A design paradigm

Divide and conquer:

(When) does decomposing a problem into smaller parts help?

Multiplying complex numbers
(from Jeff Edmonds' slides)

INPUT: Two pairs of integers, (a,b), (c,d) representing complex numbers, a+ib, c+id, respectively.
OUTPUT: The pair [(ac-bd),(ad+bc)] representing the product (ac-bd) + i(ad+bc)

Naïve approach: 4 multiplications, 2 additions.
Suppose a multiplication costs $1 and an addition cost a penny. The naïve algorithm costs $4.02.

Q: Can you do better?

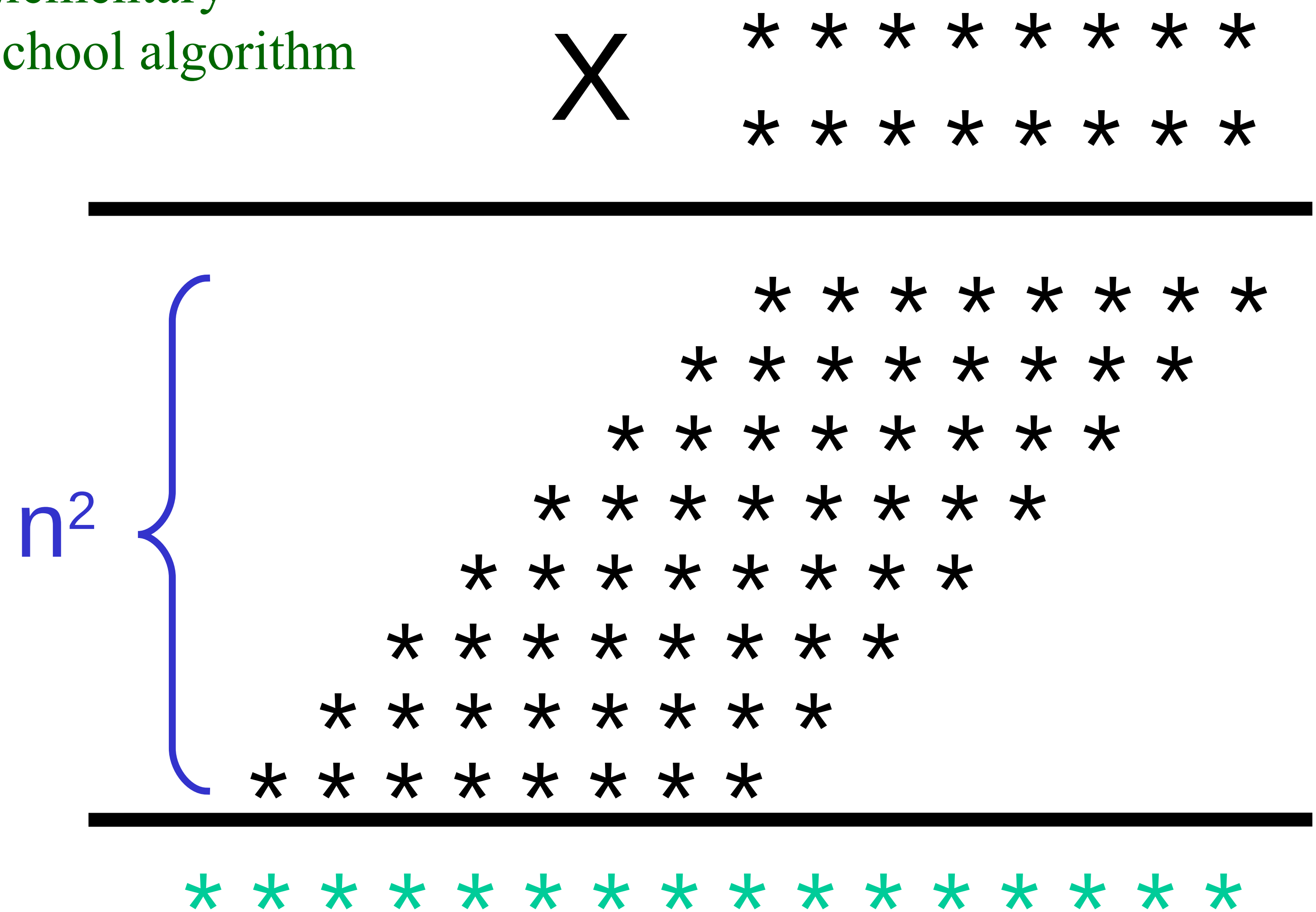
Gauss' idea

- $m_1 = ac$
- $m_2 = bd$
- $A_1 = m_1 – m_2 = ac-bd$
- $m_3 = (a+b)(c+d) = ac + ad + bc + bd$
- $A_2 = m_3 – m_1 – m_2 = ad+bc$
- **Saves 1 multiplication! Uses more additions. The cost now is $3.03.**
- This is good (saves 25% multiplications), but it leads to more dramatic asymptotic improvement elsewhere! **(aside: look for connections to known algorithms)**

Q: How fast can you multiply two n-bit numbers?

How to multiply two n-bit numbers.

Elementary School algorithm



How to multiply two n-bit numbers - contd.

Elementary
School algorithm

```
      * * * * * * * *
  X   * * * * * * * *
-------------------------------
* * * * * * * * * * * * * * * *
-------------------------------
```

Q: Is there a faster algorithm?

A: YES! Use divide-and-conquer.

Divide and Conquer

Intuition:

- **DIVIDE** my instance to the problem into smaller instances to the same problem.
- Recursively solve them.
- **GLUE** the answers together so as to obtain the answer to your larger instance.
- Sometimes the last step may be trivial.

Multiplication of two n-bit numbers

- X =
- Y =

a	b
c	d

- $X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$
- $XY = ac\,2^{n} + (ad+bc)\,2^{n/2} + bd$

MULT(X,Y):

If |X| = |Y| = 1 then RETURN XY

Break X into a;b and Y into c;d

RETURN

MULT(a,c) 2^n + (MULT(a,d) + MULT(b,c)) $2^{n/2}$ + MULT(b,d)

Time complexity of MULT

- T(n) = time taken by MULT on two n-bit numbers
- What is T(n)? Is it $\theta(n^2)$?
- Hard to compute directly
- Easier to express as a **recurrence relation**!
- T(1) = k for some constant k
- $T(n) = 4\,T(n/2) + c_1 n + c_2$ for some constants c_1 and c_2
- How can we get a θ() expression for T(n)?

MULT(X,Y):

If |X| = |Y| = 1 then RETURN XY

Break X into a;b and Y into c;d

RETURN

$MULT(a,c)\,2^n + (MULT(a,d) + MULT(b,c))\,2^{n/2} + MULT(b,d)$

Time complexity of MULT

Make it concrete

- $T(1) = 1$
- $T(n) = 4\, T(n/2) + n$

<u>Technique 1:</u> Guess and verify

$T(n) = 2n^2 - n$

Holds for n=1

$T(n) = 4\, (2(n/2)^2 - n/2 + n)$

$\quad = 2n^2 - n$

Time complexity of MULT

- $T(1) = 1$ & $T(n) = 4\,T(n/2) + n$

Technique 2: Expand recursion

$T(n) = 4\,T(n/2) + n$

$= 4\,(4T(n/4) + n/2) + n = 4^2T(n/4) + n + 2n$

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

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

$= \ldots\ldots\ldots$

$= 4^kT(1) + n + 2n + 4n + \ldots + 2^{k-1}n$ where $2^k = n$

GUESS

$= n^2 + n\,(1 + 2 + 4 + \ldots + 2^{k-1})$

$= n^2 + n\,(2^k - 1)$

$= 2\,n^2 - n$ [NOT FASTER THAN BEFORE]

Gaussified MULT (Karatsuba 1962)

MULT(X,Y):

If |X| = |Y| = 1 then RETURN XY

Break X into a;b and Y into c;d

e = MULT(a,c) and f =MULT(b,d)

RETURN $e2^n$ + (MULT(a+b, c+d) – e - f) $2^{n/2}$ + f

- $T(n) = 3\ T(n/2) + n$
- Actually: $T(n) = 2\ T(n/2) + T(n/2 + 1) + kn$

Time complexity of Gaussified MULT

- $T(1) = 1$ & $T(n) = 3\,T(n/2) + n$

Technique 2: Expand recursion

$T(n) = 3\,T(n/2) + n$

$= 3\,(3T(n/4) + n/2) + n = 3^2T(n/4) + n + 3/2n$

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

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

$= \ldots\ldots\ldots$

$= 3^kT(1) + n + 3/2n + (3/2)^2n + \ldots + (3/2)^{k-1}n$ where $2^k = n$

$= 3^{\log_2 n} + n(1 + 3/2 + (3/2)^2 + \ldots + (3/2)^{k-1})$

$= n^{\log_2 3} + 2n\,((3/2)^k - 1)$

$= n^{\log_2 3} + 2n\,(n^{\log_2 3}/n - 1)$

$= 3n^{\log_2 3} - 2n$

Not just 25% savings!

$\theta(n^2)$ vs $\theta(n^{1.58..})$

Multiplication Algorithms

Kindergarten ? 3*4=3+3+3+3	$n2^n$ Show
Grade School	n^2
Karatsuba	$n^{1.58\ldots}$
Fastest Known	n logn loglogn

Next…

1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 2 of the text.
2. Next, Strassen's algorithm for matrix multiplication
3. Later: more design and conquer algorithms: MergeSort. Solving recurrences and the Master Theorem.

Matrix multiplication

- Fundamental operation in Linear Algebra
- Used for numerical differentiation, integration, optimization etc

Naïve matrix multiplication

SimpleMatrixMultiply (A,B)

```
n ← A.rows
C ← CreateMatrix(n,n)
for i ← 1 to n
  for j ← 1 to n
      C[i,j] ← 0
      for k ← 1 to n
            C[i,j] ← C[i,j] + A[i,k]*B[k,j]
return C
```

- Argue that the running time is $\theta(n^3)$

Faster Algorithm?

- Idea: Similar to multiplication in N, C
- Divide and conquer approach provides unexpected improvements

First attempt and Divide & Conquer

Divide A,B into 4 n/2 x n/2 matrices

- $C_{11} = A_{11} B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11} B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21} B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21} B_{12} + A_{22}B_{22}$

Simple Recursive implementation. Running time is given by the following recurrence.

- $T(1) = C$, and for $n>1$
- $T(n) = 8T(n/2) + \theta(n^2)$
- $\theta(n^3)$ time-complexity

Strassen's algorithm

Avoid one multiplication (details on page 80)
(but uses more additions)

Recurrence:

- $T(1) = C$, and for $n>1$
- $T(n) = 7T(n/2) + \theta(n^2)$

- How can we solve this?
- Will see that $T(n) = \theta(n^{\lg 7})$, $\lg 7 =$ **2.8073….**

The maximum-subarray problem

- Given an array of integers, find a contiguous subarray with the maximum sum.
- Very naïve algorithm:
- Brute force algorithm:
- At best, $\theta(n^2)$ time complexity

Can we do divide and conquer?

- Want to use answers from left and right half subarrays.
- Problem: The answer may not lie in either!

- Key question: What information do we need from (smaller) subproblems to solve the big problem?

- Related question: how do we get this information?

A divide and conquer algorithm

Algorithm in Ch 4.1:

Recurrence:

- $T(1) = C$, and for $n>1$
- $T(n) = 2T(n/2) + \theta(n)$

- $T(n) = \theta(n \log n)$

More divide and conquer : Merge Sort

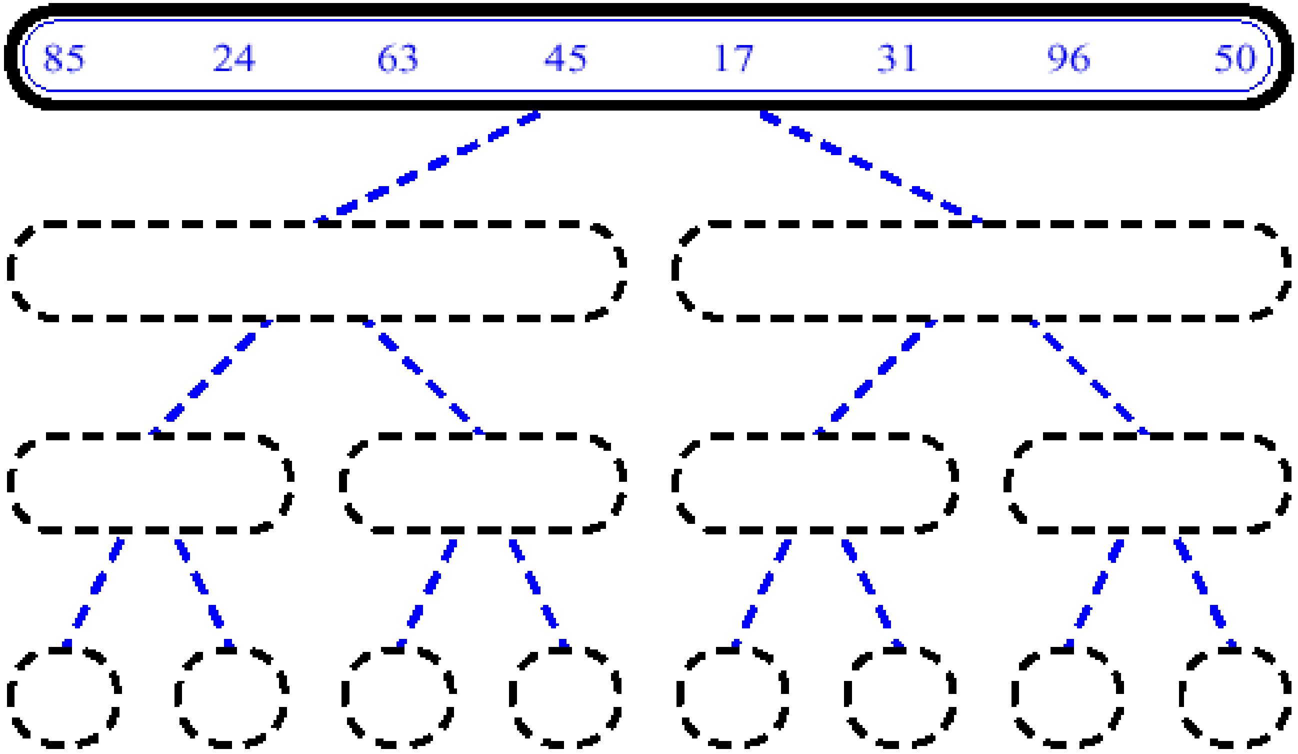
- **Divide**: If *S* has at least two elements (nothing needs to be done if *S* has zero or one elements), remove all the elements from *S* and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 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

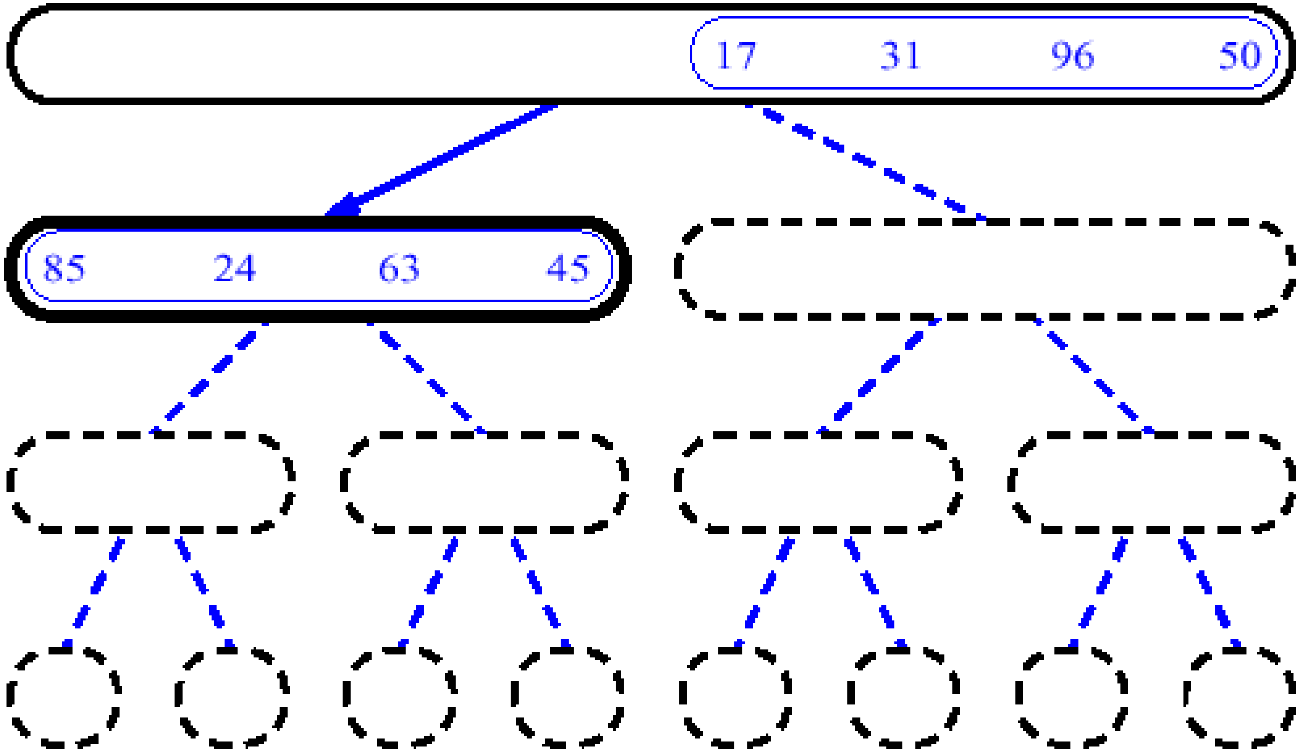
```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

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

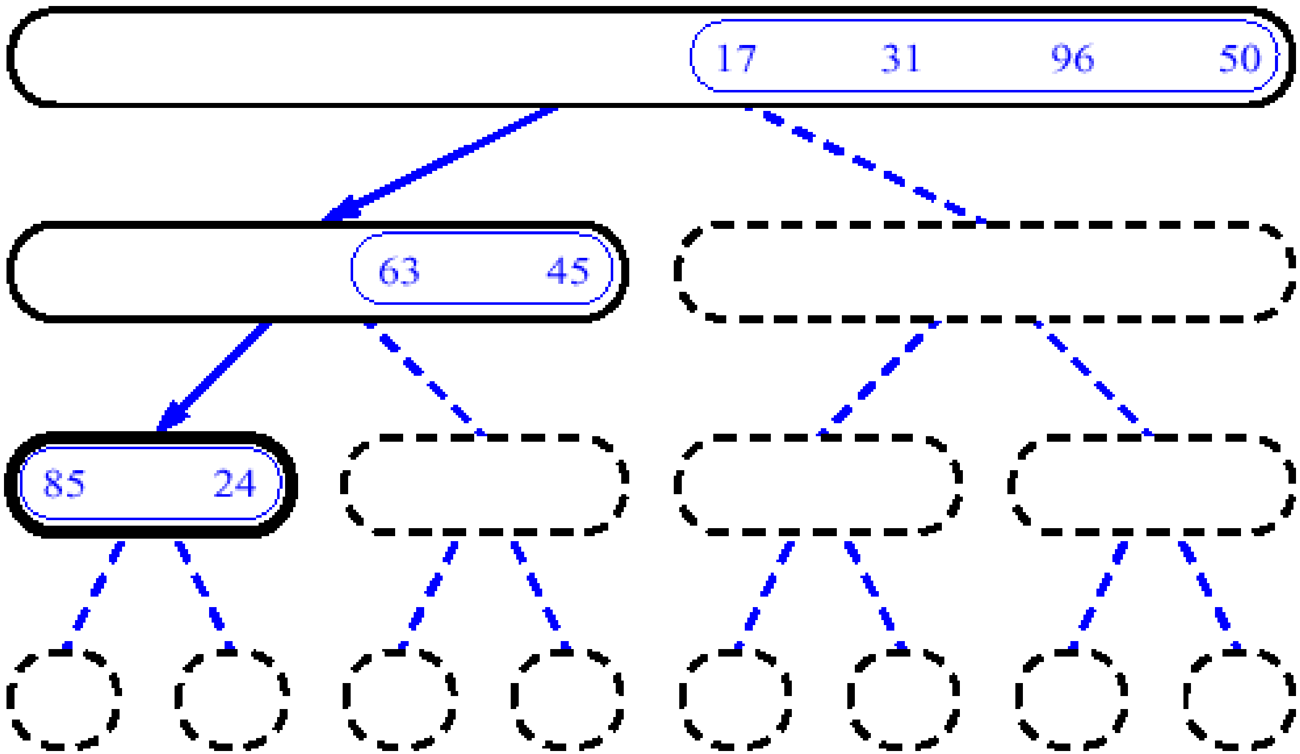
Merge Sort: example



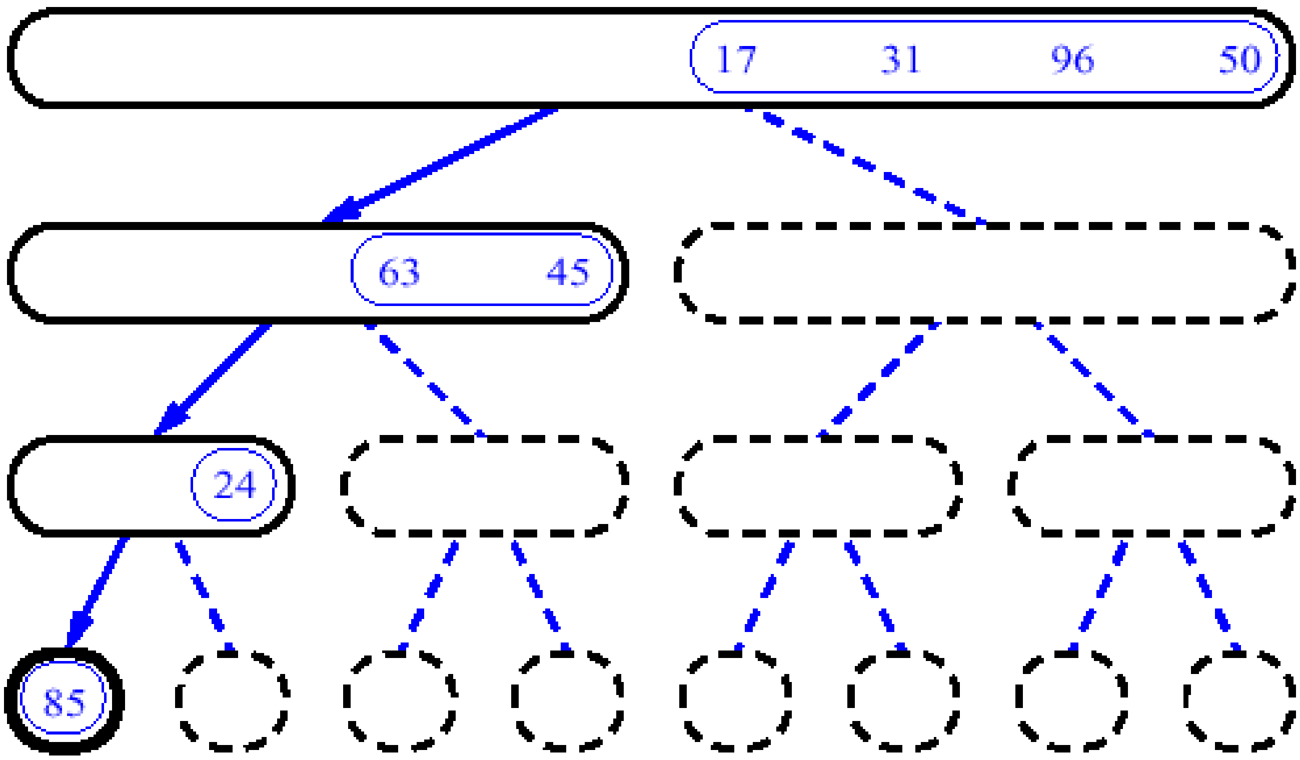
Merge Sort: example



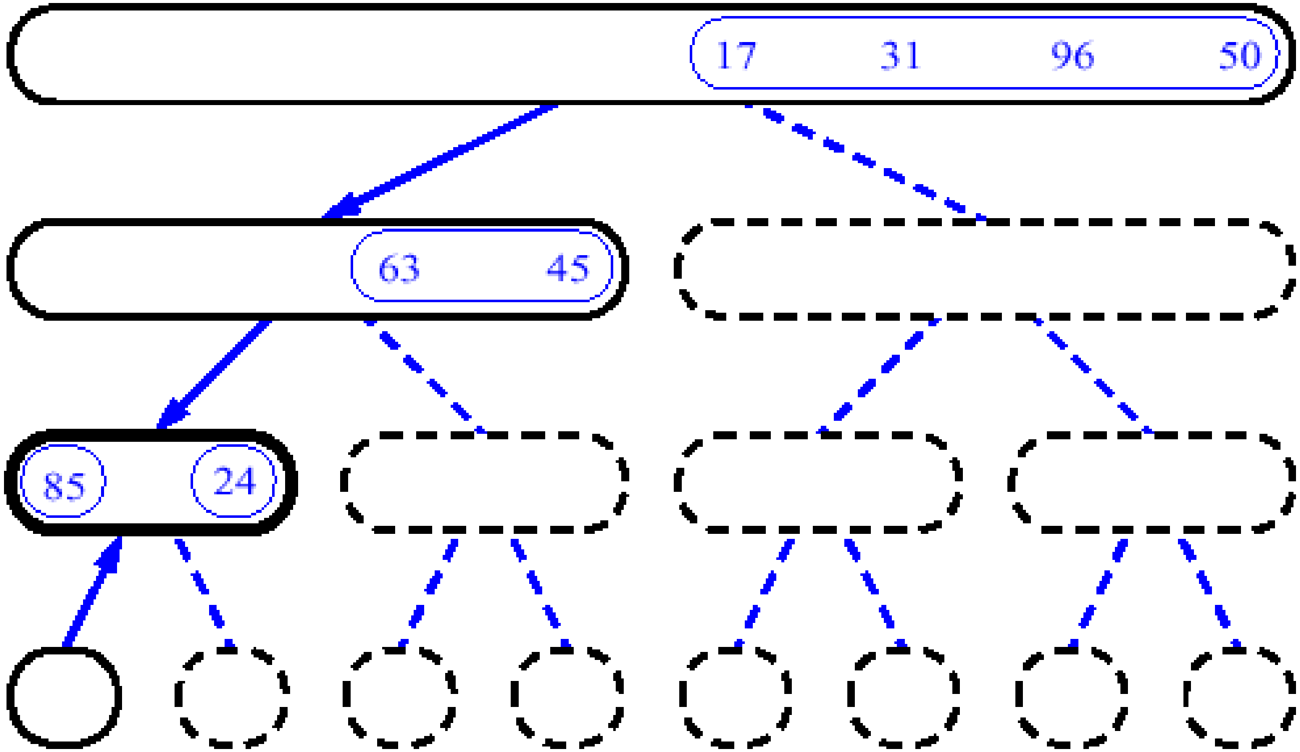
Merge Sort: example



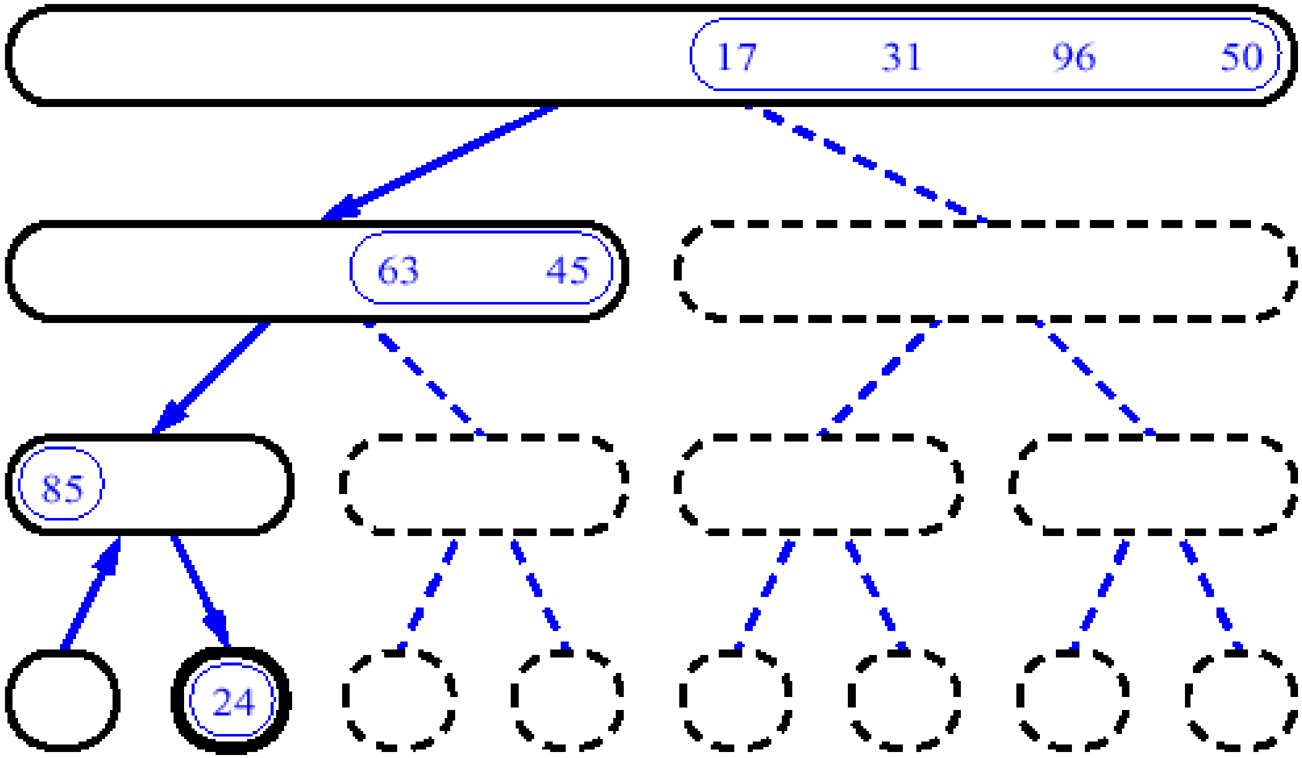
Merge Sort: example



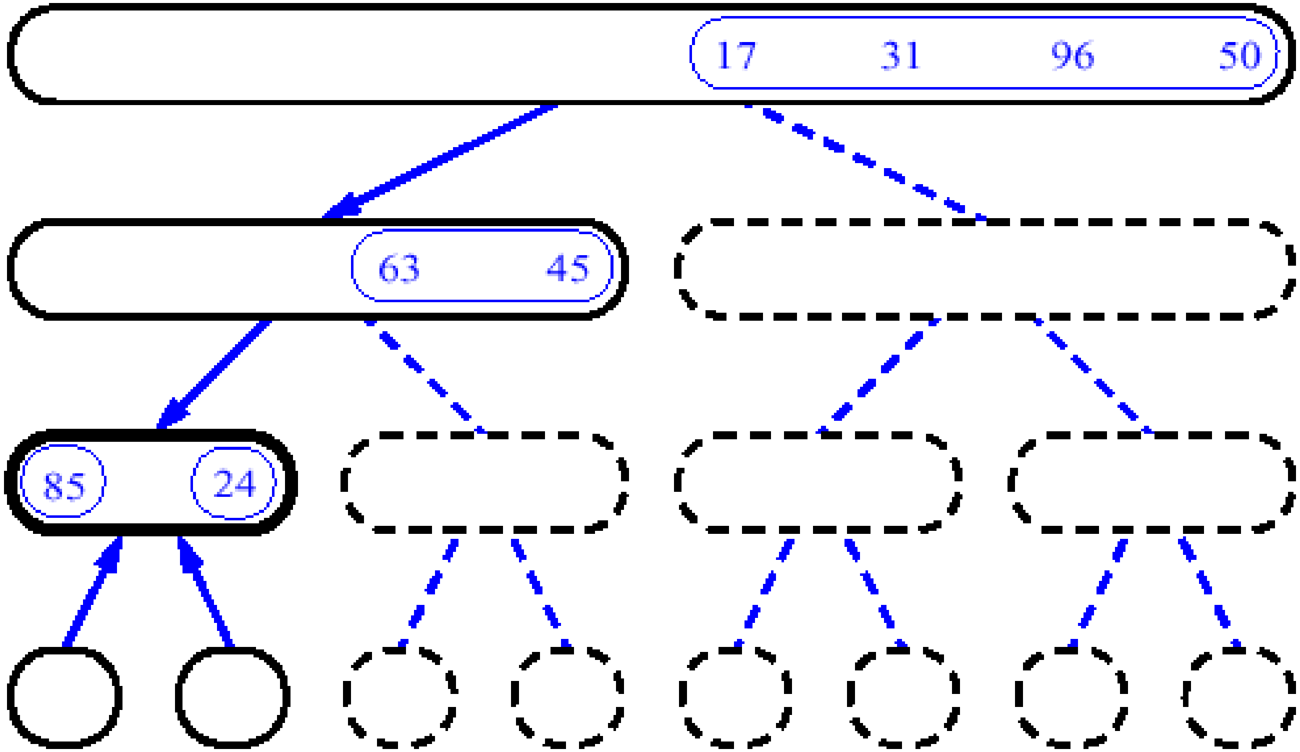
Merge Sort: example



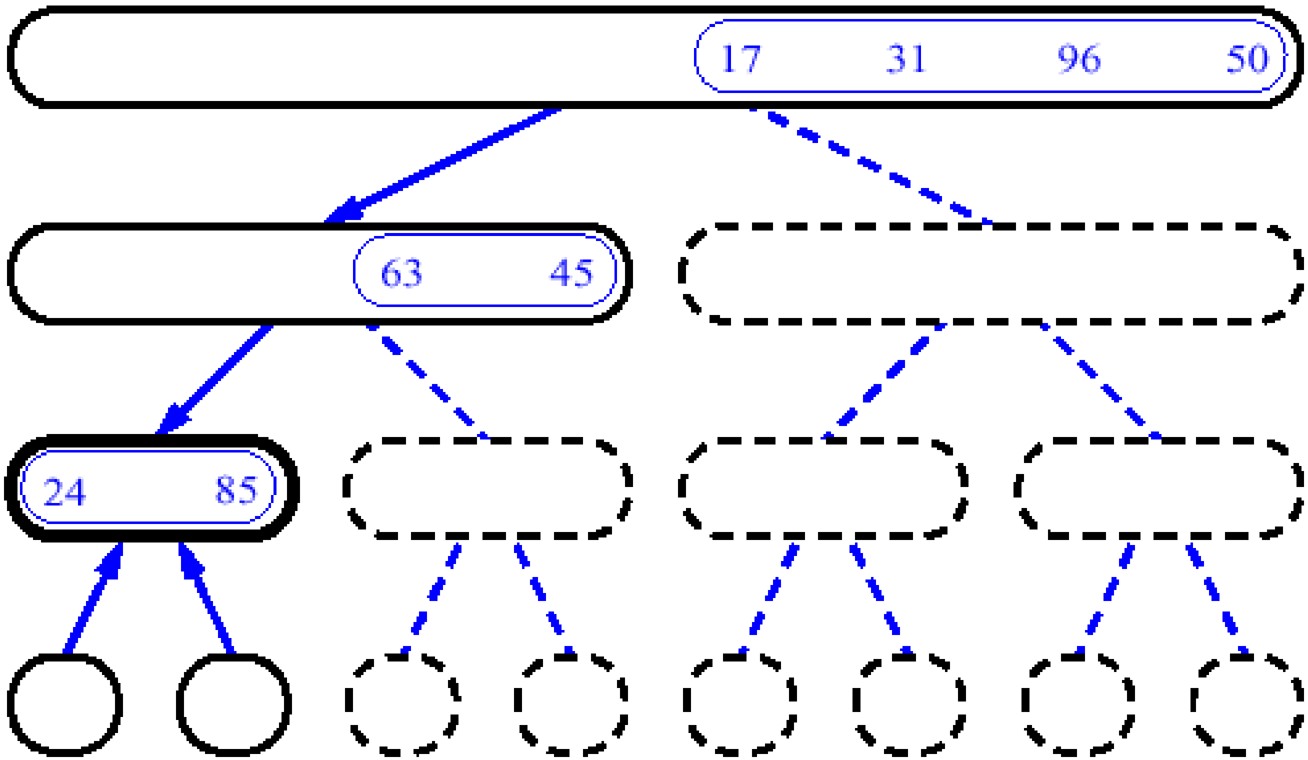
Merge Sort: example



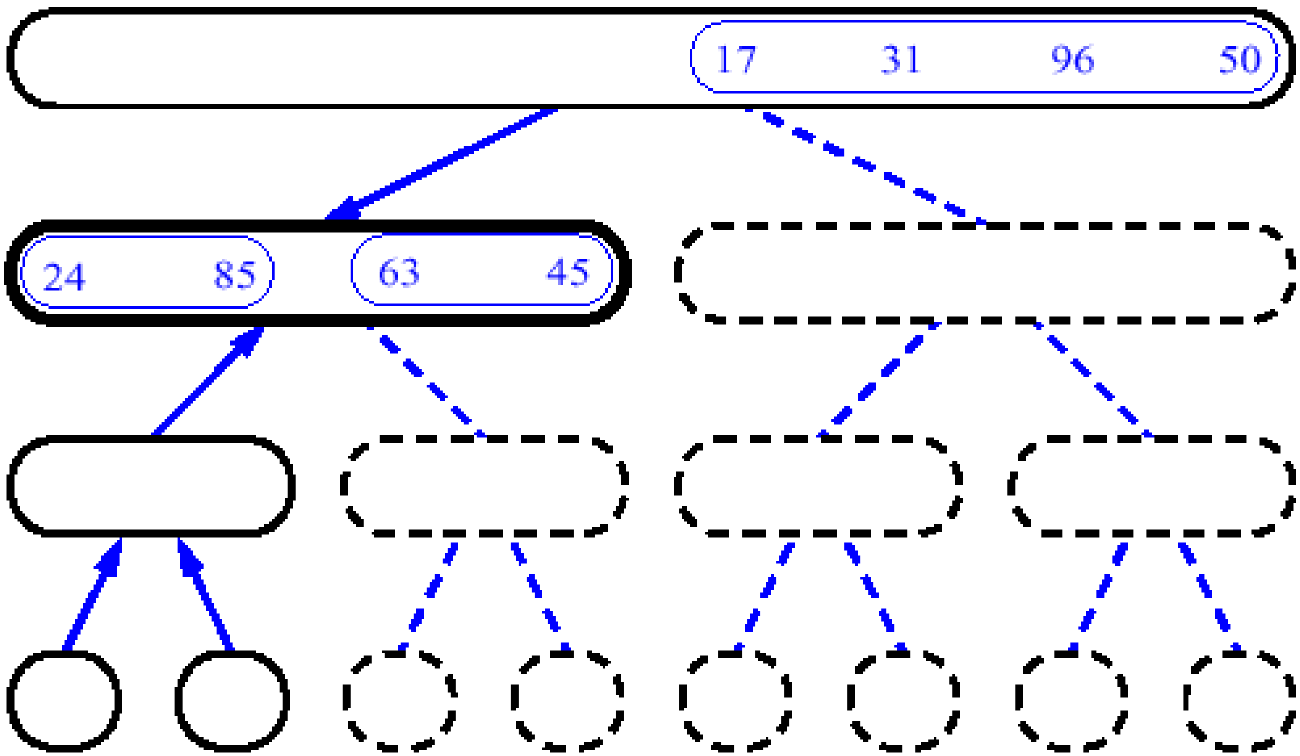
Merge Sort: example



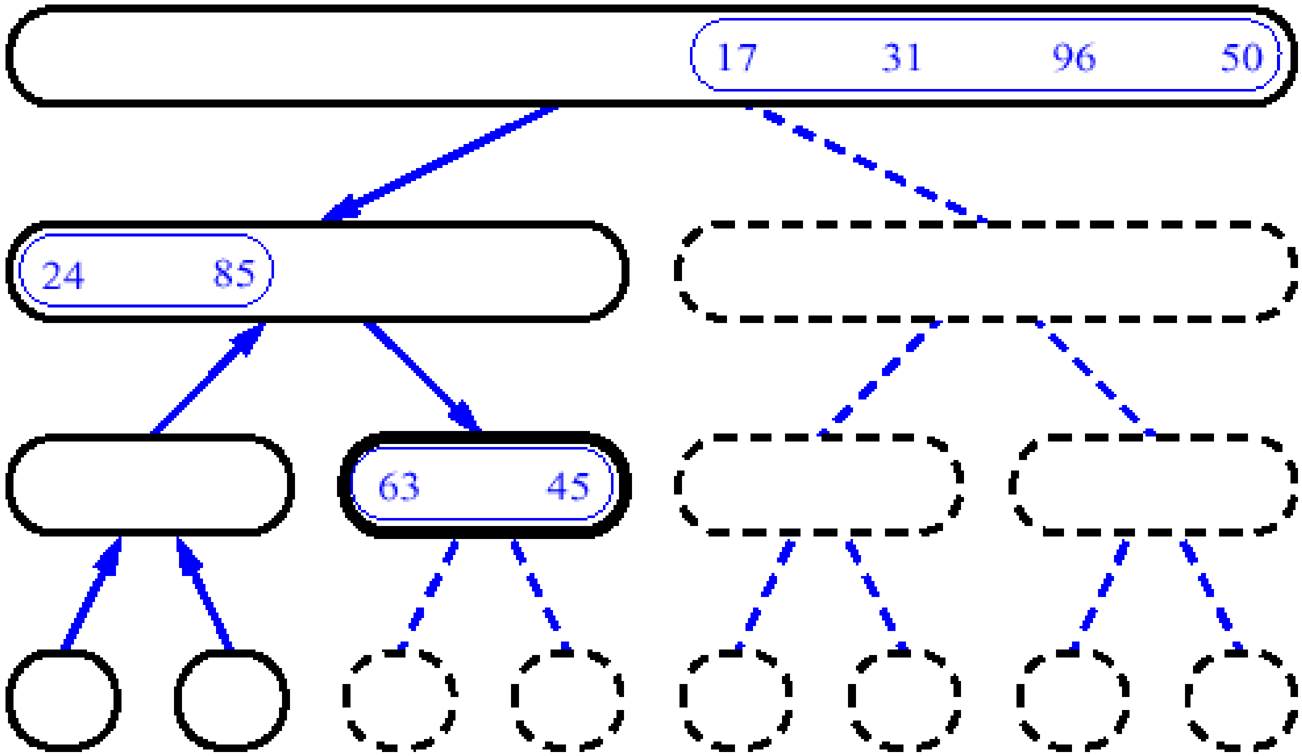
Merge Sort: example



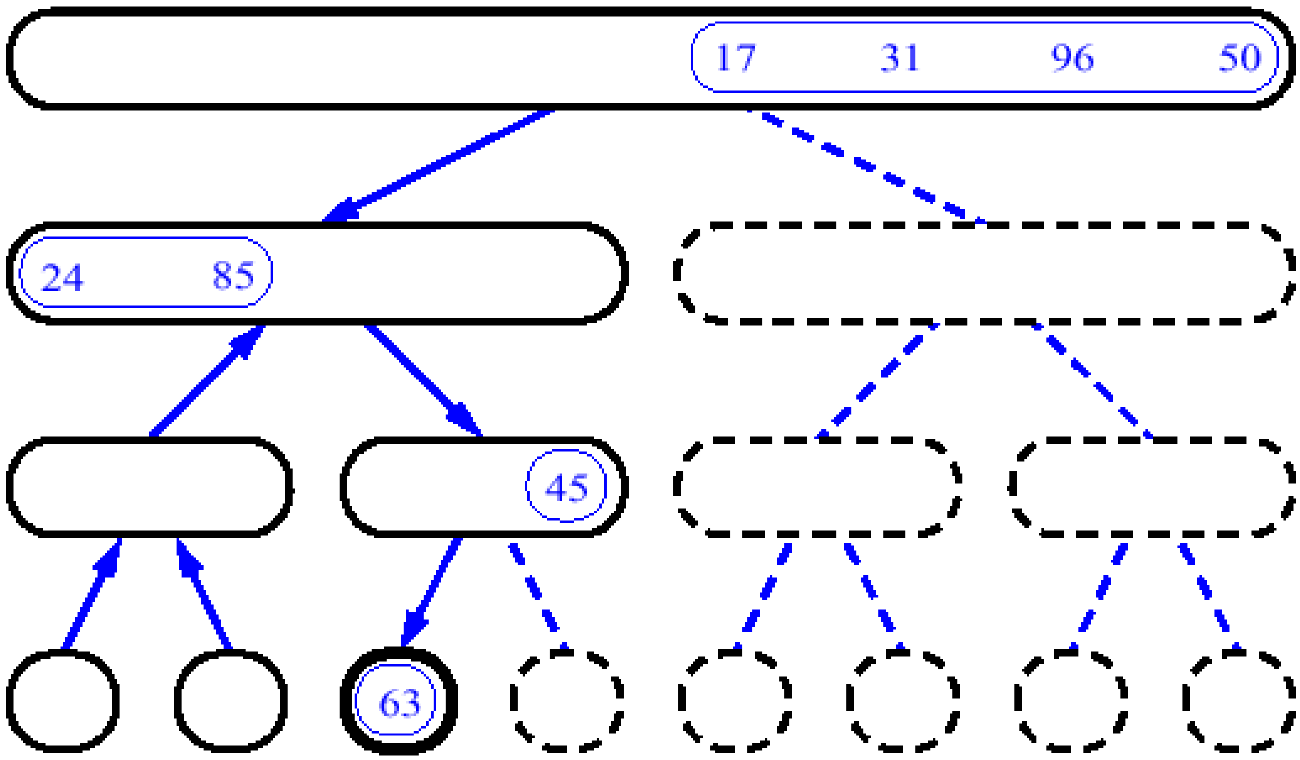
Merge Sort: example



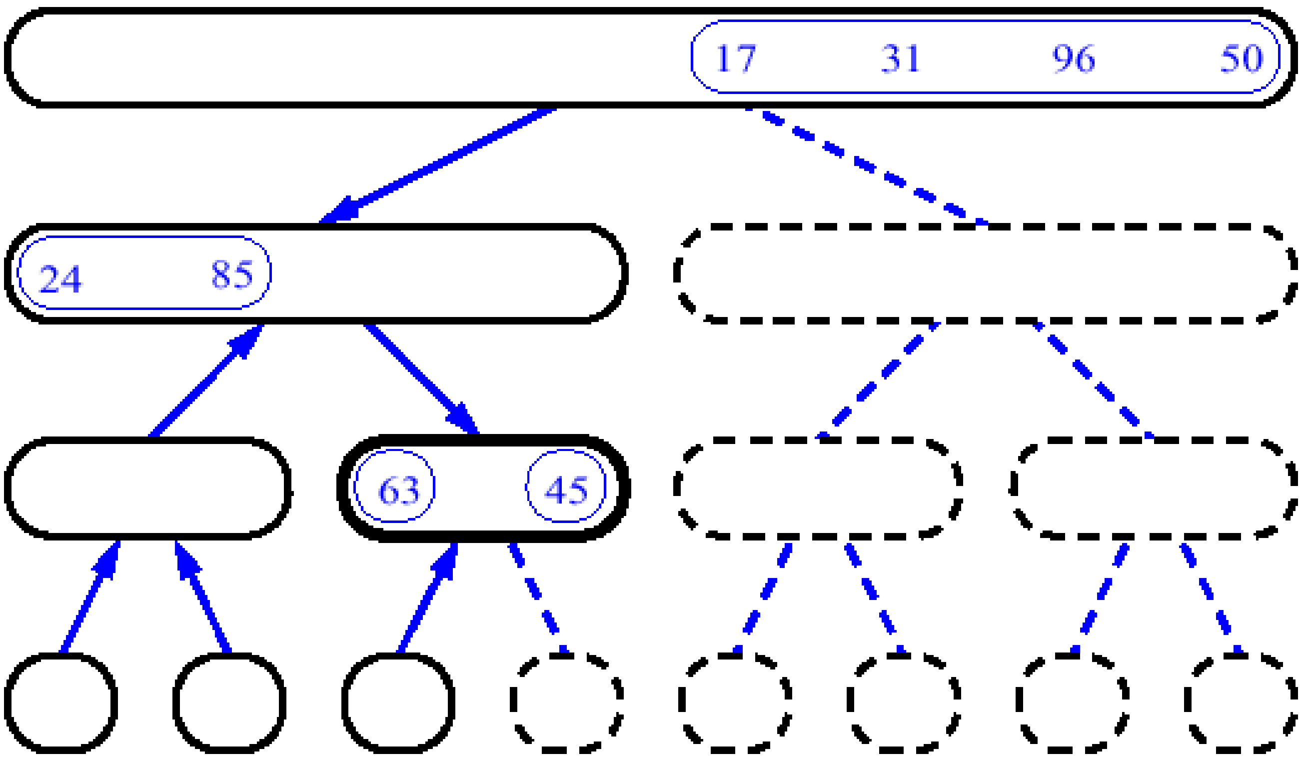
Merge Sort: example



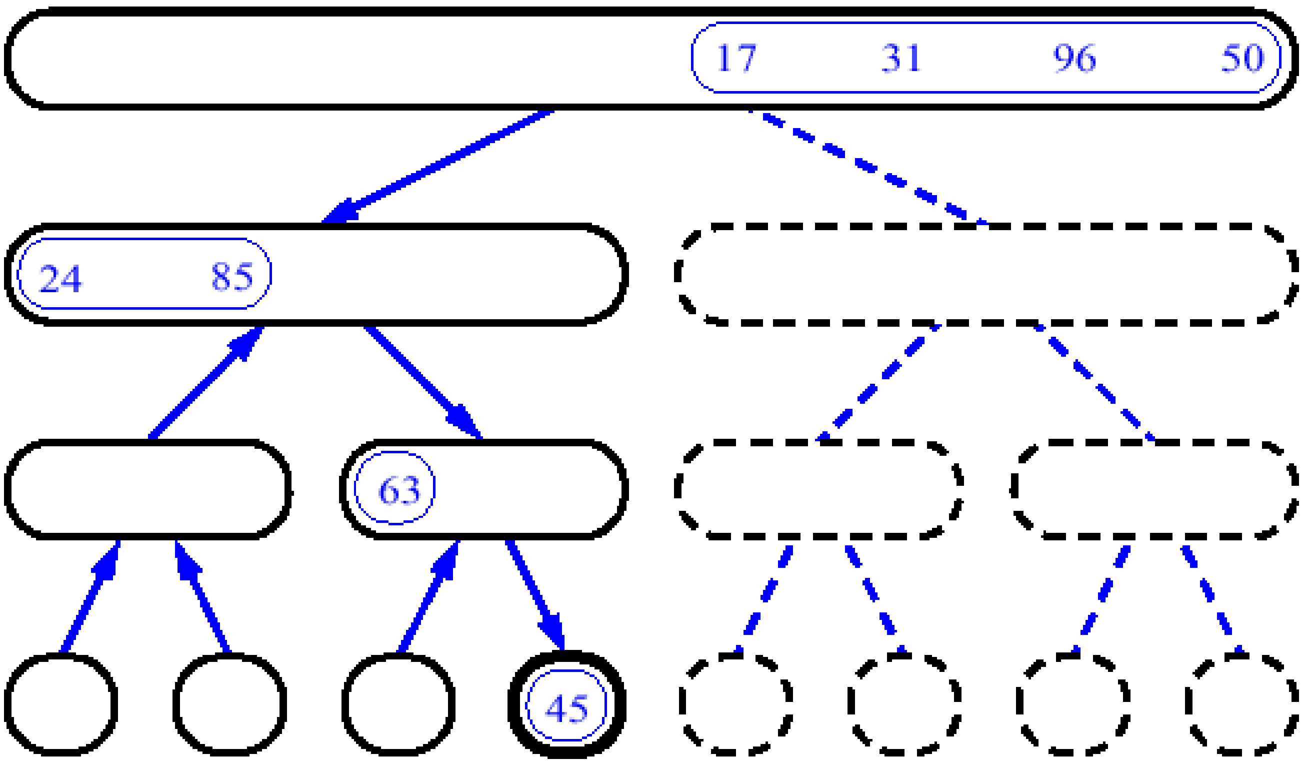
Merge Sort: example



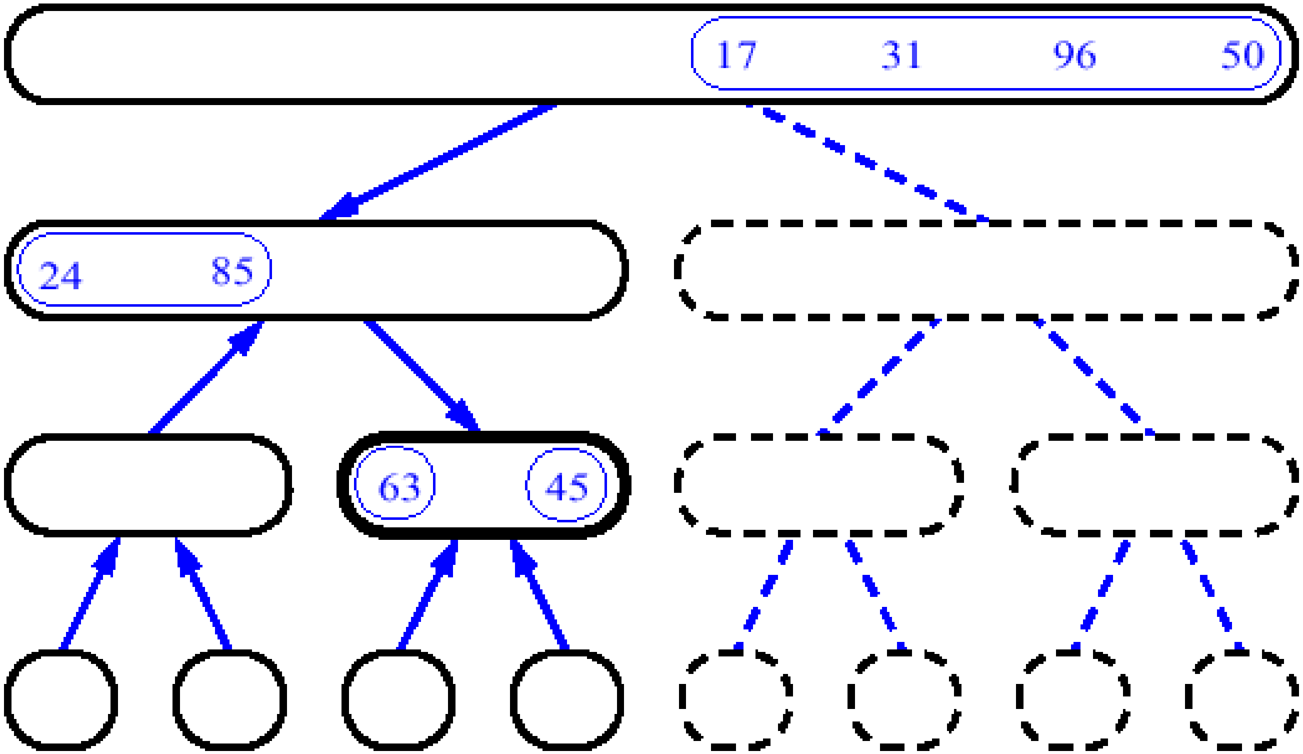
Merge Sort: example



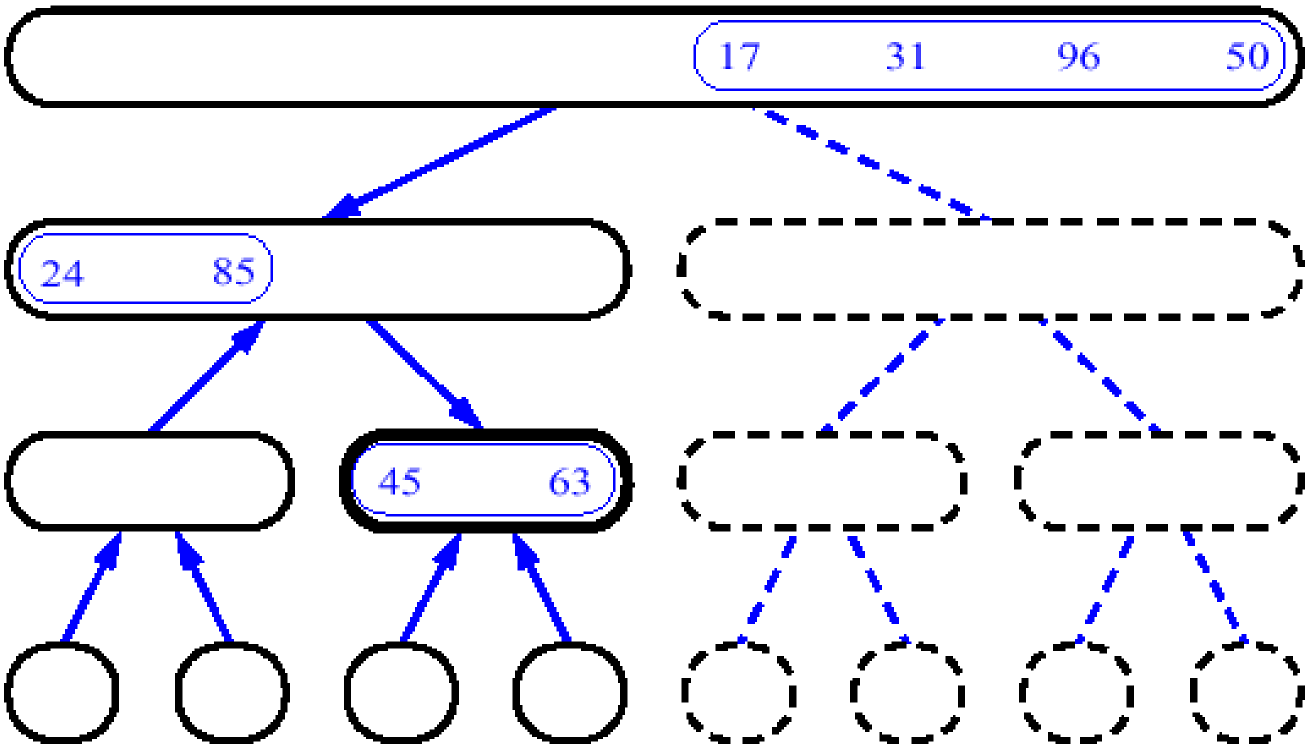
Merge Sort: example



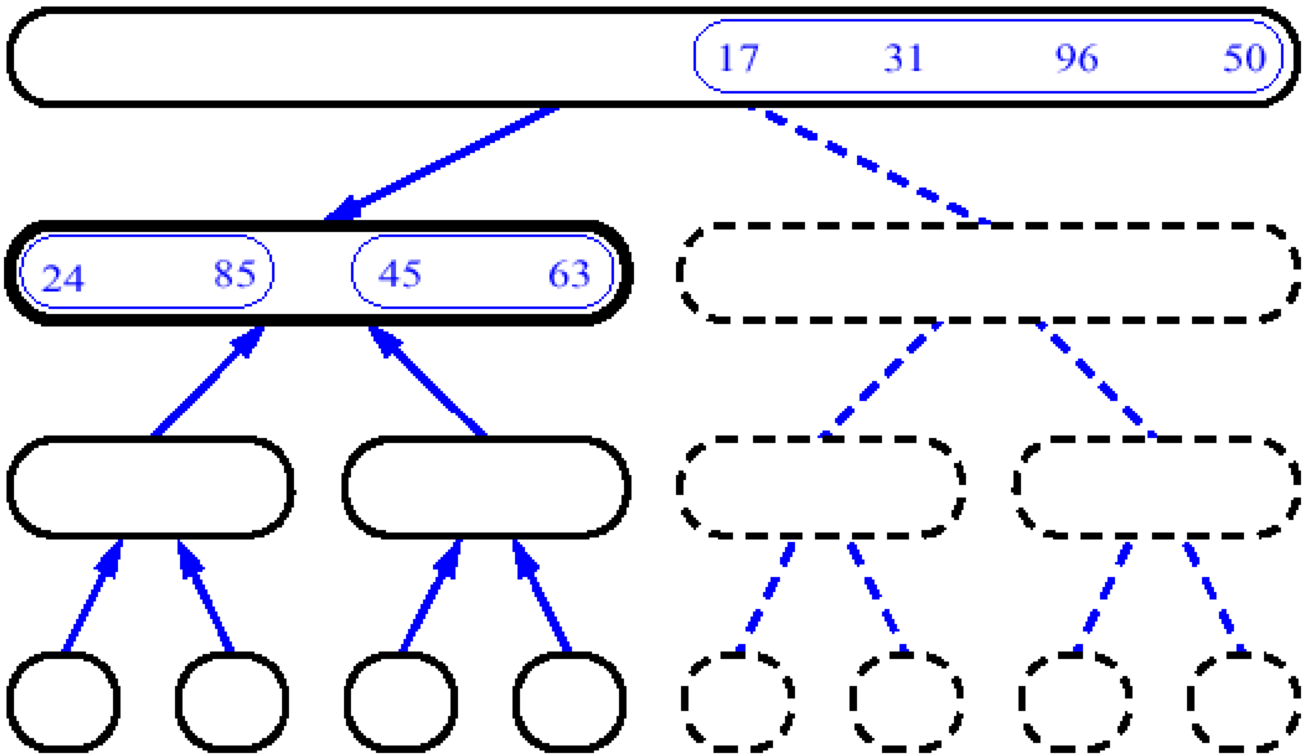
Merge Sort: example



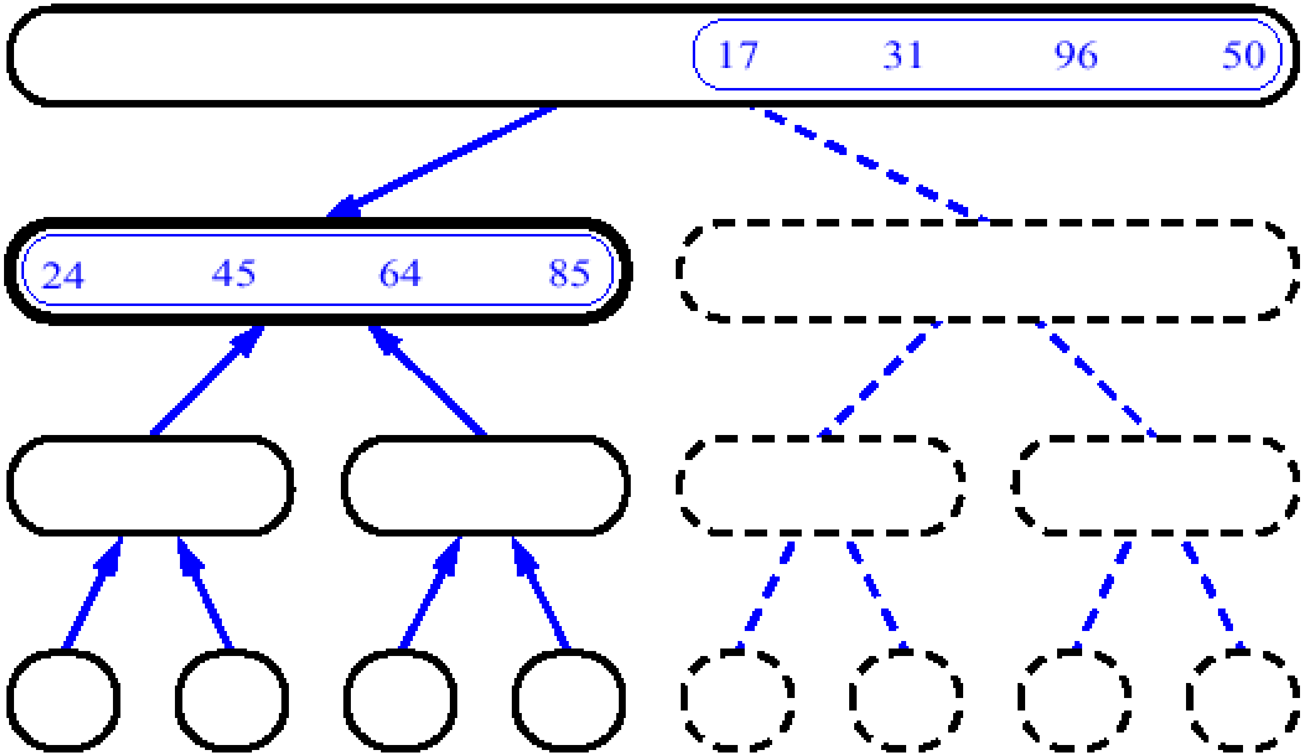
Merge Sort: example



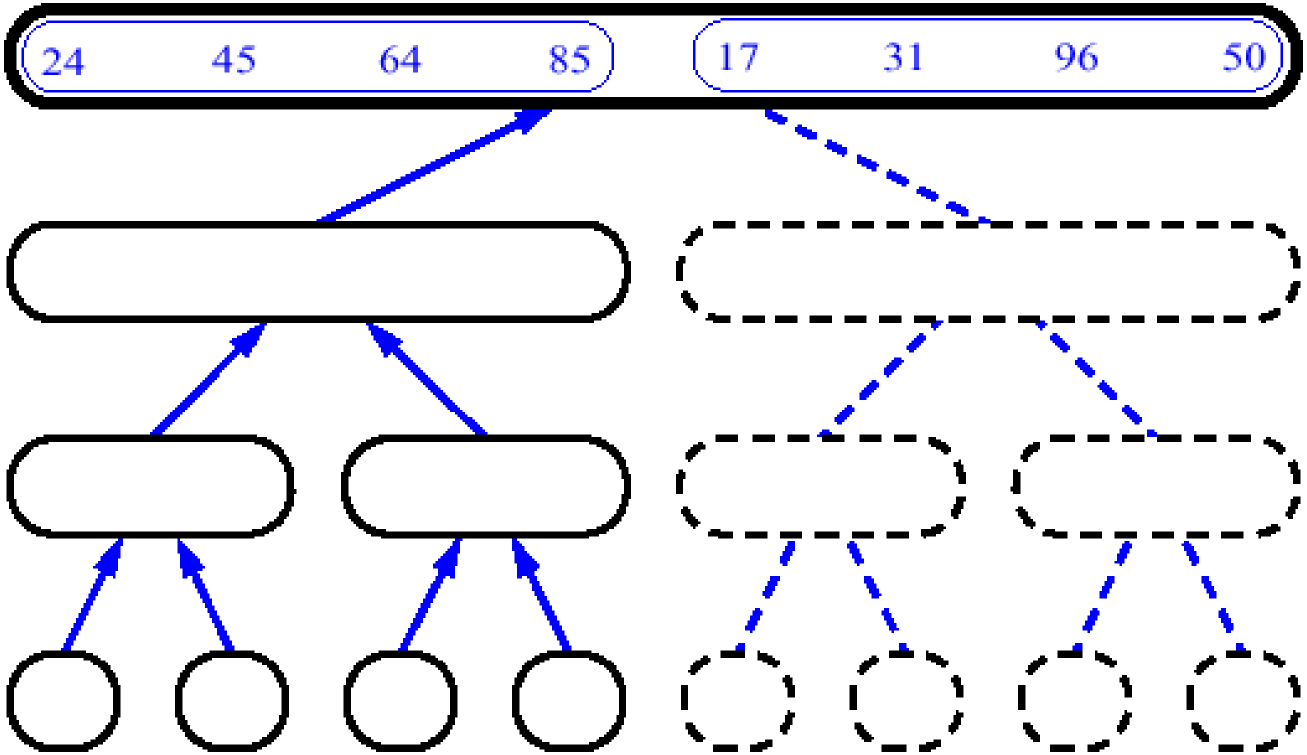
Merge Sort: example



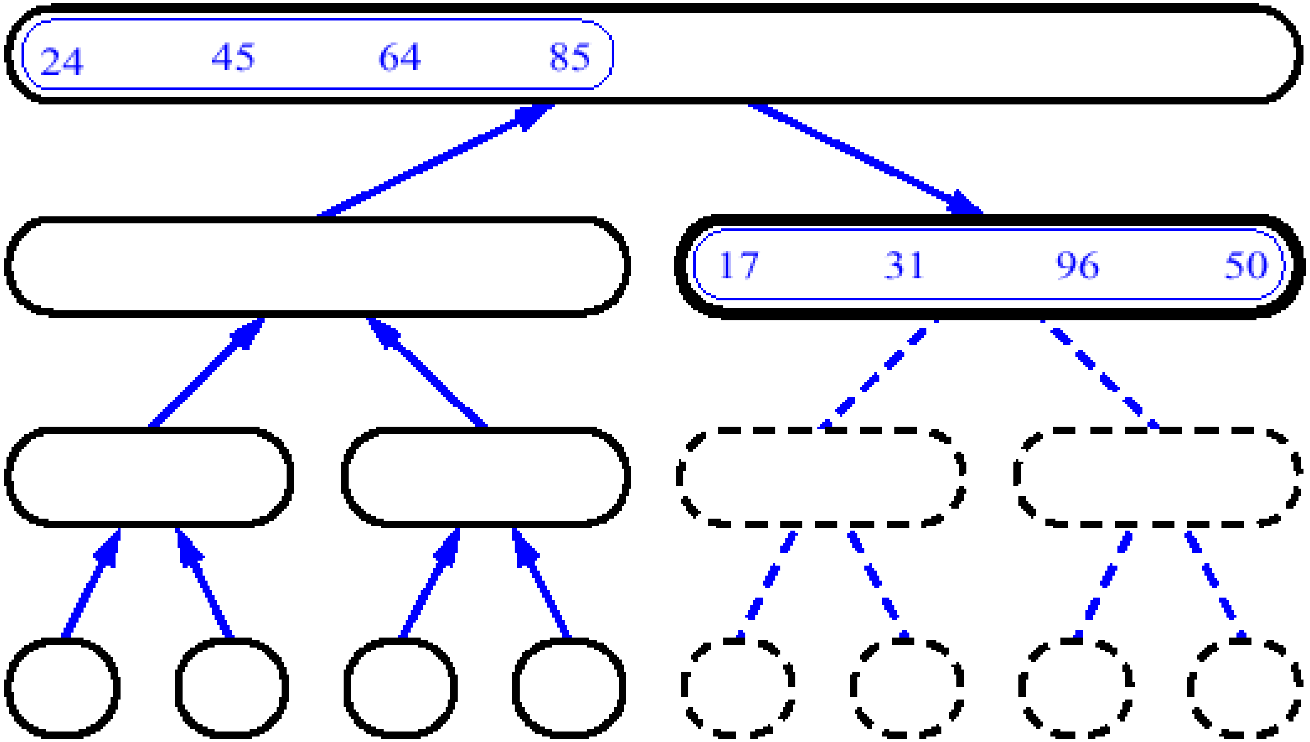
Merge Sort: example



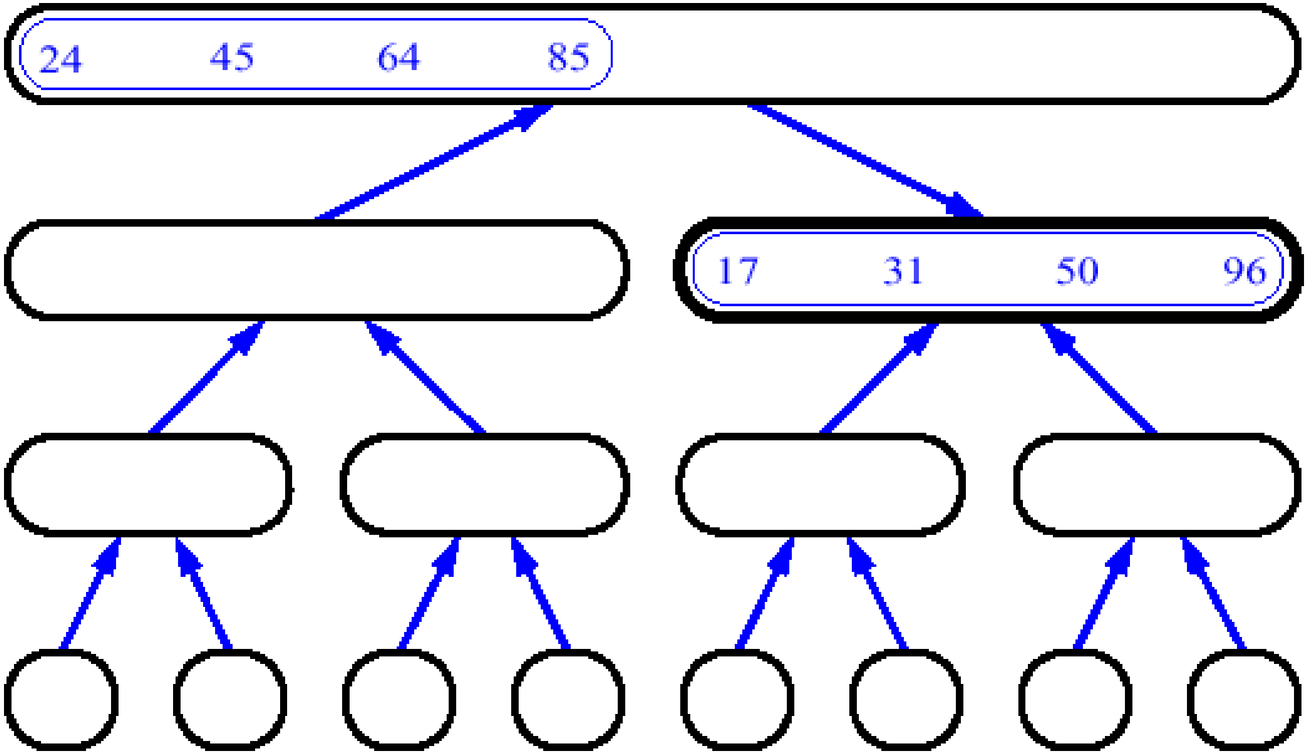
Merge Sort: example



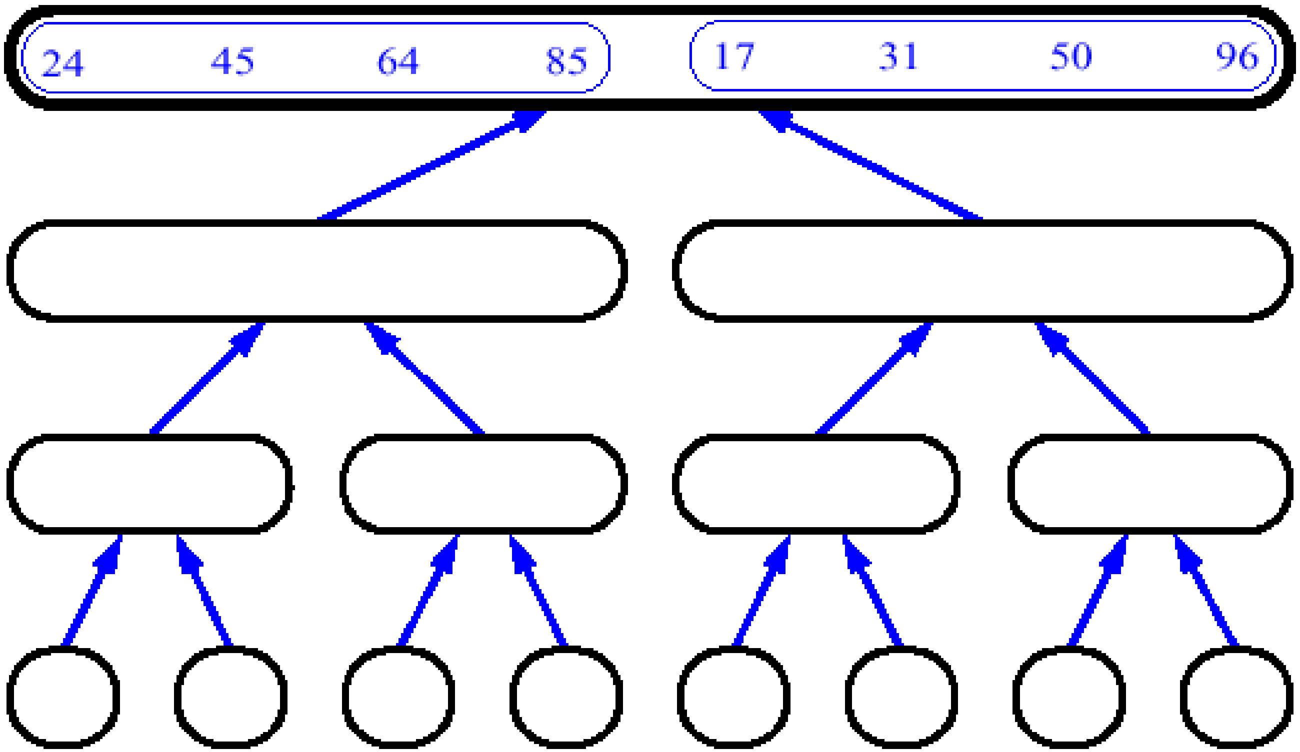
Merge Sort: example



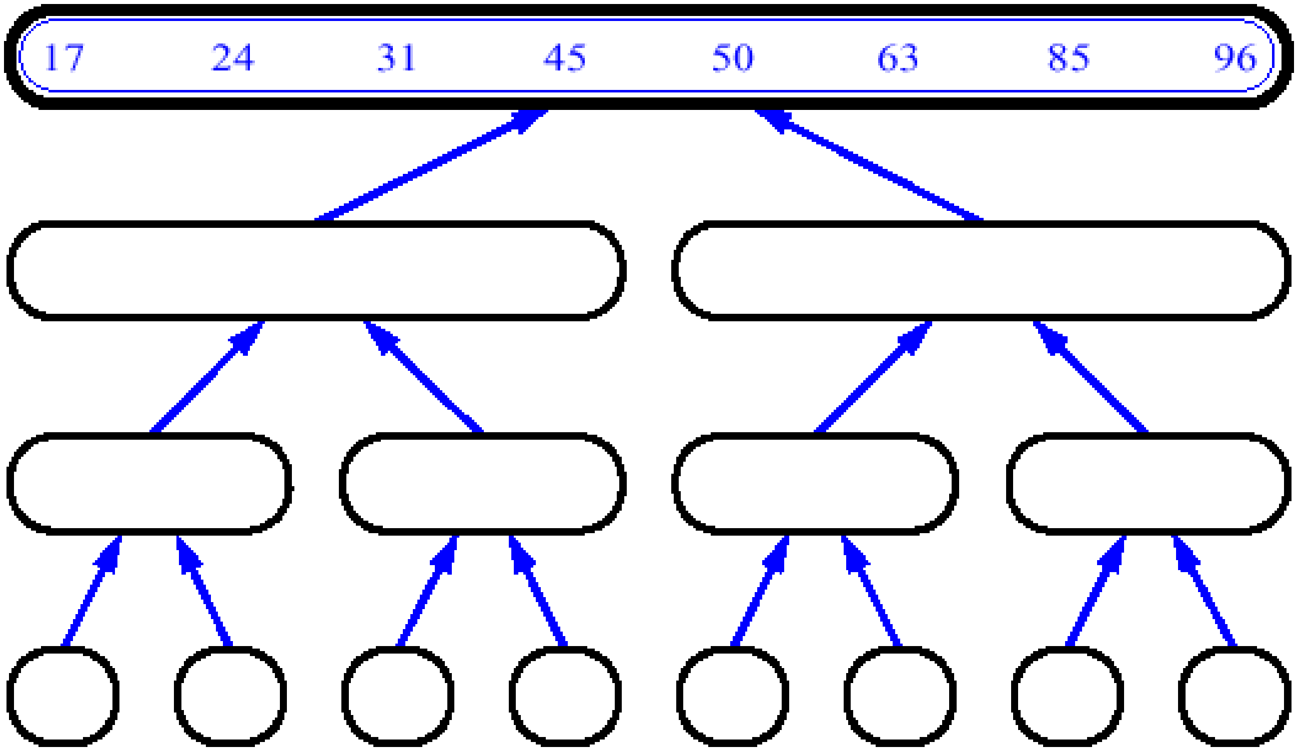
Merge Sort: example



Merge Sort: example

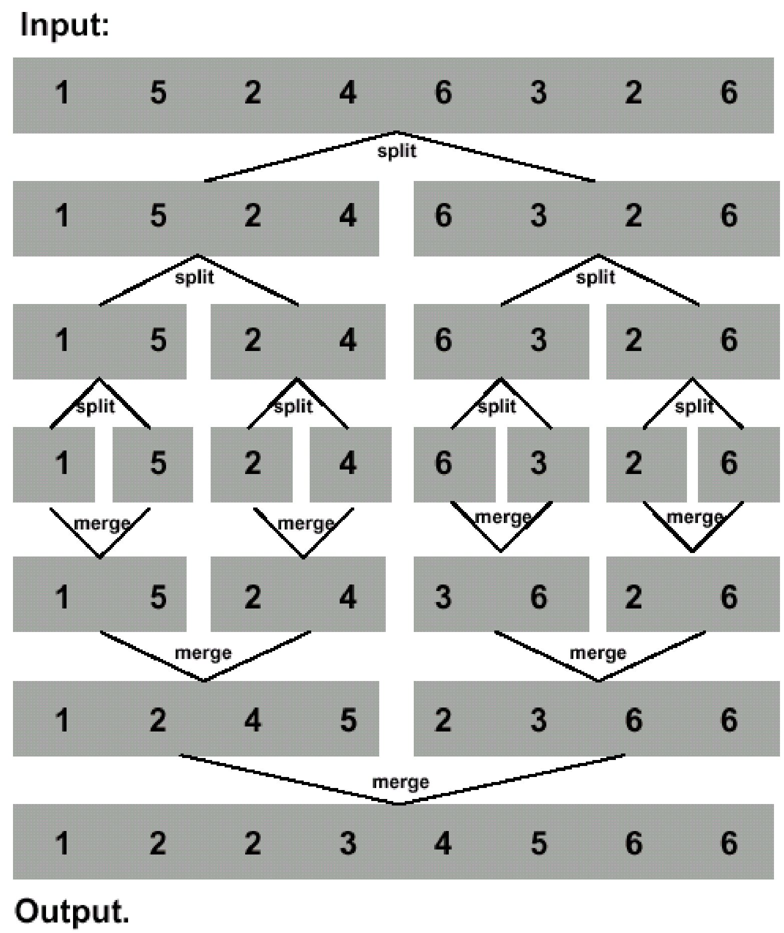


Merge Sort: example



Merge Sort: summary

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



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

Example: Merge Sort

$$T(n) = \begin{cases} \text{solving\_trivial\_problem} & \text{if } n = 1 \\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

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

Solving recurrences

- Repeated substitution method
  - Expanding the recurrence by substitution and noticing patterns
- Substitution method
  - guessing the solutions
  - verifying the solution by the mathematical induction
- Recursion-trees
- Master method
  - templates for different classes of recurrences

Repeated Substitution Method

- Let's find the running time of merge sort (let's assume that $n=2^b$, for some b).

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

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

Repeated Substitution Method

- The procedure is straightforward:
  - Substitute
  - Expand
  - Substitute
  - Expand
  - …
  - Observe a pattern and write how your expression looks after the *i*-th substitution
  - Find out what the value of *i* (e.g., lg *n*) should be to get the base case of the recurrence (say $T(1)$)
  - Insert the value of $T(1)$ and the expression of *i* into your expression

Substitution method

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

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

2) Assume $T(k) \le ck^3$ for $k \le n/2$ and

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

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \text{ (recurrence)} \\
&\le 4\mathrm{c}(\mathrm{n}/2)^3 + n \text{ (ind. hypoth.)} \\
&= \frac{c}{2}n^3 + n \text{ (simplify)} \\
&= cn^3 - \left(\frac{c}{2}n^3 - n\right) \text{ (rearrange)} \\
&\le cn^3 \text{ if } c \ge 2 \text{ and } n \ge 1 \text{ (satisfy)}
\end{aligned}
$$

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

Subtlety: Must choose c big enough to handle

$T(n) = \Theta(1)$ for $n < n_0$ for some n_0

Substitution method

- Achieving tighter bounds

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

Assume $T(k) \leq ck^2$

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &\leq cn^2 \text{ for no choice of } c > 0. \end{aligned}$$

Substitution method

The problem: We could not rewrite the equality

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

as:

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

in order to show the inequality we wanted

- Sometimes to prove inductive step, try to strengthen your hypothesis
  - $T(n) \leq$ (answer you want) - (something > 0)

Substitution method

- Corrected proof: the idea is to strengthen the inductive hypothesis by subtracting lower-order terms!

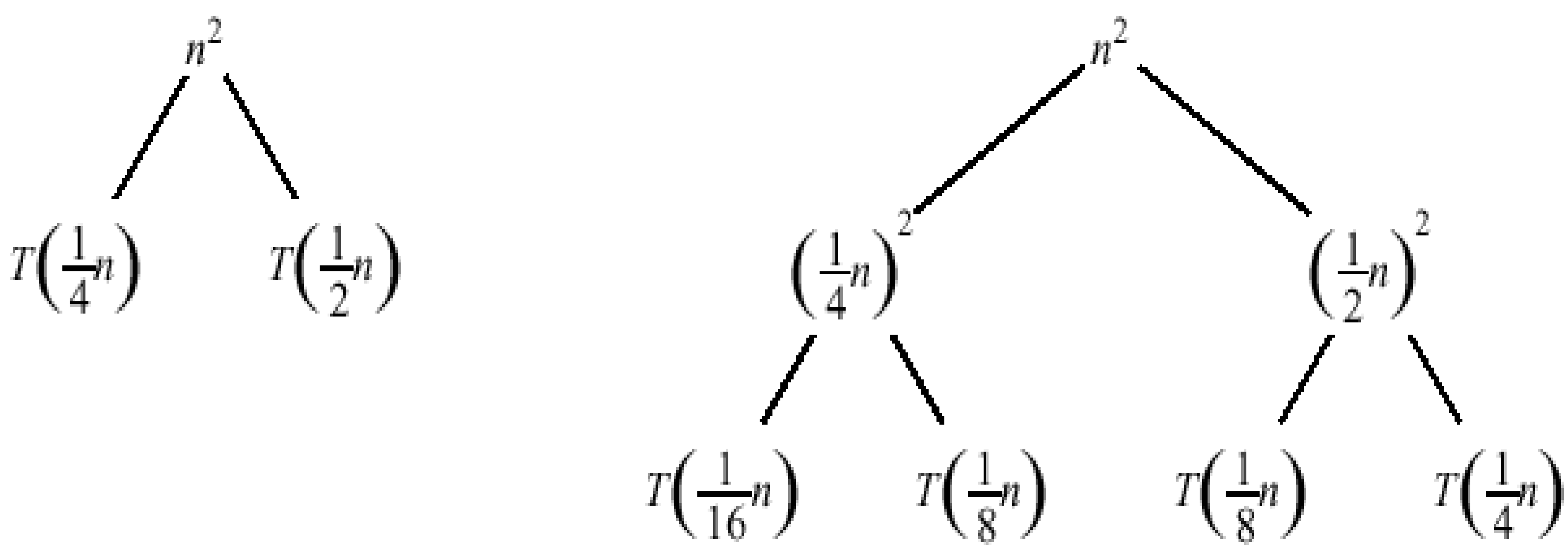
$$\text{Assume } T(k) \le c_1 k^2 - c_2 k \text{ for } k < n$$

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\le c_1 n^2 - c_2 n \text{ if } c_2 \ge 1 \end{aligned}$$

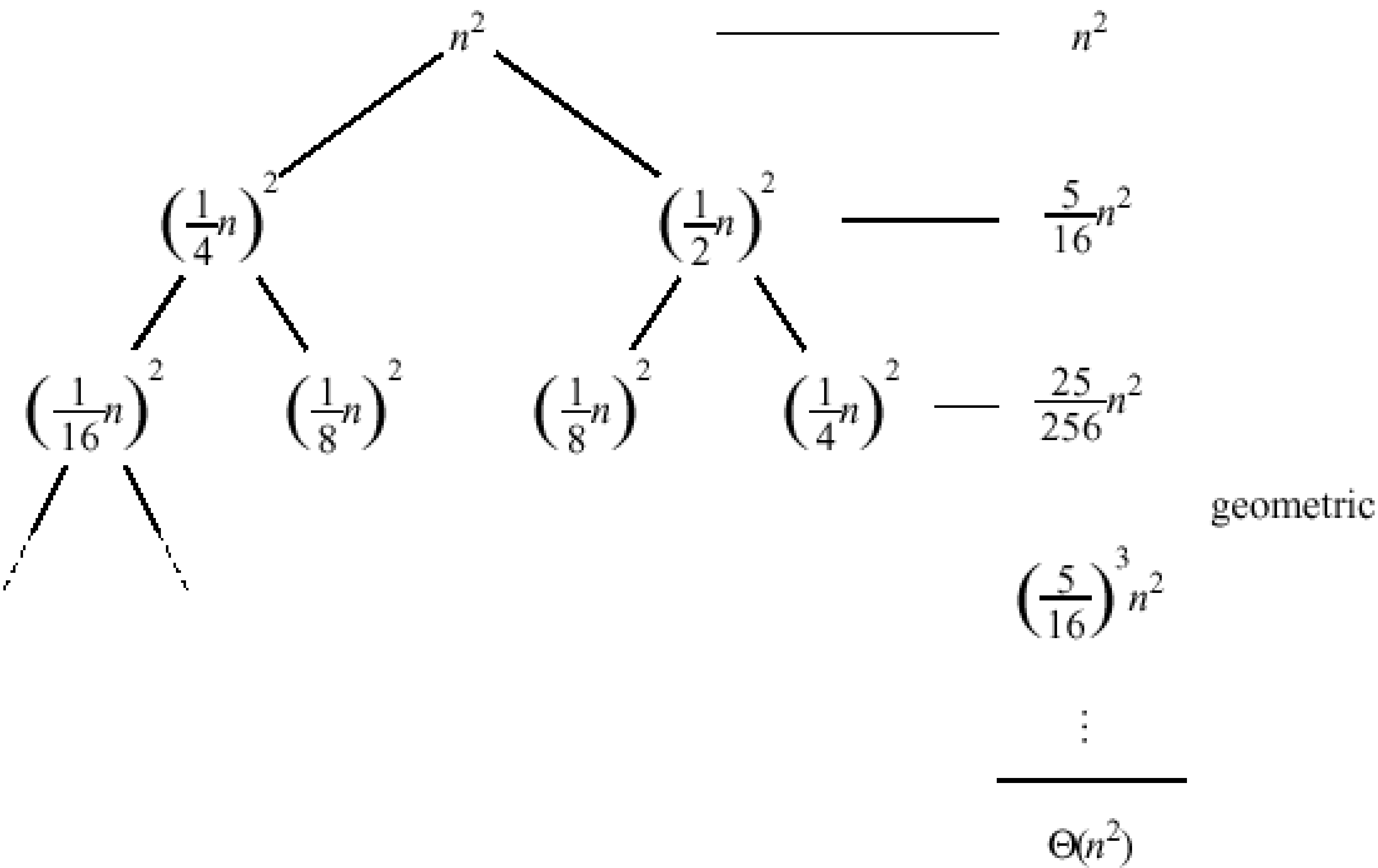
Recursion Tree

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Construction of a recursion tree

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

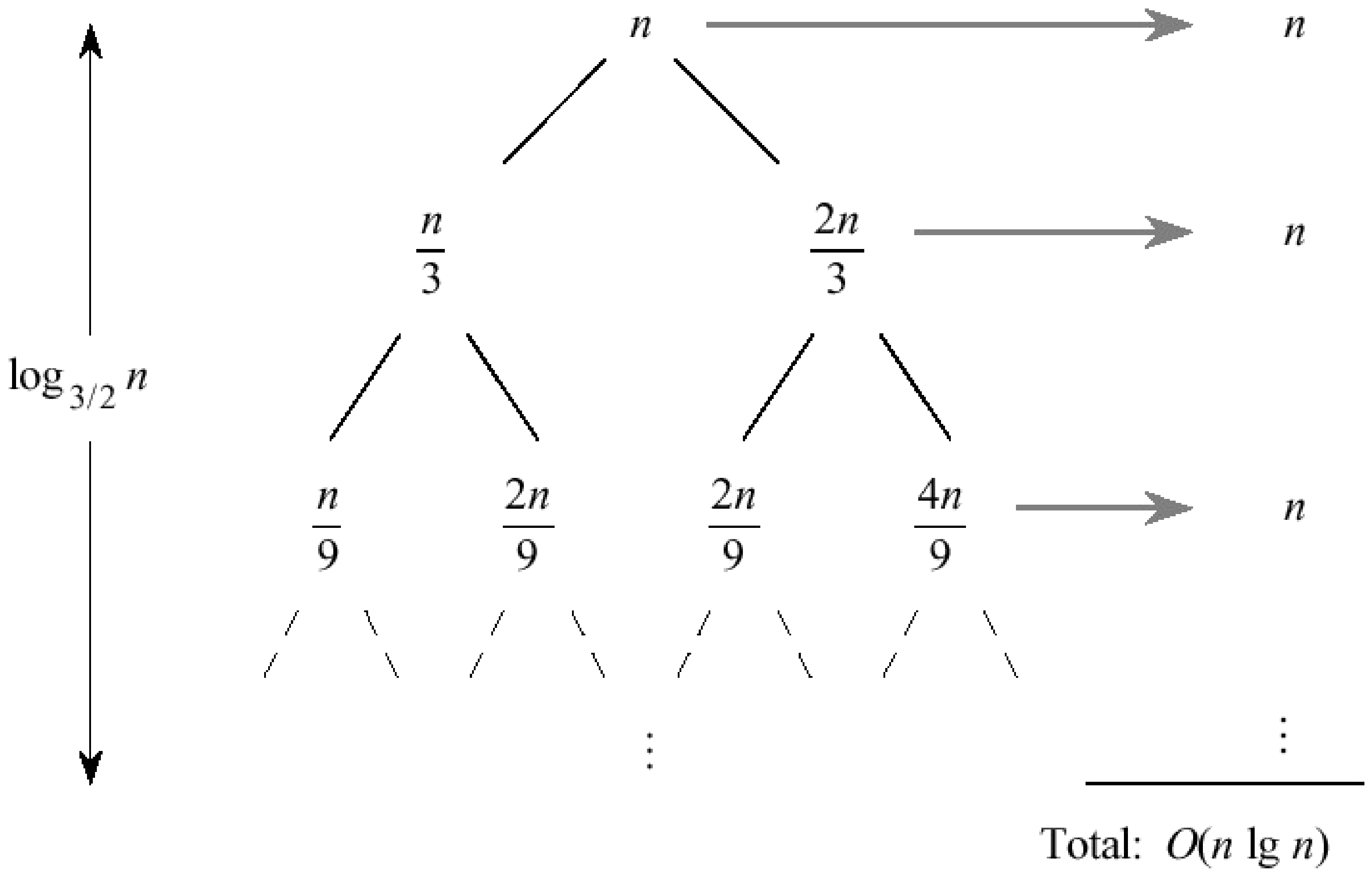


Recursion Tree



Recursion Tree

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



EECS 3101

Master Method

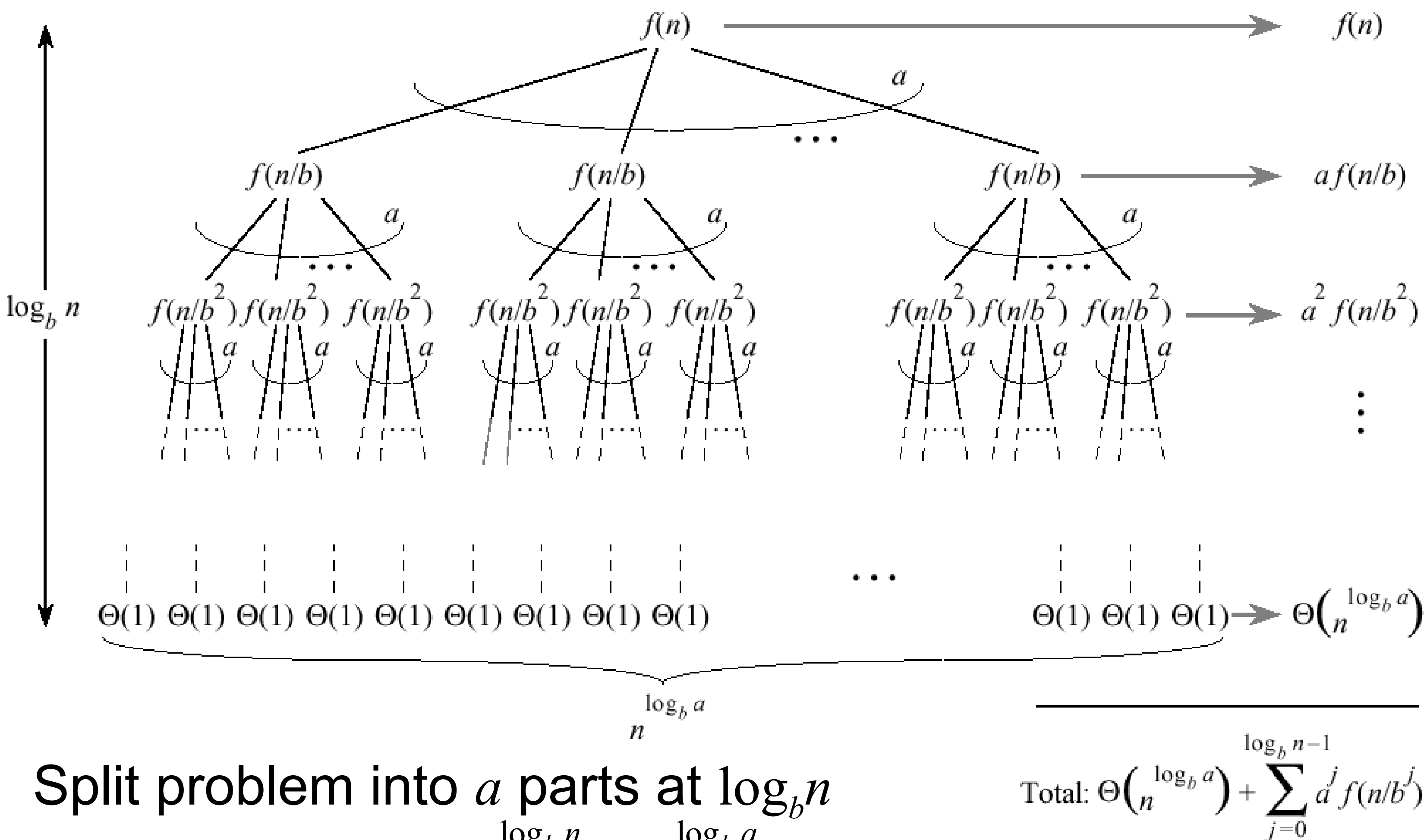
- The idea is to solve a class of recurrences that have the form

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

- a $\geq$ 1 and b > 1, and f 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



Split problem into a parts at $\log_b n$ levels. There are $a^{\log_b n} = n^{\log_b a}$ leaves

Master method

- Number of leaves:
- Iterating the recurrence, expanding the tree yields

$$a^{\log_b n} = n^{\log_b a}$$

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

Thus,

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

  – The first term is a division/recombination cost (totaled across all levels of the tree)
  – The second term is the cost of doing all $n^{\log_b a}$ subproblems of size 1 (total of all work pushed to 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 root
- Consequently, to solve the recurrence, we need only to characterize 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 (by factor n^{ε}) slower than $n^{\log_b a}$
- **The work at the leaf level dominates**
  - Summation of recursion-tree levels $O(n^{\log_b a})$
  - Cost of all the leaves $\Theta(n^{\log_b a})$
  - Thus, the overall cost $\Theta(n^{\log_b a})$

Master method Case 2

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

- **The work is distributed equally throughout the tree** $T(n) = \Theta(n^{\log_b a} \lg n)$
  - (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

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

- **The work at the root dominates**

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

Master Theorem Summarized

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$

1. $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$

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

2. $f(n) = \Theta\left(n^{\log_b a}\right)$

   $\Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3. $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ and $af(n/b) \le cf(n)$, for some $c < 1, n > n_0$

   $\Rightarrow T(n) = \Theta\left(f(n)\right)$

- The master method cannot solve every recurrence of this form; there is a gap between cases 1 and 2, as well as cases 2 and 3

Using the Master Theorem

- Extract a, b, and f(n) from a given recurrence
- Determine $n^{\log_b a}$
- Compare f(n) and $n^{\log_b a}$ asymptotically
- Determine appropriate MT case, and apply
- Example merge sort

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

$$a = 2,\ b = 2;\ n^{\log_b a} = n^{\log_2 2} = n = \Theta(n)$$

$$\text{Also } f(n) = \Theta(n)$$

$$\Rightarrow \text{Case 2:}\ \ T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta\left(n \lg n\right)$$

Examples

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

$$a = 1, b = 2;\ n^{\log_2 1} = 1$$

$$\text{also } f(n) = 1, f(n) = \Theta(1)$$

$$\Rightarrow \text{Case 2: } T(n) = \Theta(\lg n)$$

```
Binary-search(A, p, r, s):
    q←(p+r)/2
    if A[q]=s then return q
    else if A[q]>s then
          Binary-search(A, p, q-1, s)
    else Binary-search(A, q+1, r, s)
```

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

$$a = 9, b = 3;$$

$$f(n) = n, f(n) = O(n^{\log_3 9 - \varepsilon}) \text{ with } \varepsilon = 1$$

$$\Rightarrow \text{Case 1: } T(n) = \Theta\left(n^2\right)$$

Examples

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

$$a = 3, b = 4;\ n^{\log_4 3} = n^{0.793}$$

$$f(n) = n\lg n,\ f(n) = \Omega(n^{\log_4 3+\varepsilon}) \text{ with } \varepsilon \approx 0.2$$

$\Rightarrow$ Case 3:

Regularity condition

$$af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n\lg n = cf(n) \text{ for } c = 3/4$$

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

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

$$a = 2, b = 2;\ n^{\log_2 2} = n^1$$

$$f(n) = n\lg n,\ f(n) = \Omega(n^{1+\varepsilon}) \text{ with } \varepsilon\text{?}$$

also $n\lg n / n^1 = \lg n$

$\Rightarrow$ neither Case 3 nor Case 2!

Examples

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

$$a = 4, b = 2;\ n^{\log_2 4} = n^2$$

$$f(n) = n^3;\ f(n) = \Omega(n^2)$$

$\Rightarrow$ Case 3: $T(n) = \Theta\left(n^3\right)$

Checking the regularity condition

$$4f(n/2) \le cf(n)$$

$$4n^3/8 \le cn^3$$

$$n^3/2 \le cn^3$$

$$c = 3/4 < 1$$

Next...

1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 4 of the text.
2. Next, more sorting algorithms.