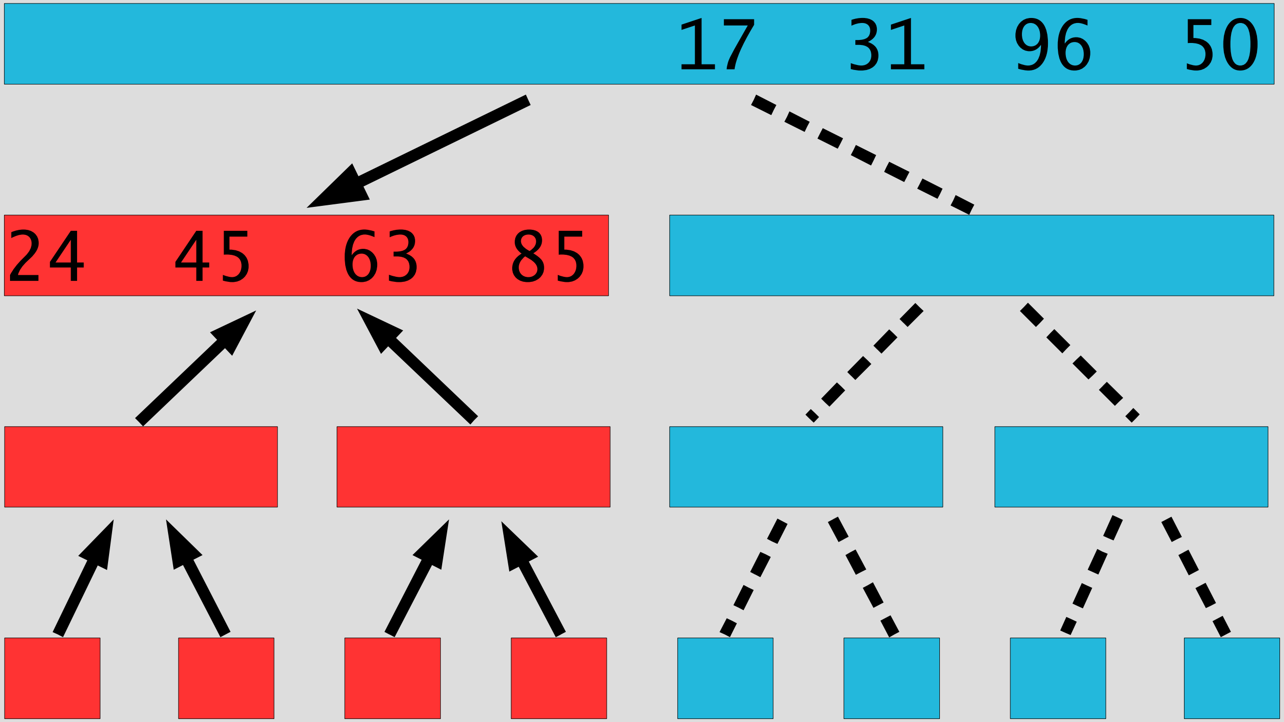
MergeSort Example/15



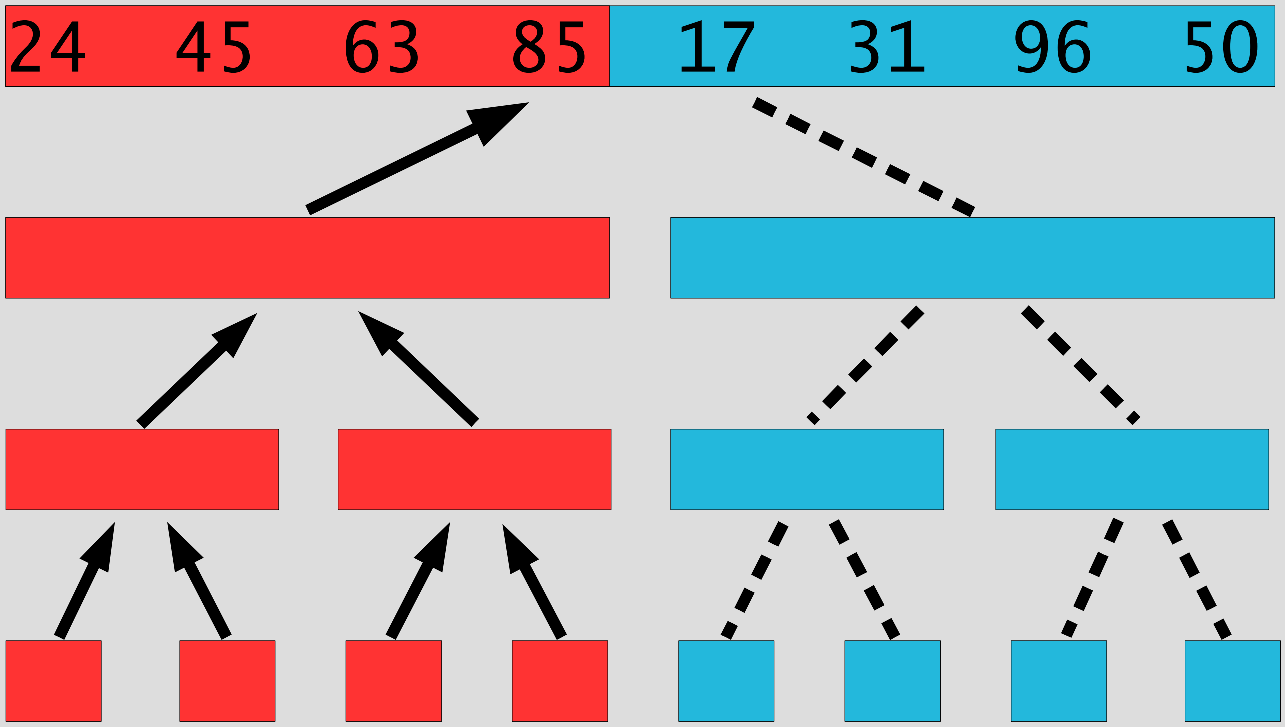
MergeSort Example/16



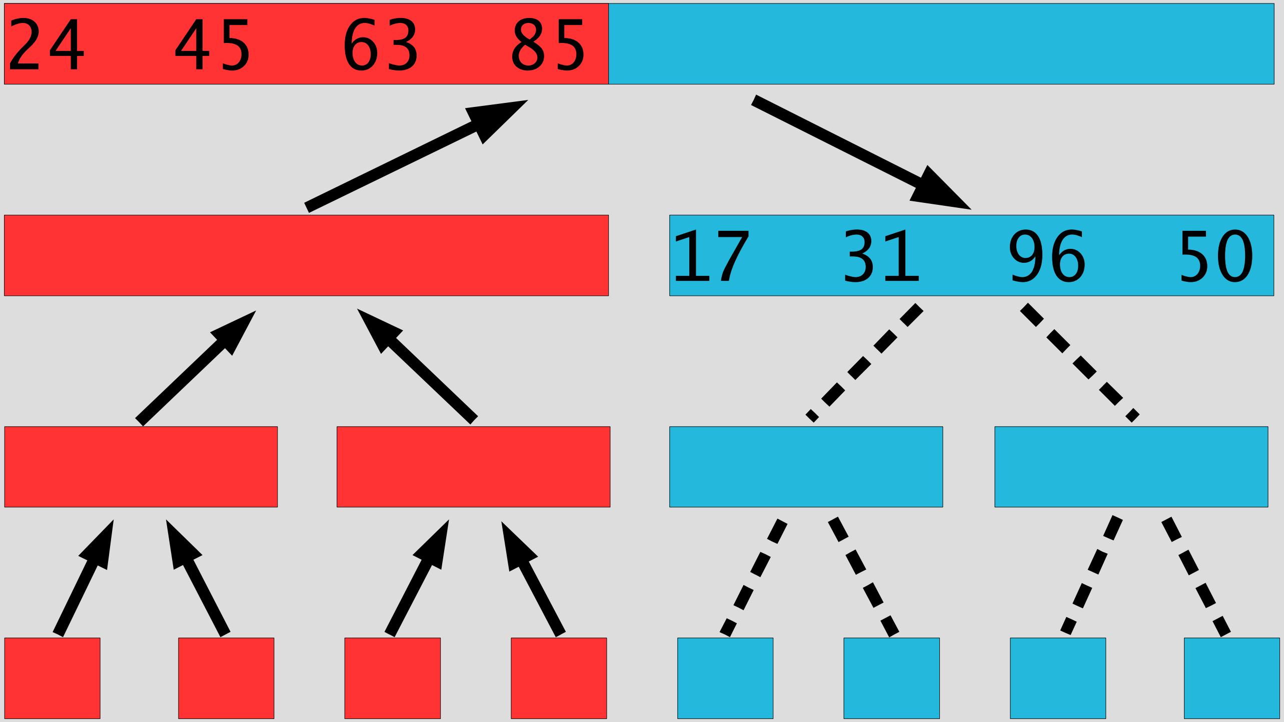
MergeSort Example/17



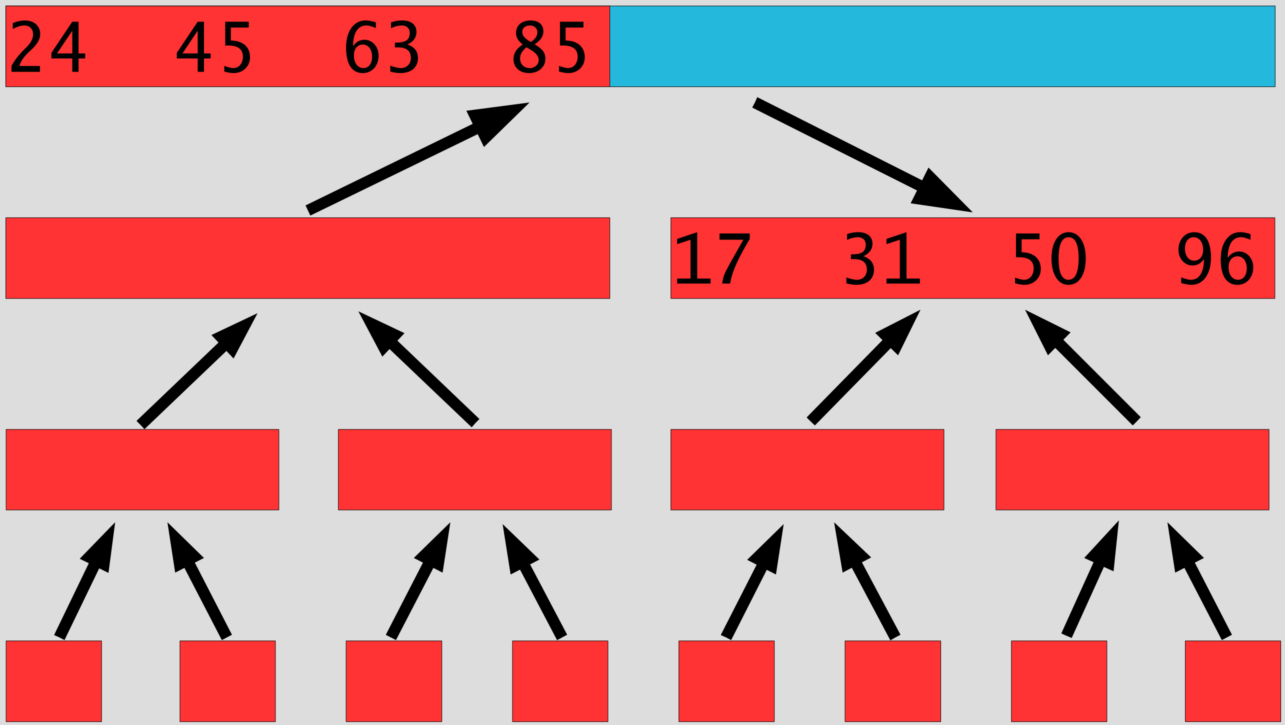
MergeSort Example/18



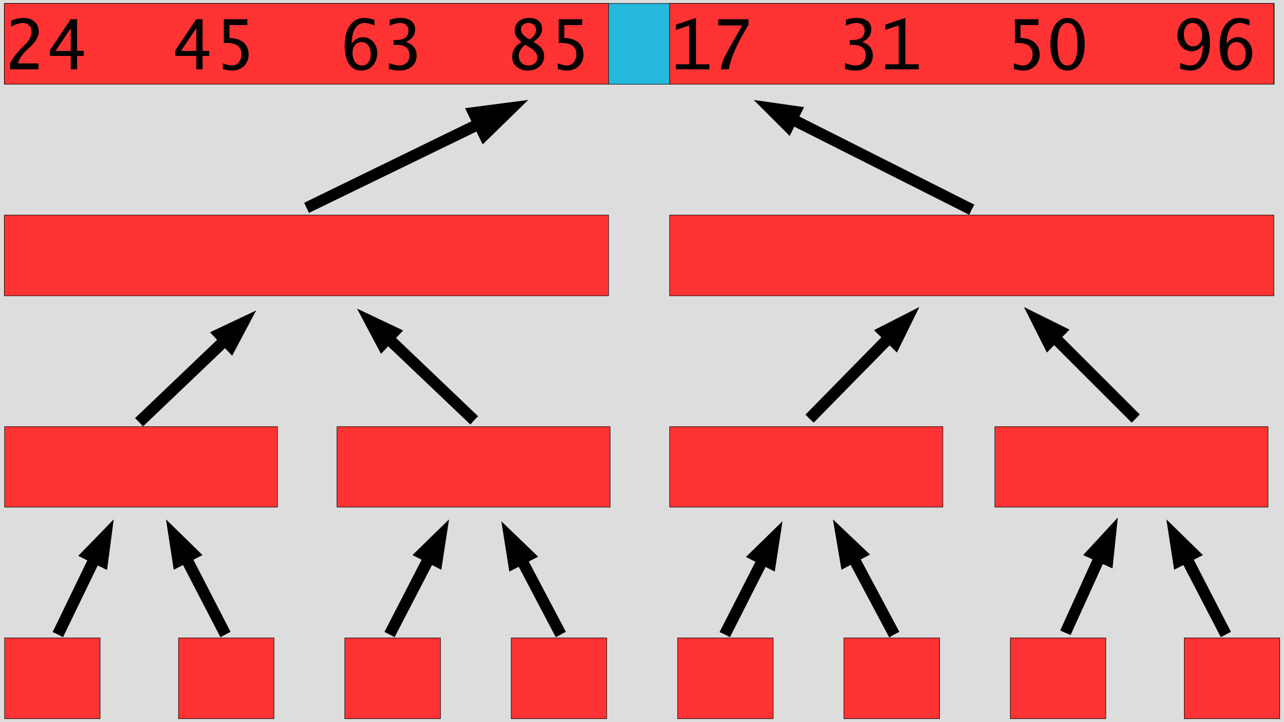
MergeSort Example/19



MergeSort Example/20

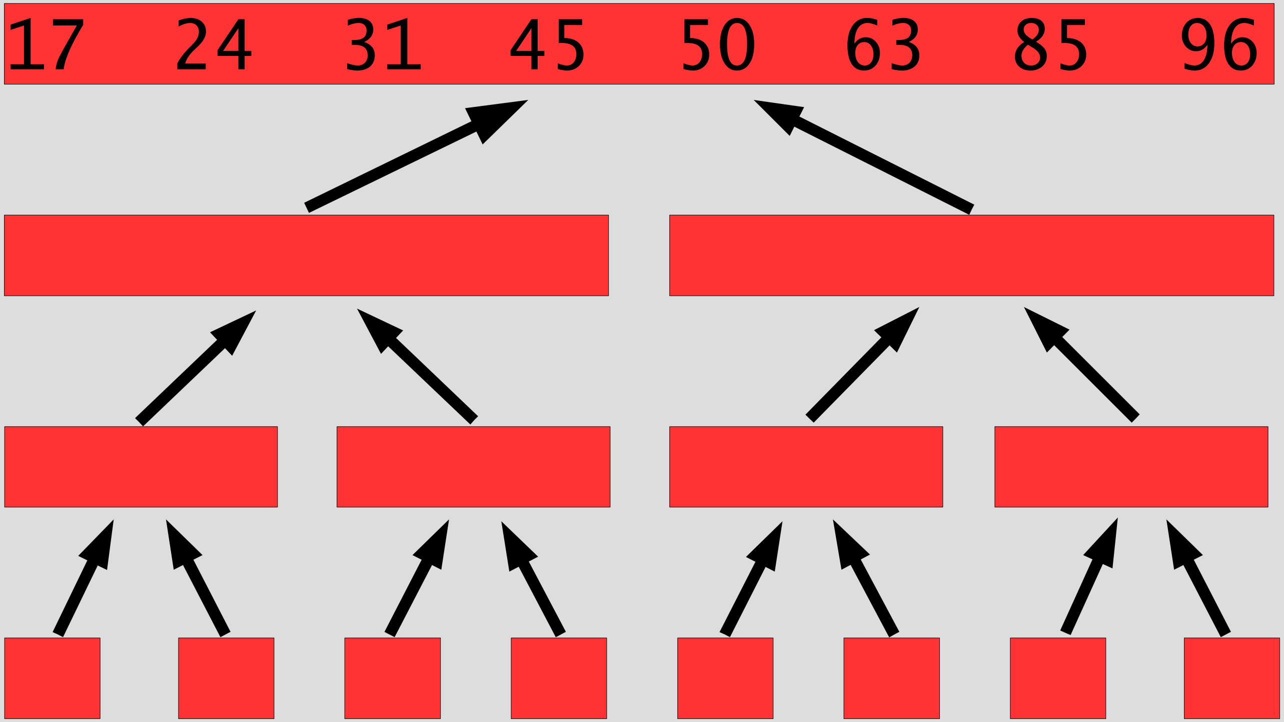


MergeSort Example/21



MergeSort Example/22

5



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, 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 MergeSort

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

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

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

Repeated Substitution Method

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

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

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

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

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

$$= n + n \log n$$

Suggested exercises

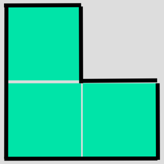
- Implement merge
- Implement mergeSort

Data Structures and Algorithms

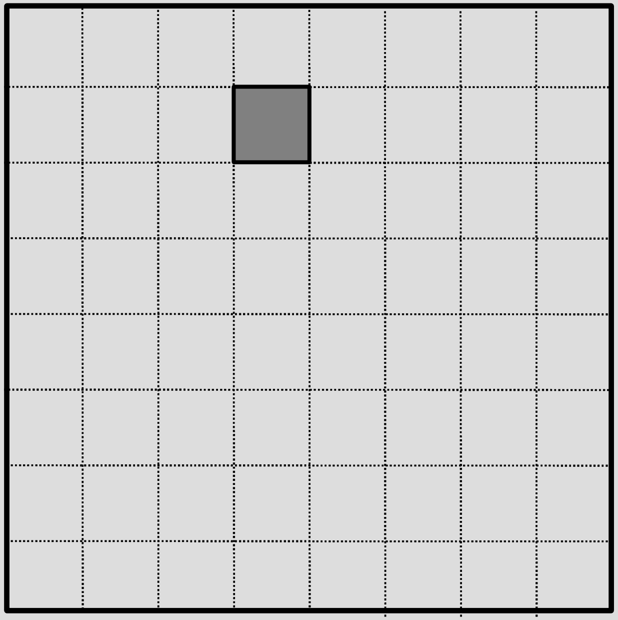
Week 3

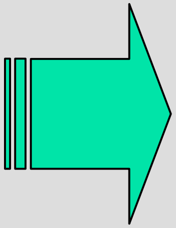
1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. Recurrences

Tiling

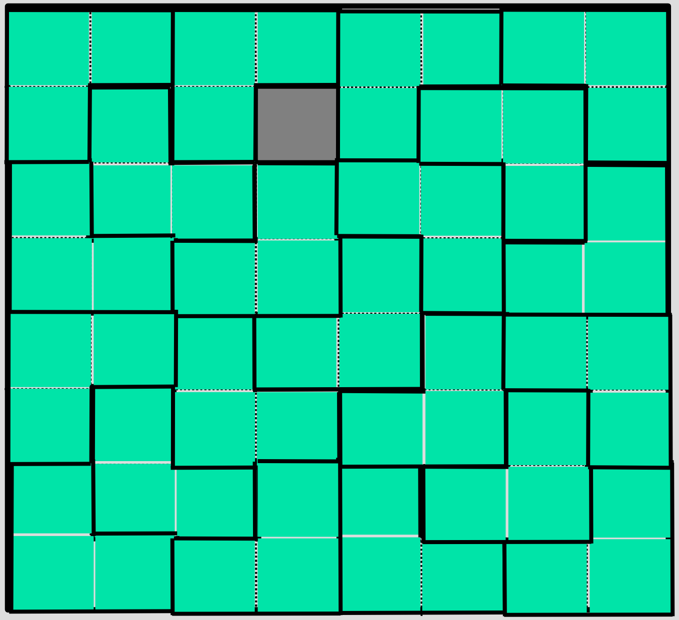
A tromino tile: 

A 2^nx2^n board with a hole:



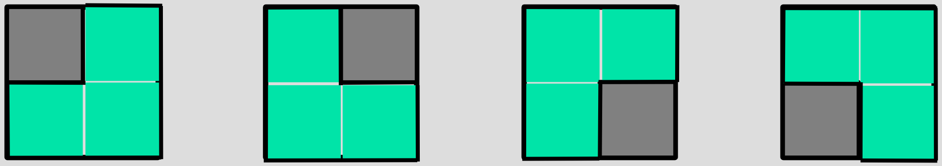


A tiling of the board with trominos:



Tiling: Trivial Case (n = 1)

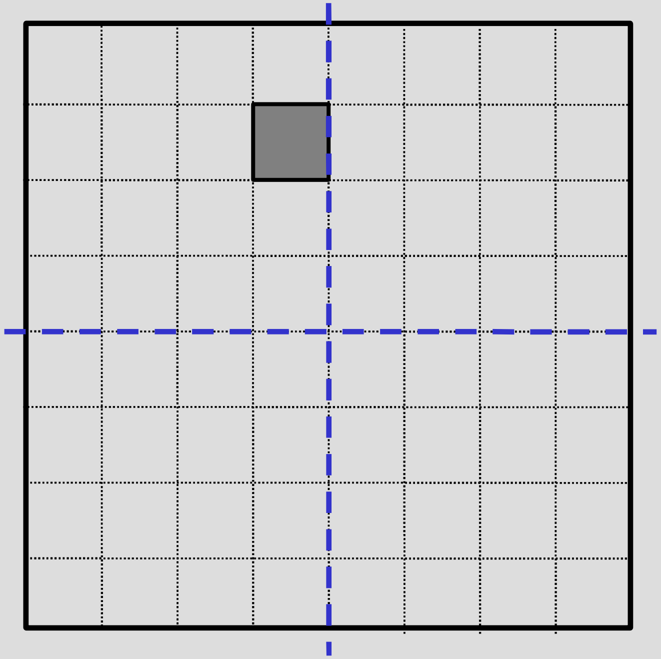
- Trivial case (n = 1): tiling a 2x2 board with a hole:



- 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

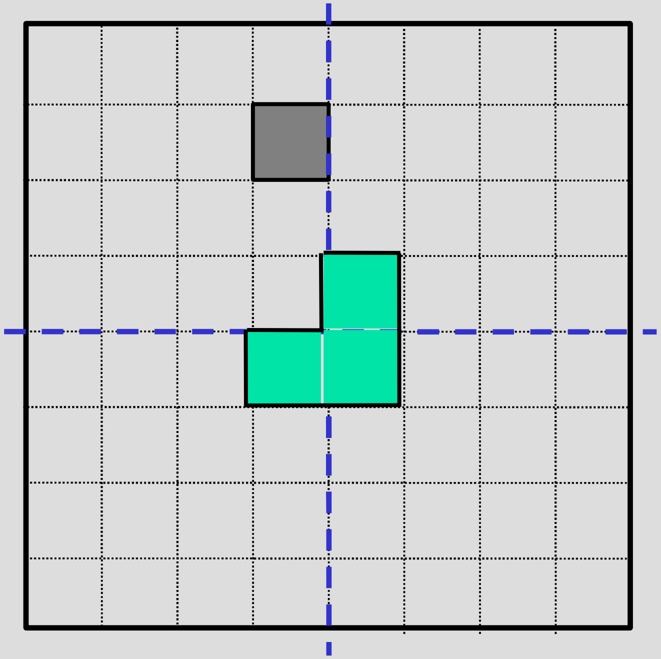
- To get smaller square boards let's divide the original board into four boards.



- Good: We have a problem of size 2^{n-1}x2^{n-1}!
- Bad: The other three problems are not similar to the original problem – they do not have holes!

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.

Data Structures and Algorithms

Week 3

1. Divide and conquer
2. Merge sort, repeated substitutions
3. Tiling
4. Recurrences

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

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

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 MergeSort

- 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

Analysis of MergeSort/2

$$T(n) = 2^i T(n/2^i) + 2in + 3\sum_{j=0}^{i-1} 2^j$$

An upper bound for i is log n

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

$$= 5n + 2n\log n - 3$$

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

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* $T(n) \le cn^3$ *by induction*

$$\begin{aligned} T(n) &= 4T(n/2) + n && \text{recurrence} \\ &\le 4c(n/2)^3 + n && \text{induction hypothesis} \\ &= 0.5cn^3 + n && \text{simplify} \\ &= cn^3 - (0.5cn^3 - n) && \text{rearrange} \\ &\le cn^3 \text{ if } c>=2 \text{ and } n>=1 && \end{aligned}$$

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

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

  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

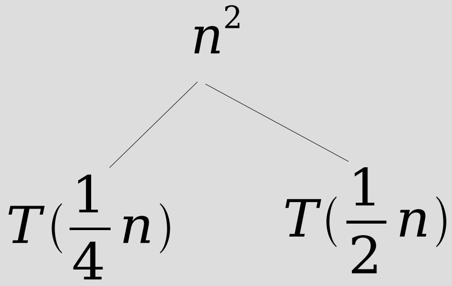
- Fixed proof: strengthen the inductive hypothesis by subtracting lower-order terms:

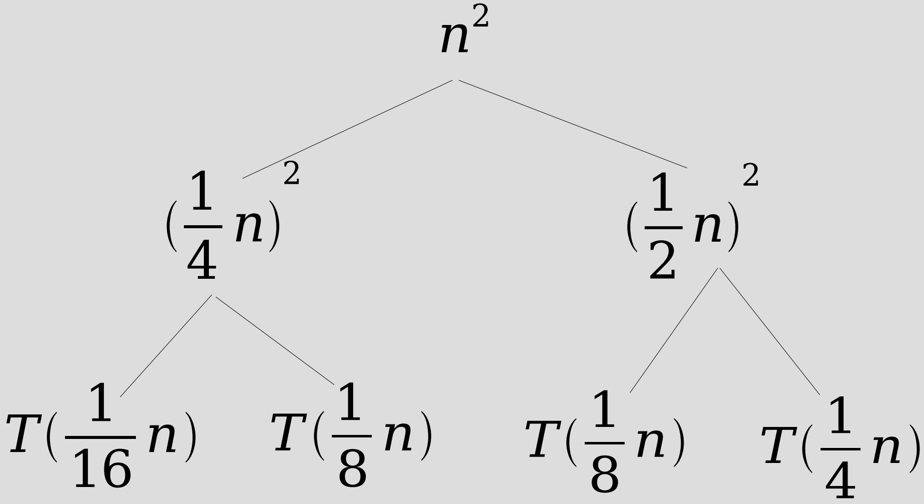
  Prove $T(n) \leq cn^2 - dn$ by induction

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

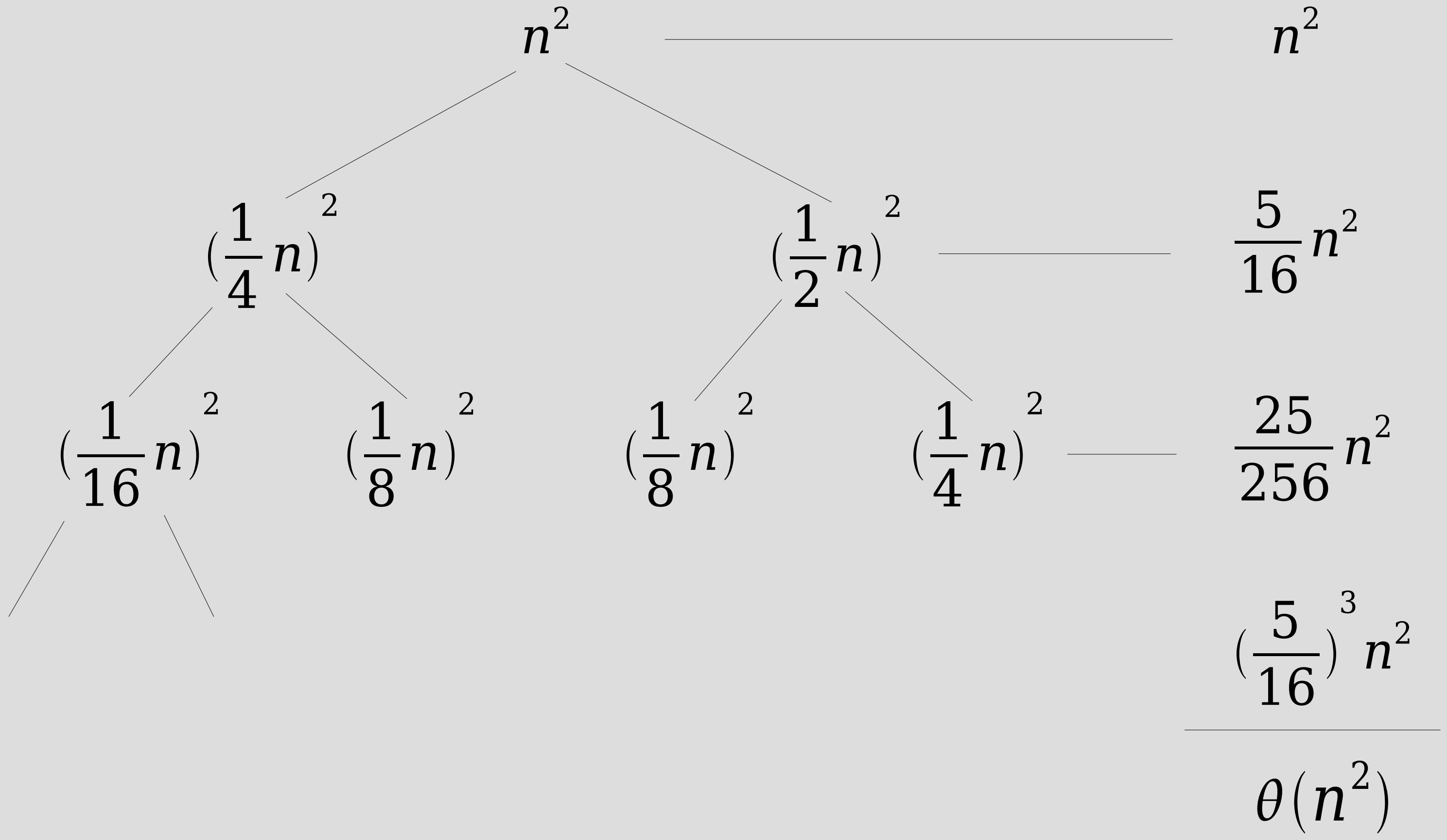
Recursion Tree

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated.
  - Each node labeled with $f(n/b^i)$
  - Helps guessing asymptotic solutions to recurrences





Recursion Tree/2



n^2 $\quad$ n^2

$(\frac{1}{4}n)^2$ $\quad$ $(\frac{1}{2}n)^2$ $\quad$ $\frac{5}{16}n^2$

$(\frac{1}{16}n)^2$ $\quad$ $(\frac{1}{8}n)^2$ $\quad$ $(\frac{1}{8}n)^2$ $\quad$ $(\frac{1}{4}n)^2$ $\quad$ $\frac{25}{256}n^2$

$(\frac{5}{16})^3 n^2$

$\theta(n^2)$

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 subproblems 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

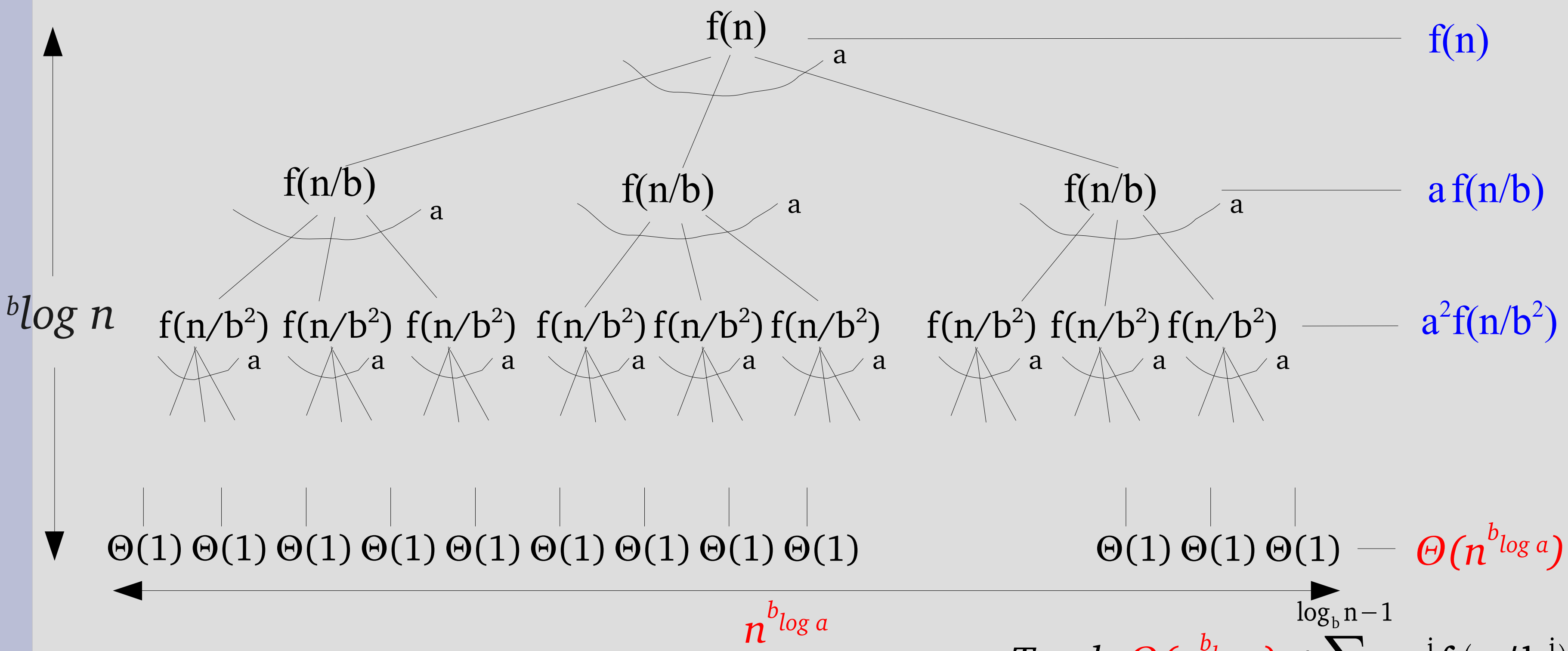
- Iterating the recurrence (expanding the tree) yields

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

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

- 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



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:
  1. Running time dominated by cost at leaves.
  2. Running time evenly distributed throughout the tree.
  3. 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 ε>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 (true for most functions of practical interest):

$$\exists c < 1 \text{ and } n_0 > 0 \text{ such that } af(n/b) \leq cf(n)\ \forall n > n_0$$

- **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. $f(n) = O(n^{{}^{b}\log a-\varepsilon})$

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

2. $f(n) = \Theta(n^{{}^{b}\log a})$

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

3. $f(n) = \Omega(n^{{}^{b}\log a+\varepsilon})$ and

   $af(n/b) \leq cf(n)$ for some $c<1$, $n>n_0$

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

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 Master Theorem 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: $af(n/b) <= cf(n)$

$af(n/b) = 3(n/4)\log(n/4) <=$

$(3/4)n \log n = cf(n)$ 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 $\forall j (1 \le j \le n)$: A[j] $\neq$ q
BinarySearchRec1(A, l, r, q):
   if l = r then
     if A[l] = q then return l else return NIL
   m := $\lfloor (l+r)/2 \rfloor$
   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: Binary Search

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

- 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 $\forall j(1 \le j \le n)$: A[j] $\neq$ q
BinarySearchRec2(A, l, r, q):
   if l = r then
      if A[l] = q then return l
      else return NIL
   m := $\lfloor (l+r)/2 \rfloor$
   if A[m] $\le$ 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

- Divide and conquer
- Merge sort
- Tiling
- Recurrences
  - repeated substitutions
  - substitution
  - master method
- Example recurrences: Binary search

Suggested exercises

- Using Master Method, compute (when possible) the bounds for $T(n) =$
  - $3T(n/3)+\log(n);3T(n/3)+n;3T(n/3)+n^2$
  - $4T(n/2)+n;4T(n/2)+n^2;4T(n/2)+n^3;$
  - $8T(n/2)+n^2\log(n);8T(n/2)+n^3;8T(n/2)+n^4;$
  - $T(2n/3)+1;T(2n/3)+n;$
  - $2T(3n/4)+n^2; 2T(3n/4)+n^3;$
  - $100T(n/100)+n;99T(n/100)+n;100T(n/99)+n$
  - $2T(n/2)+n/(1+\log(n));2T(n/2)+n\log(n)$
- See also exercises in CLRS

Suggested exercises/2

- Let $T(n) = aT(n/b)+n^c(\log(n))^{d.}$ Implement a java program taking the values a,b,c,d of the recurrence and printing:
  - The values computed recursively
  - The values computed as case 1,2,3
- plot the relative values for n = 1..100
- Try it with the examples of previous page

Next Week

- Sorting
  - HeapSort
  - QuickSort