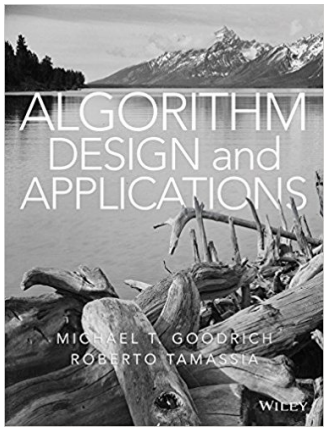
MANCHESTER 1824
The University of Manchester

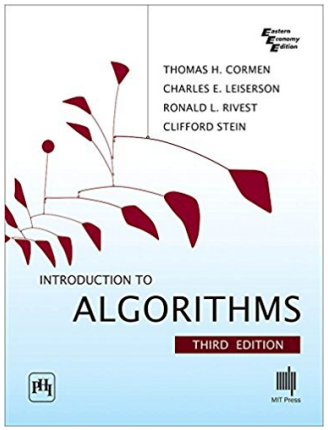
COMP26120: Divide and Conquer (2019/20)

Lucas Cordeiro
lucas.cordeiro@manchester.ac.uk

Divide-and-Conquer (Recurrence)

- References:
  - *Algorithm Design and Applications,* Goodrich, Michael T. and Roberto Tamassia (Chapter 8)
  - *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein (Chapters 2 and 4)





Intended Learning Outcomes

- **Understand** the **divide-and-conquer** paradigm and how **recurrence** can be obtained
- Solve recurrences using **substitution** method
- Describe **various examples** to analyse divide-and-conquer algorithms and how to solve their recurrences

Divide-and-Conquer

- The **divide-and-conquer** paradigm involves three steps at each level of the **recursion**:
  - **Divide** the problem into some subproblems that are smaller instances of the same problem *(D(n)))*
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, straightforwardly solve the subproblems *(aT(n/b))*
  - **Combine** the solutions to the subproblems into the solution for the original problem *(C(n))*

$$T(n) = \Theta(1) \text{ if } n \leq c,$$
$$T(n) = D(n) + aT(n/b) + C(n) \text{ otherwise.}$$

Illustrative Example (Merge Sort)

- The merge sort algorithm closely follows the **divide-and-conquer** paradigm
  - **Divide:** Divide the *n*-element sequence to be sorted into two subsequences of *n=2* elements each ($\Theta(1)$)
  - **Conquer:** Sort the two subsequences recursively using merge sort ($2T(n/2)$)
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer ($\Theta(n)$)

$$T(n) = \Theta(1) \text{ if } n = 1,$$
$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) \text{ otherwise.}$$

How Does Merge Sort Work?

- **Idea:** suppose we have two piles of cards face up on a table; **each pile is sorted**, with the smallest cards on top
- We wish to merge the two piles into a single sorted output pile, which is to be face down on the table
  1. choose the smaller of the two cards on top of the face-up piles
  2. remove it from its pile (which exposes a new top card)
  3. place this card face down onto the output pile
  4. repeat this until one input pile is empty, at which time we take the remaining input pile and place it face down onto the output pile

Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
       mid = floor((left + right) / 2);
       MergeSort(A, left, mid);
       MergeSort(A, mid+1, right);
       Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

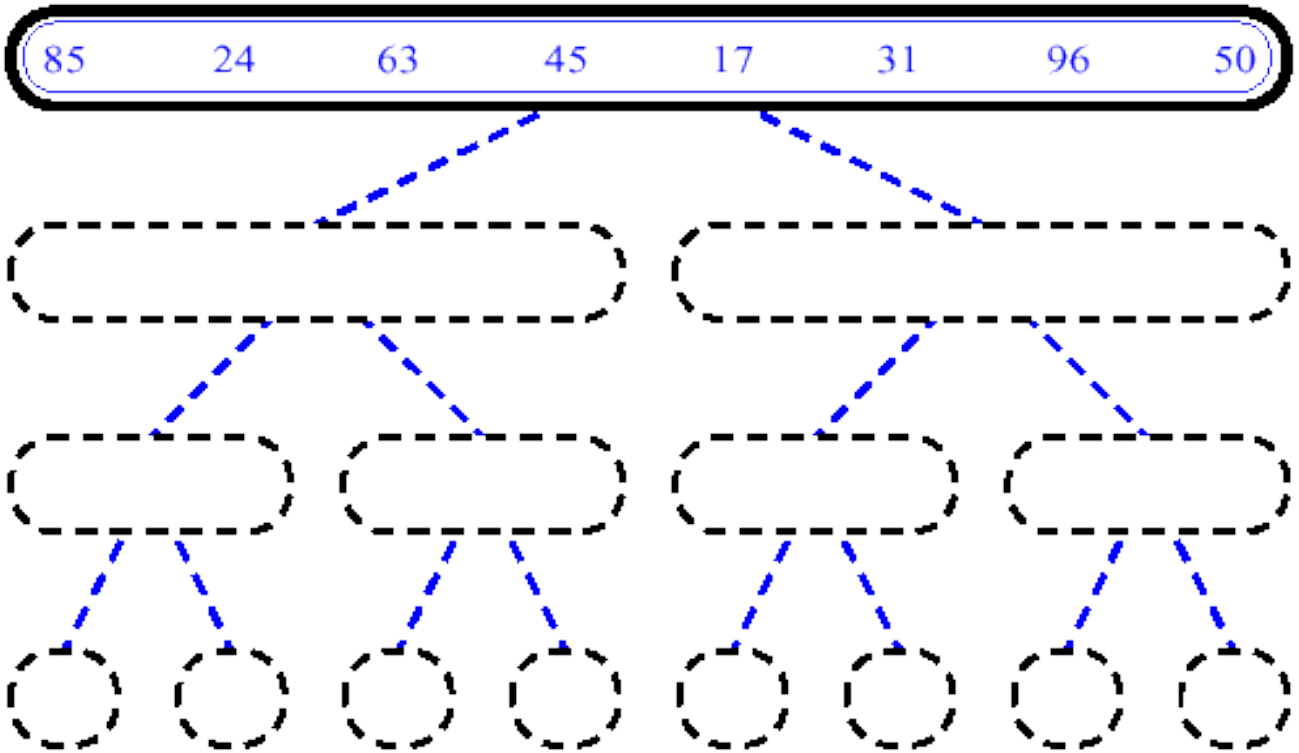
Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1
  n2 = right-mid
  create two sorted subarrays L[0..n1] and R[0..n2]
  for i=0 to n1-1
    do L[i]=A[left+i]
  for j=0 to n2-1
    do R[j]=A[mid+j+1]
  L[n1]=∞
  R[n2]=∞
  i=0
  j=0
  for k=left to right
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

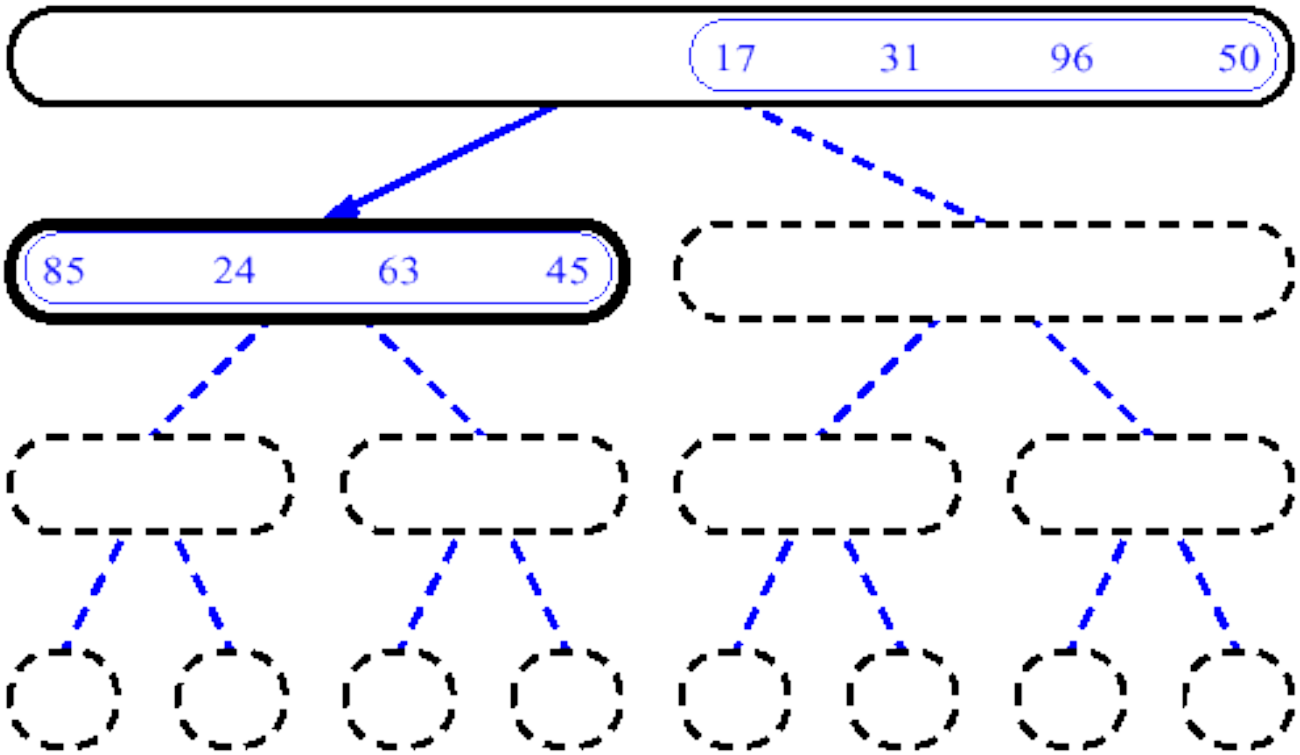
Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1                                          Θ(1)
  n2 = right-mid                                           Θ(1)
  create two sorted subarrays L[0..n1] and R[0..n2]        Θ(1)
  for i=0 to n1-1                                          Θ(n1)
    do L[i]=A[left+i]
  for j=0 to n2-1                                          Θ(n2)
    do R[j]=A[mid+j+1]
  L[n1]=∞                                                  Θ(1)
  R[n2]=∞                                                  Θ(1)
  i=0                                                      Θ(1)
  j=0                                                      Θ(1)
  for k=left to right                                      Θ(n)
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

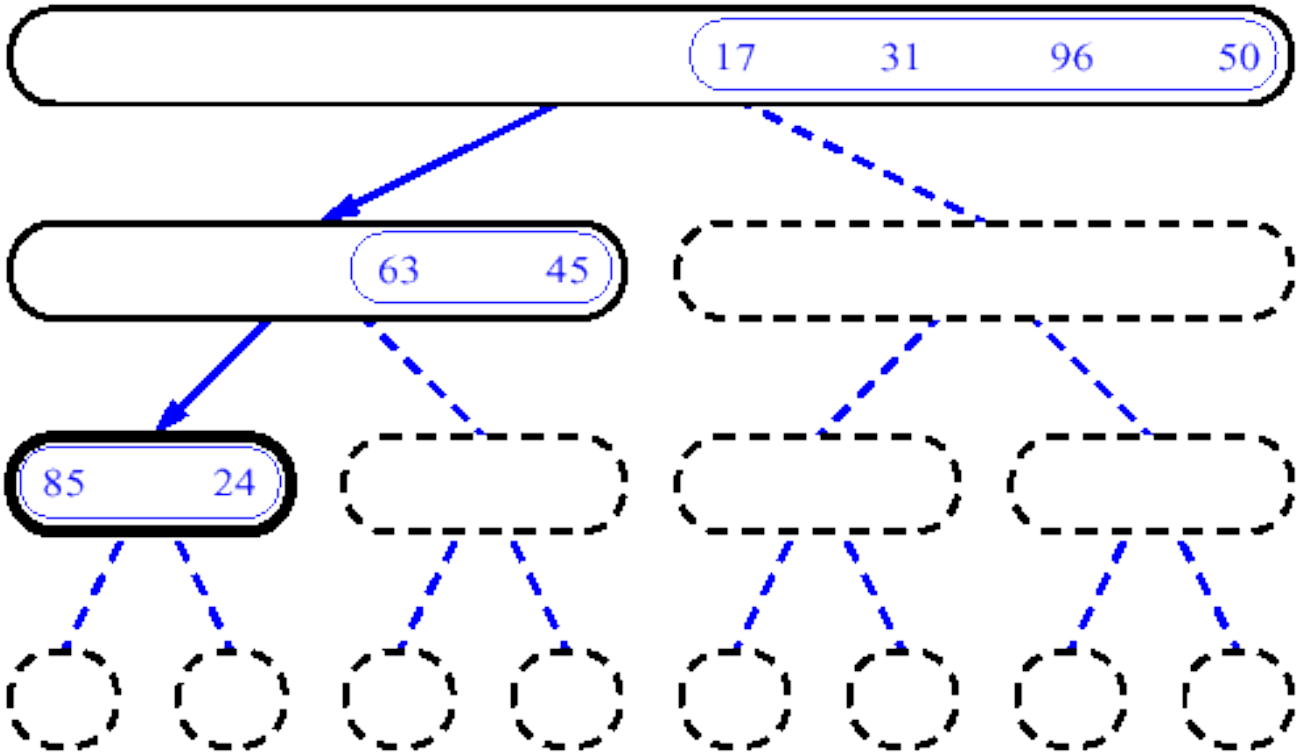
`MergeSort()` running on a array



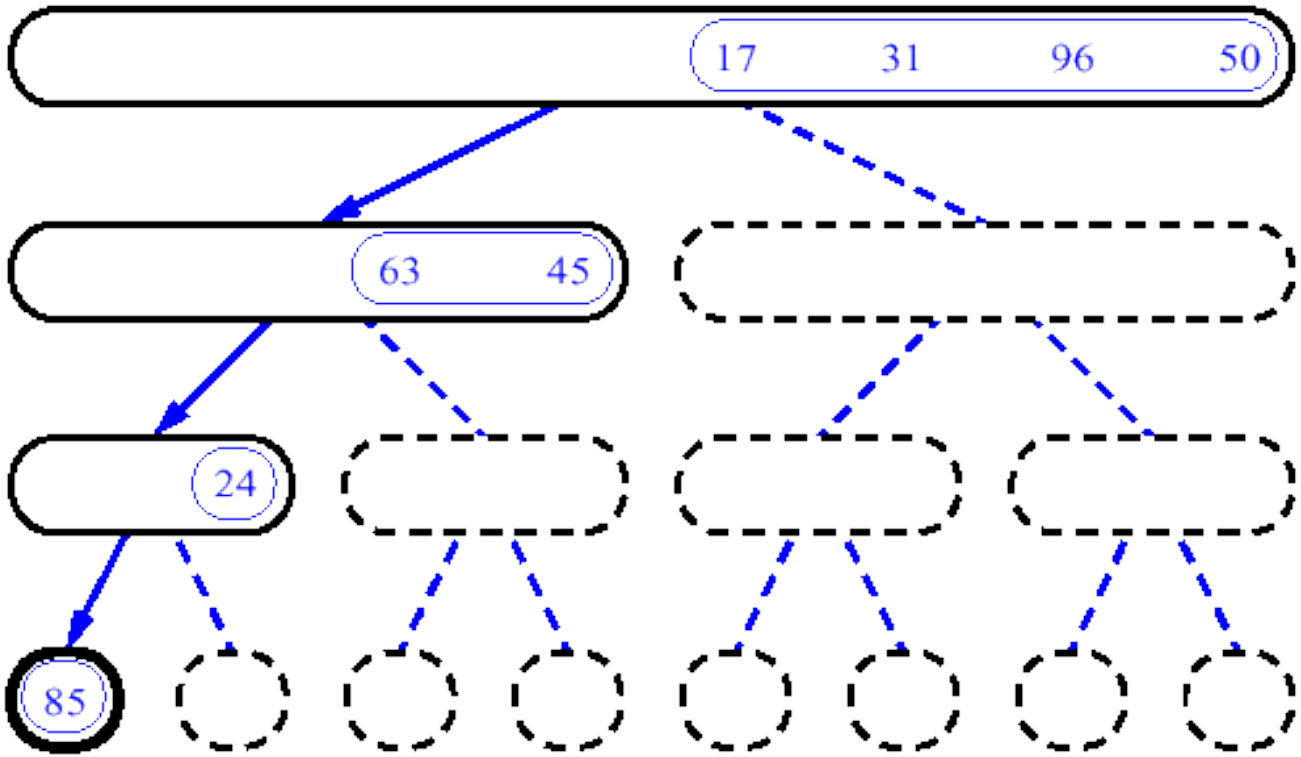
`MergeSort()` running on a array



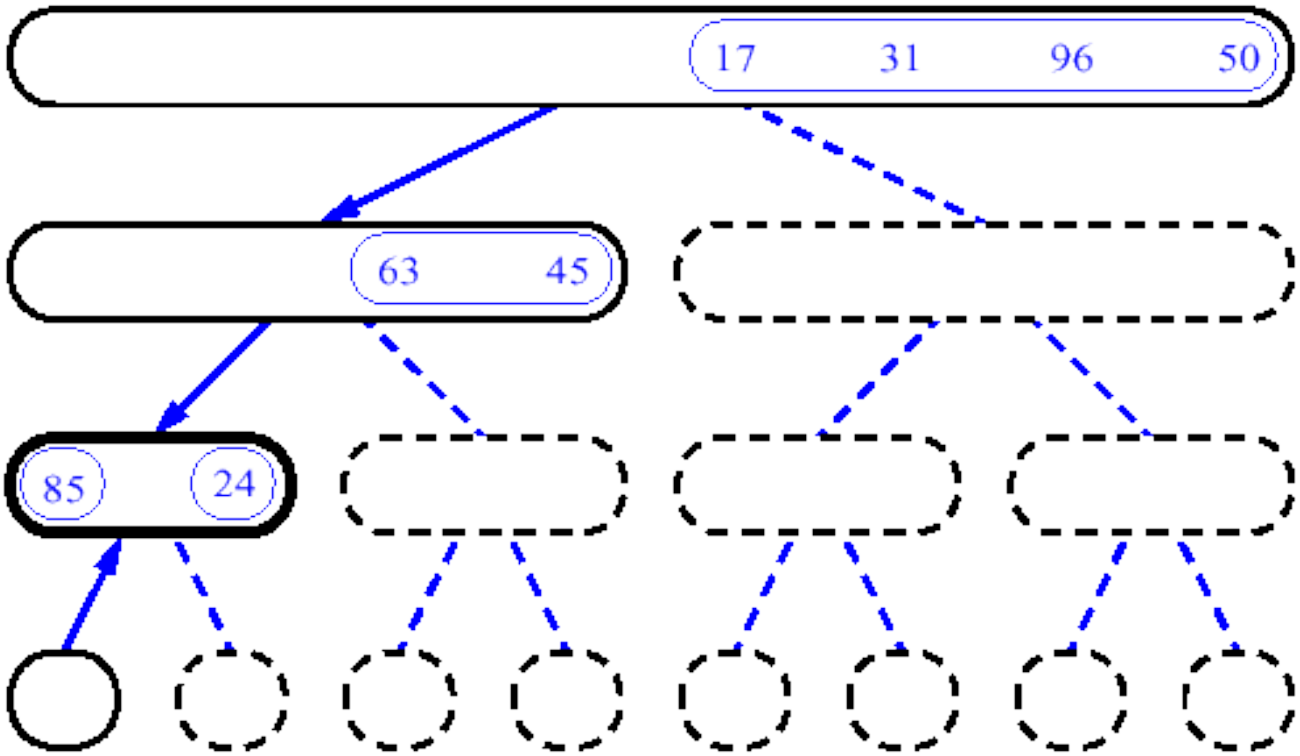
`MergeSort()` running on a array



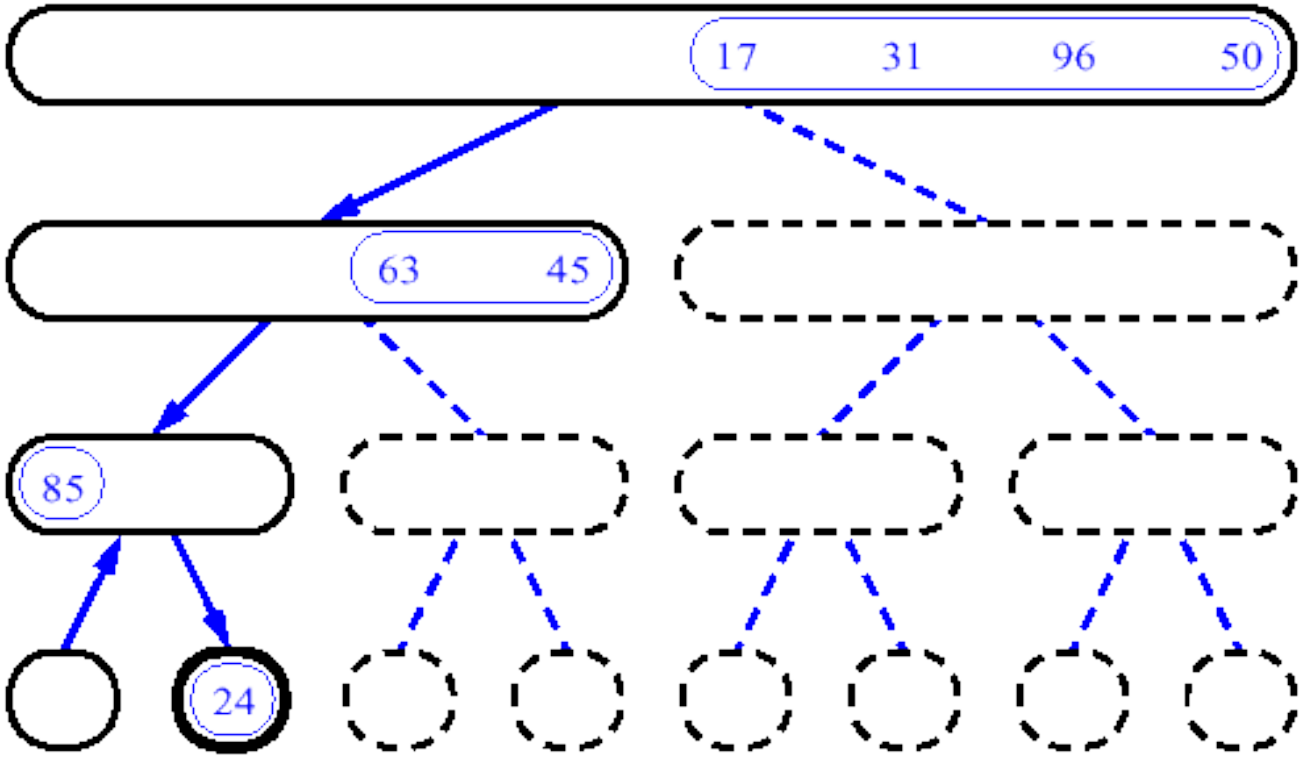
`MergeSort()` running on a array



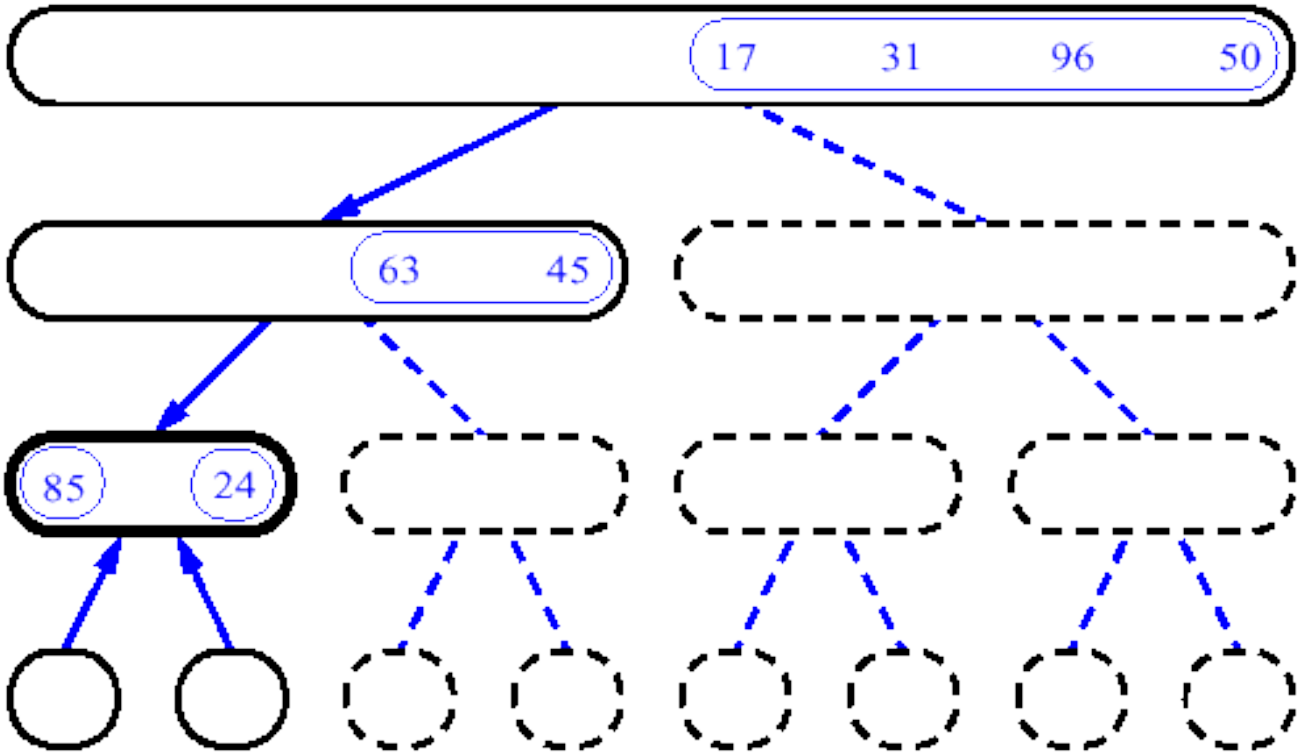
`MergeSort()` running on a array



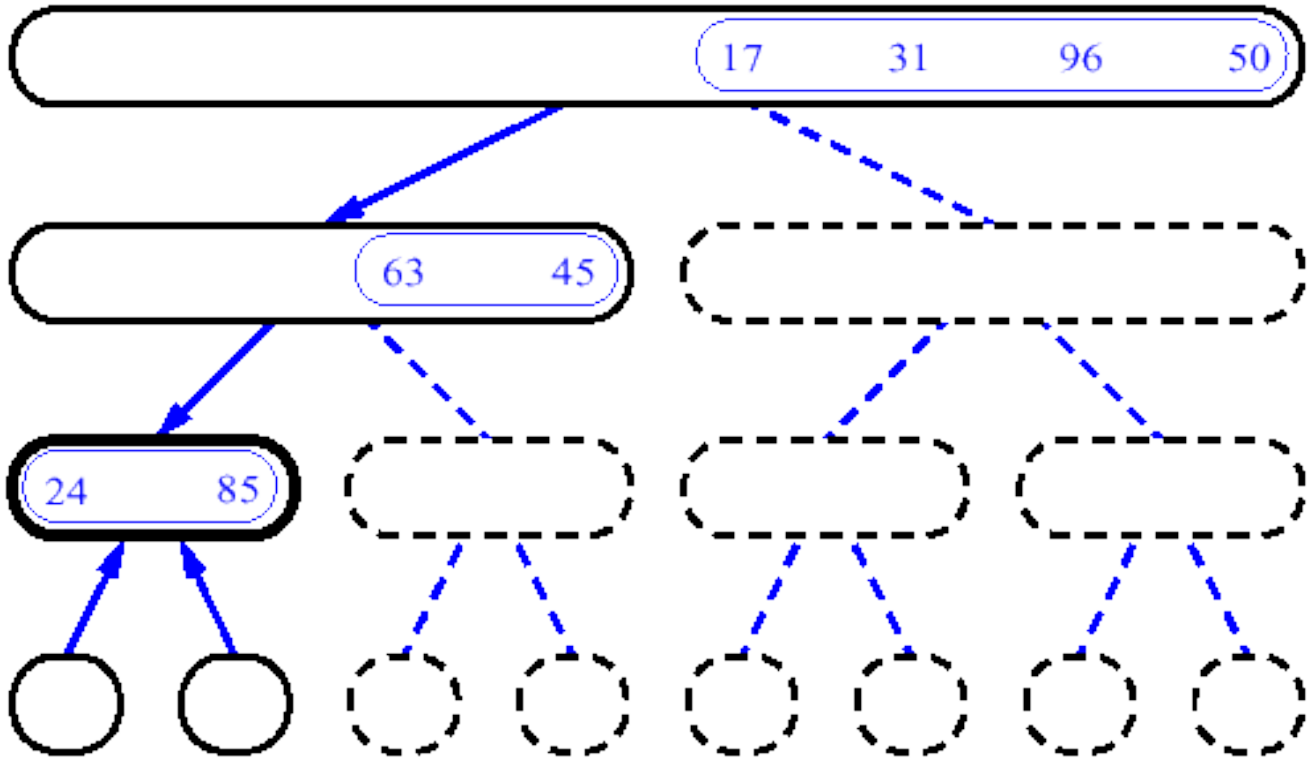
`MergeSort()` running on a array



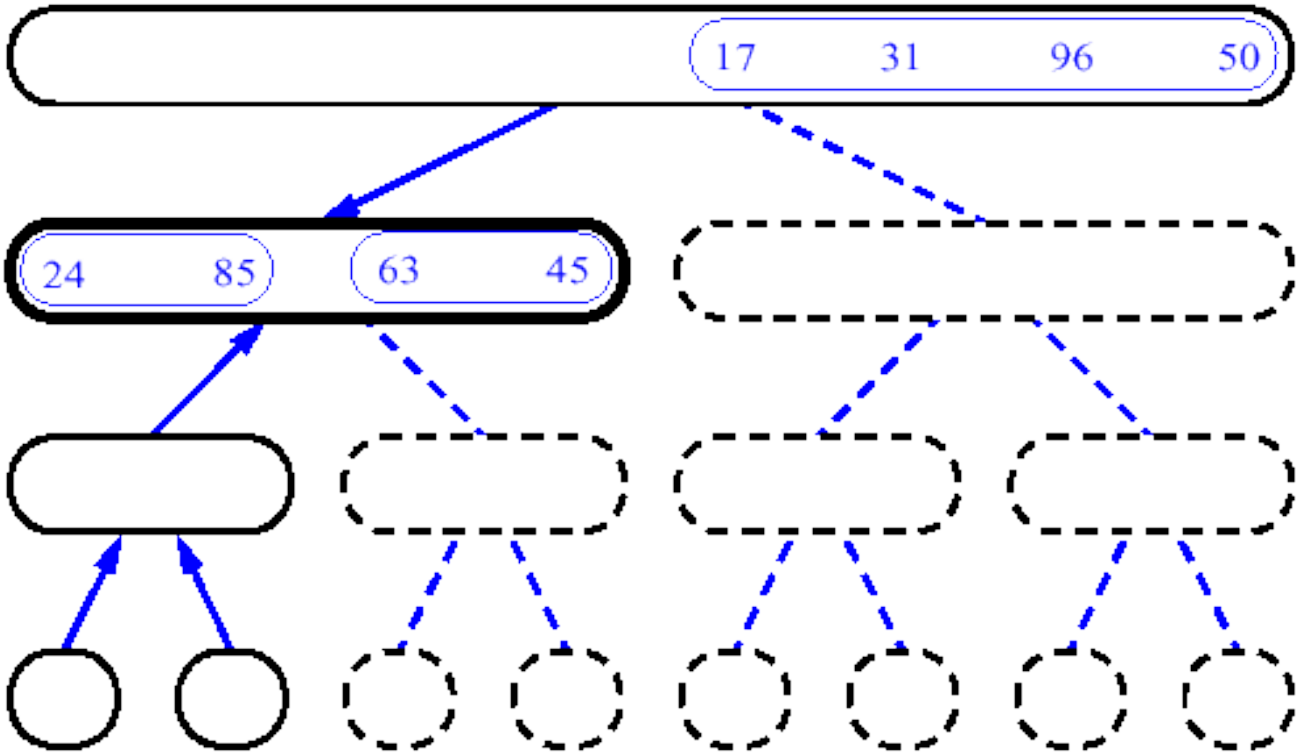
`MergeSort()` running on a array



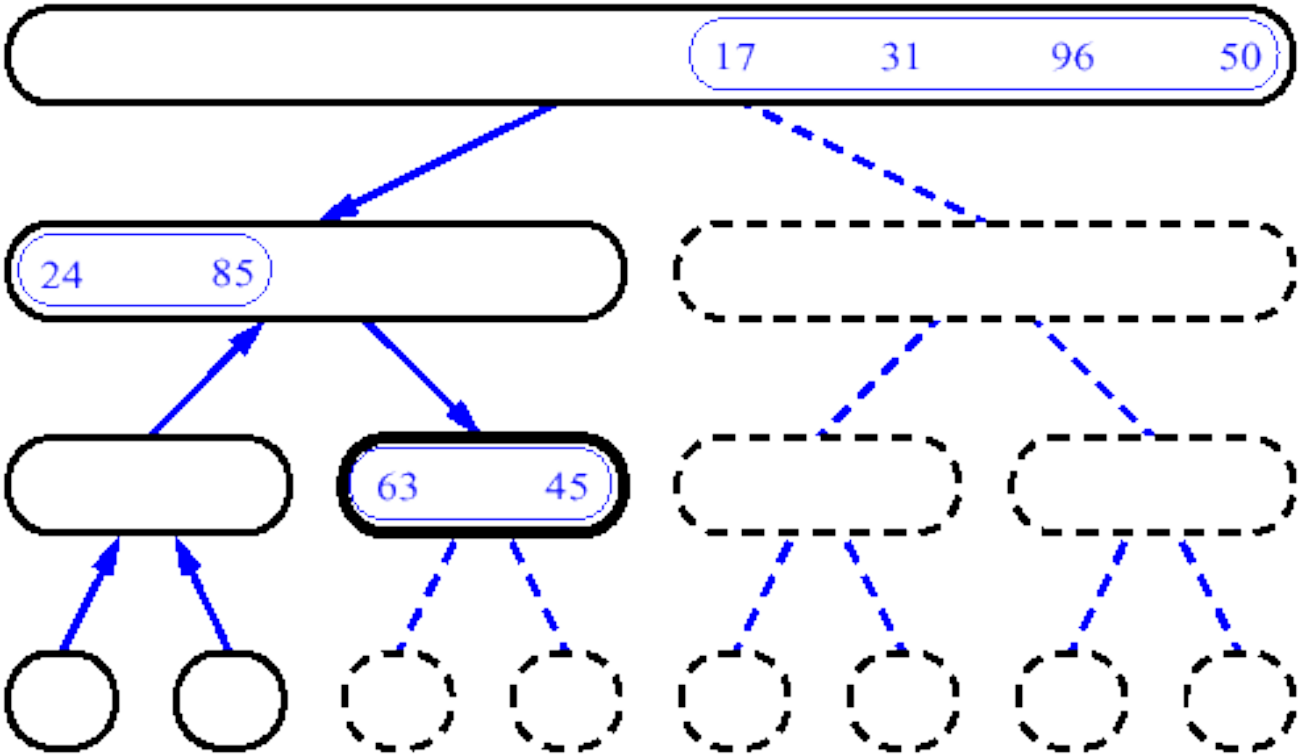
`MergeSort()` running on a array



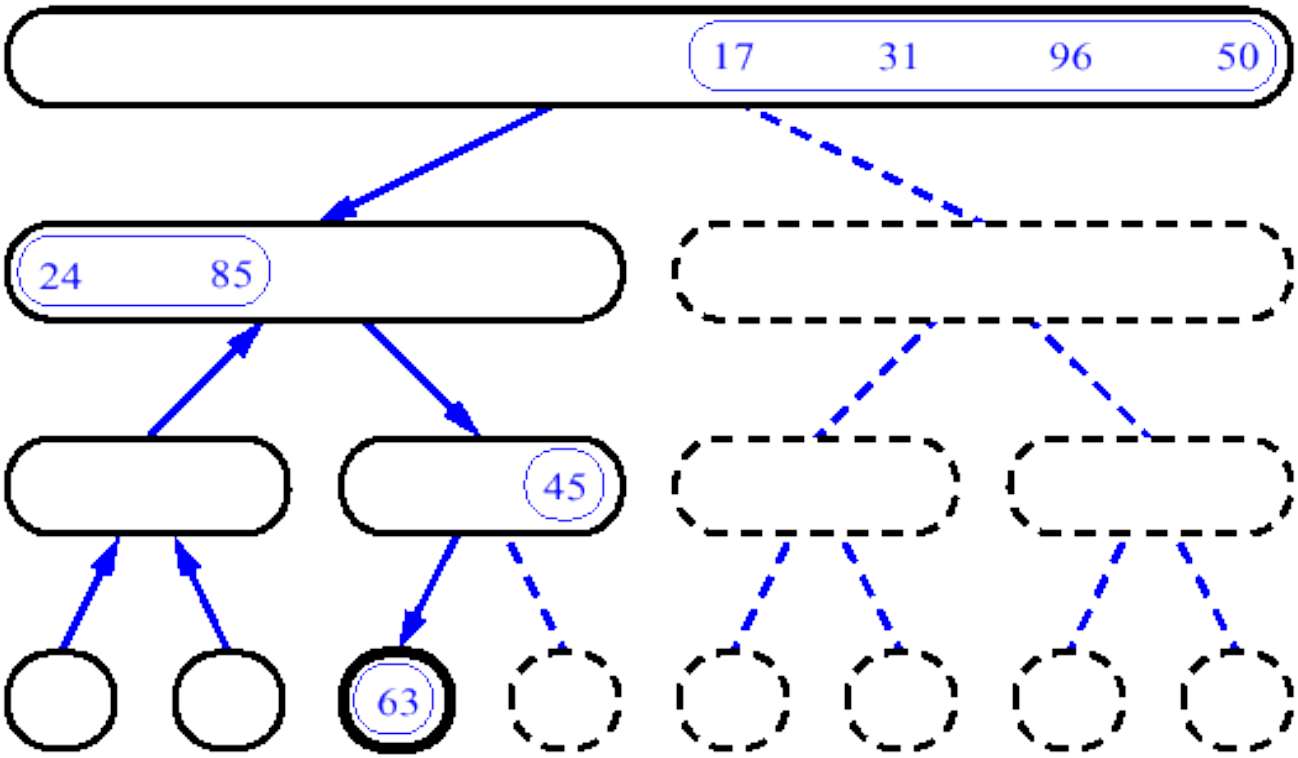
`MergeSort()` running on a array



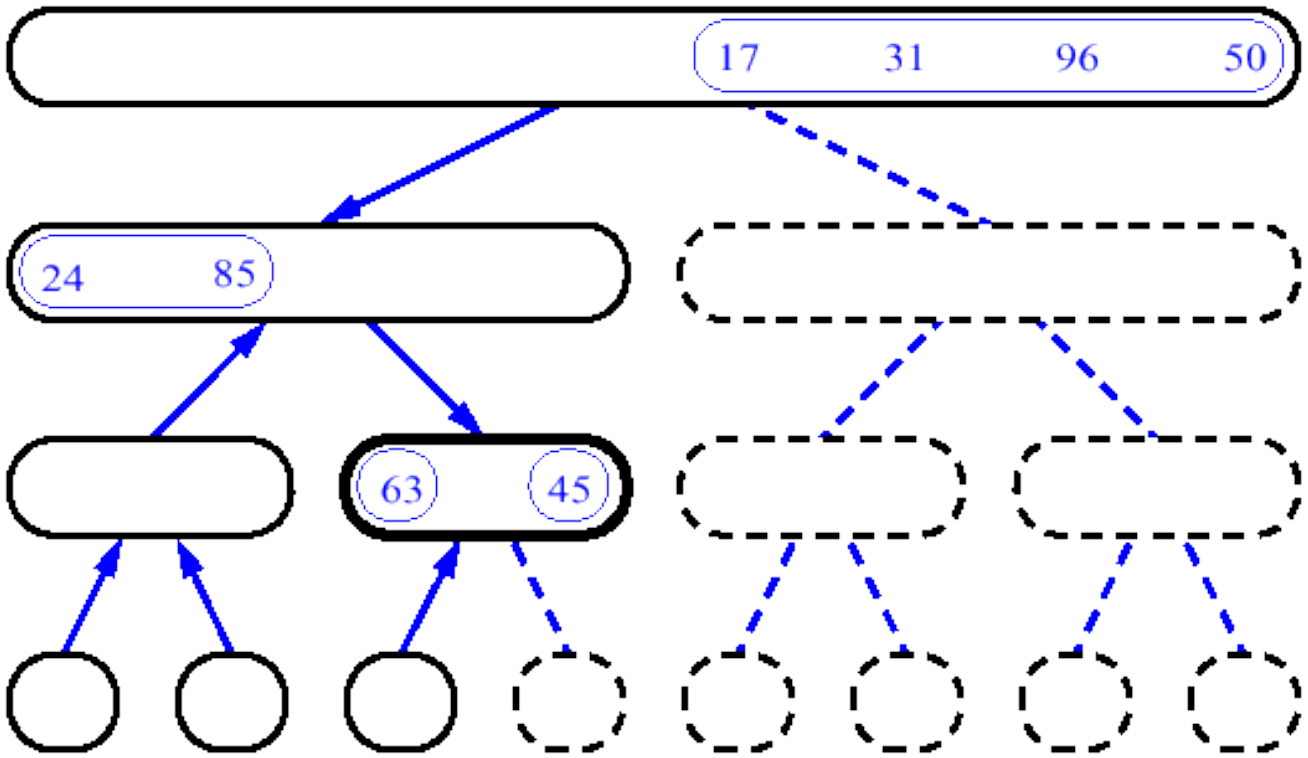
`MergeSort()` running on a array



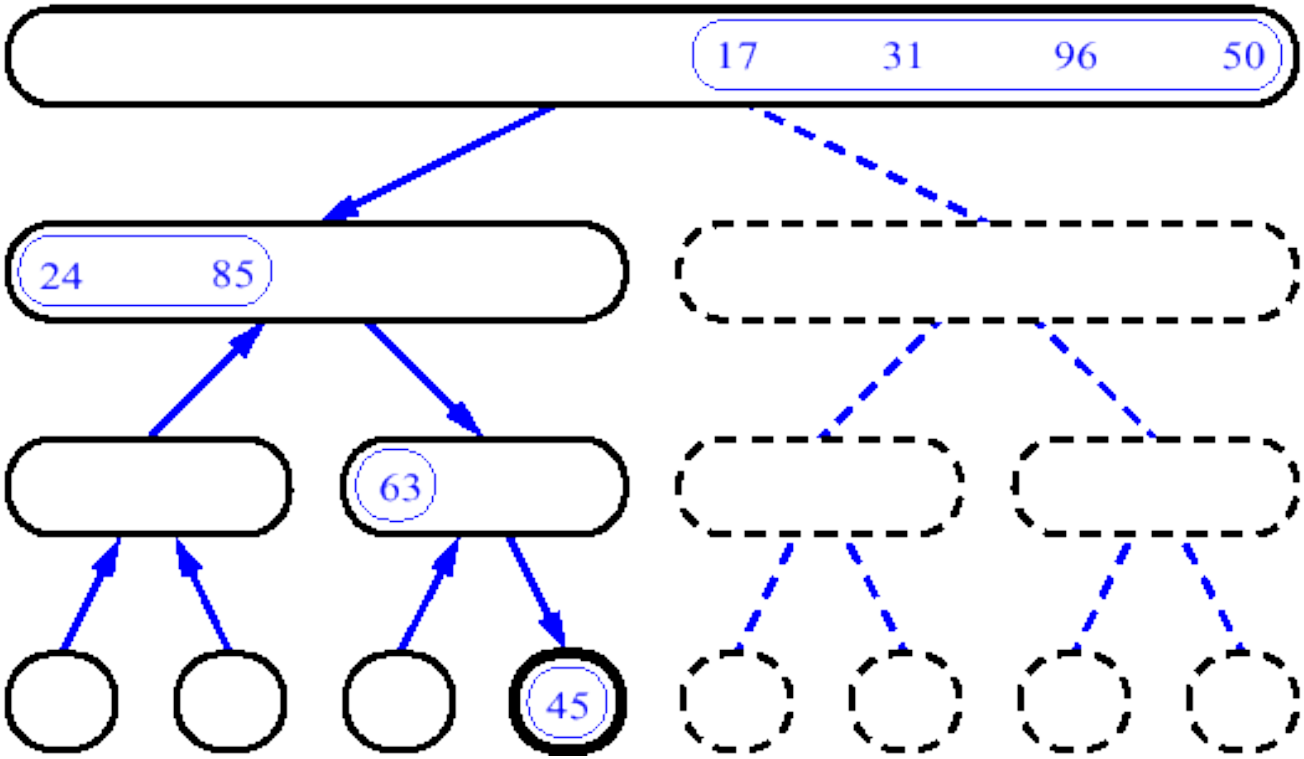
`MergeSort()` running on a array



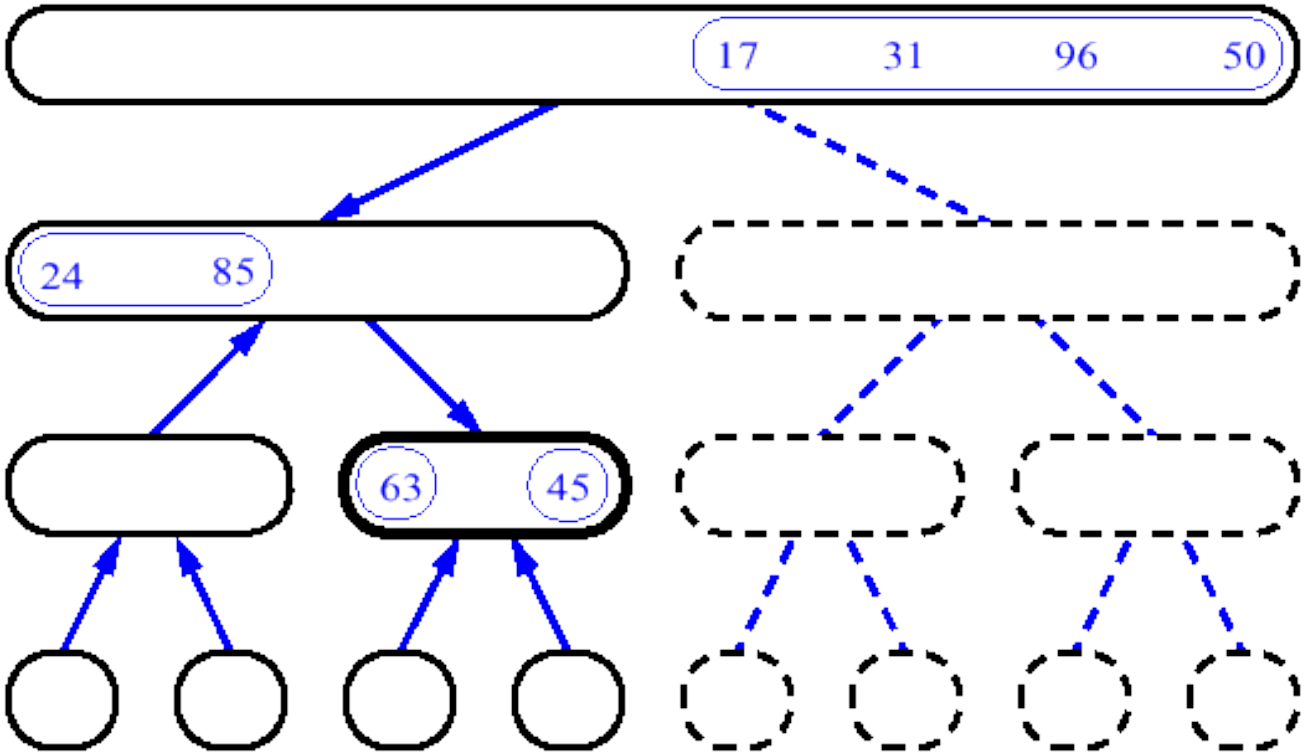
`MergeSort()` running on a array



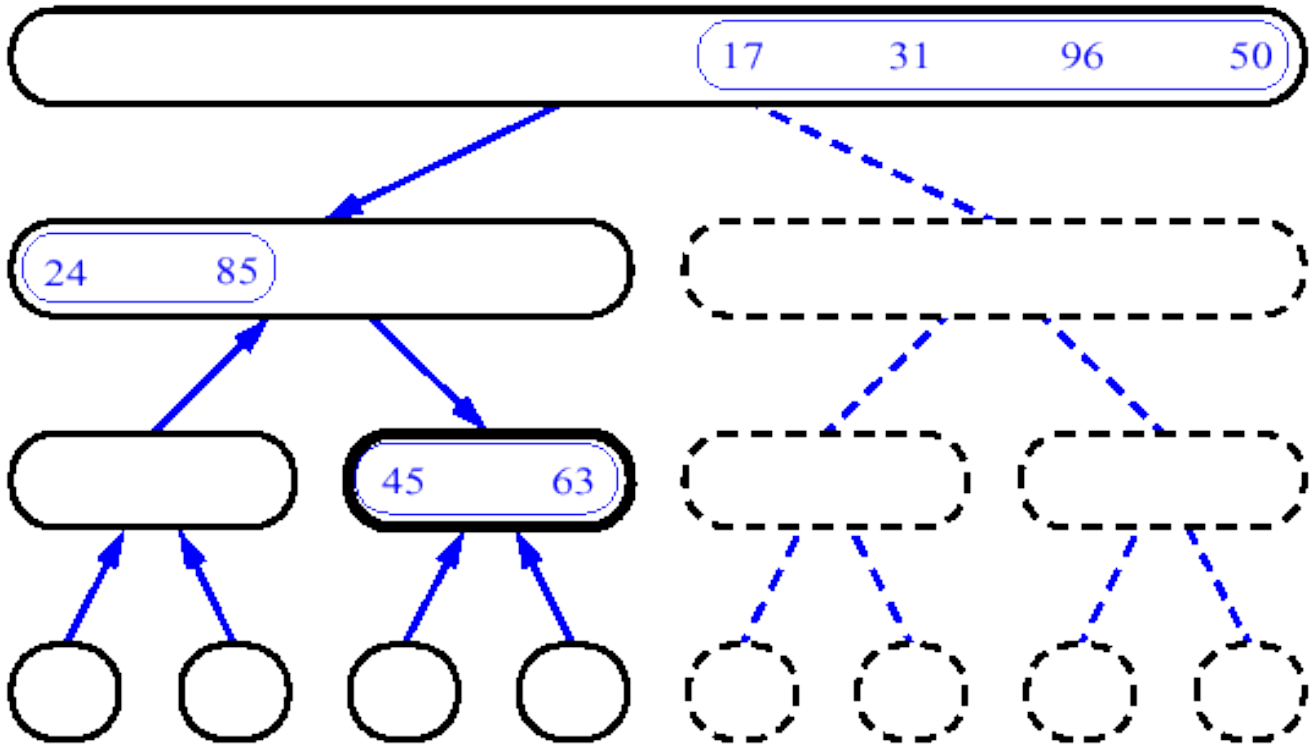
`MergeSort()` running on a array



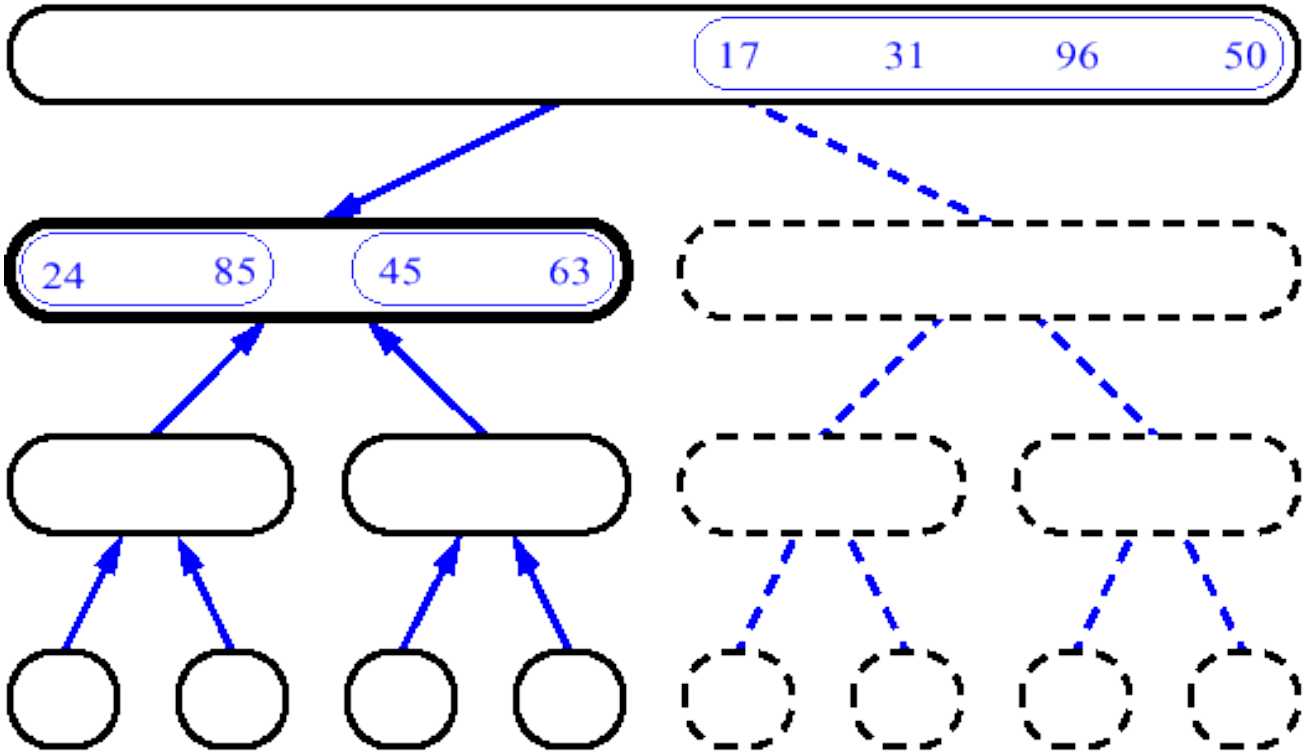
`MergeSort()` running on a array



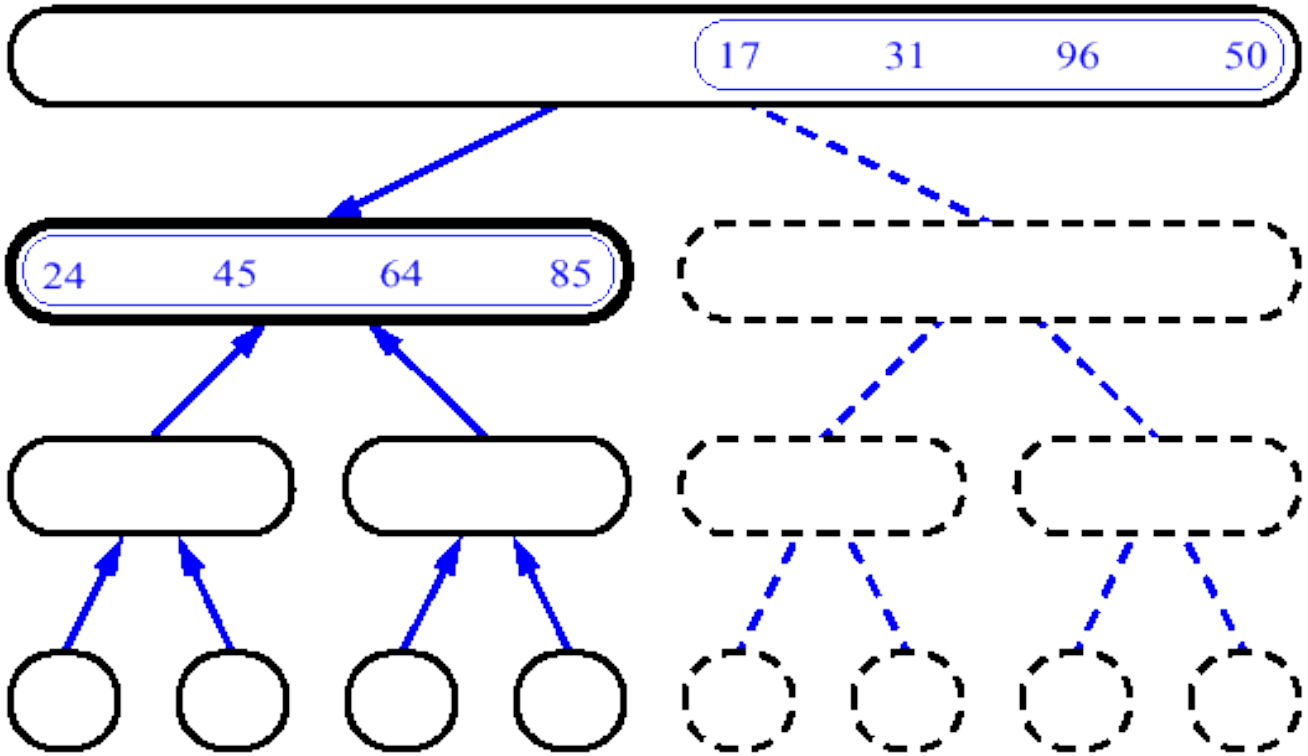
`MergeSort()` running on a array



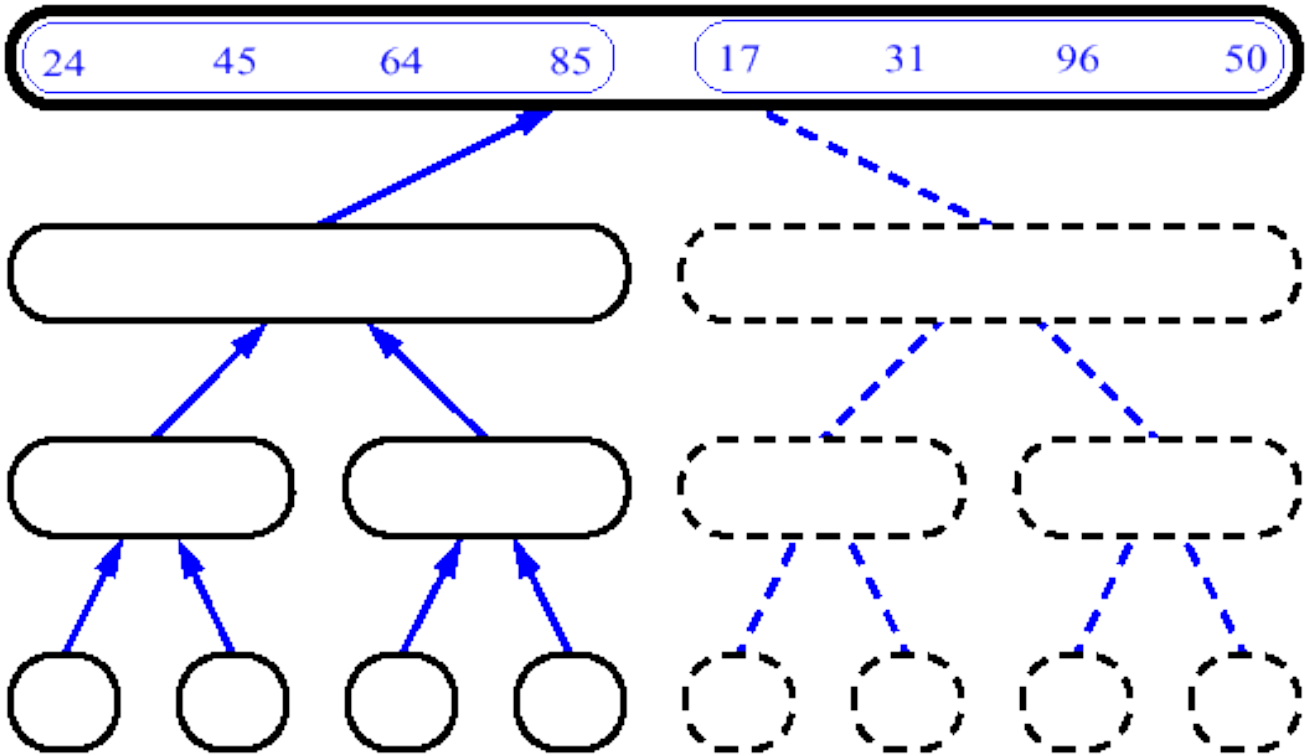
`MergeSort()` running on a array



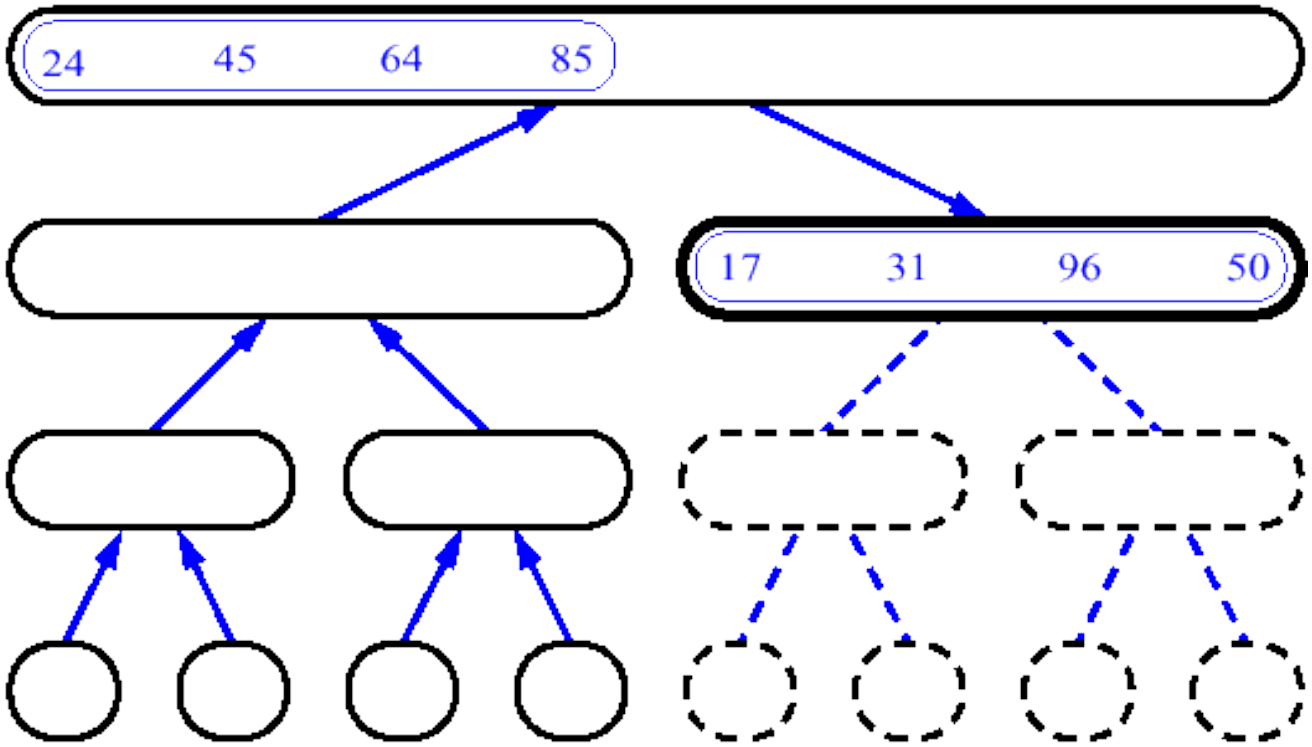
`MergeSort()` running on a array



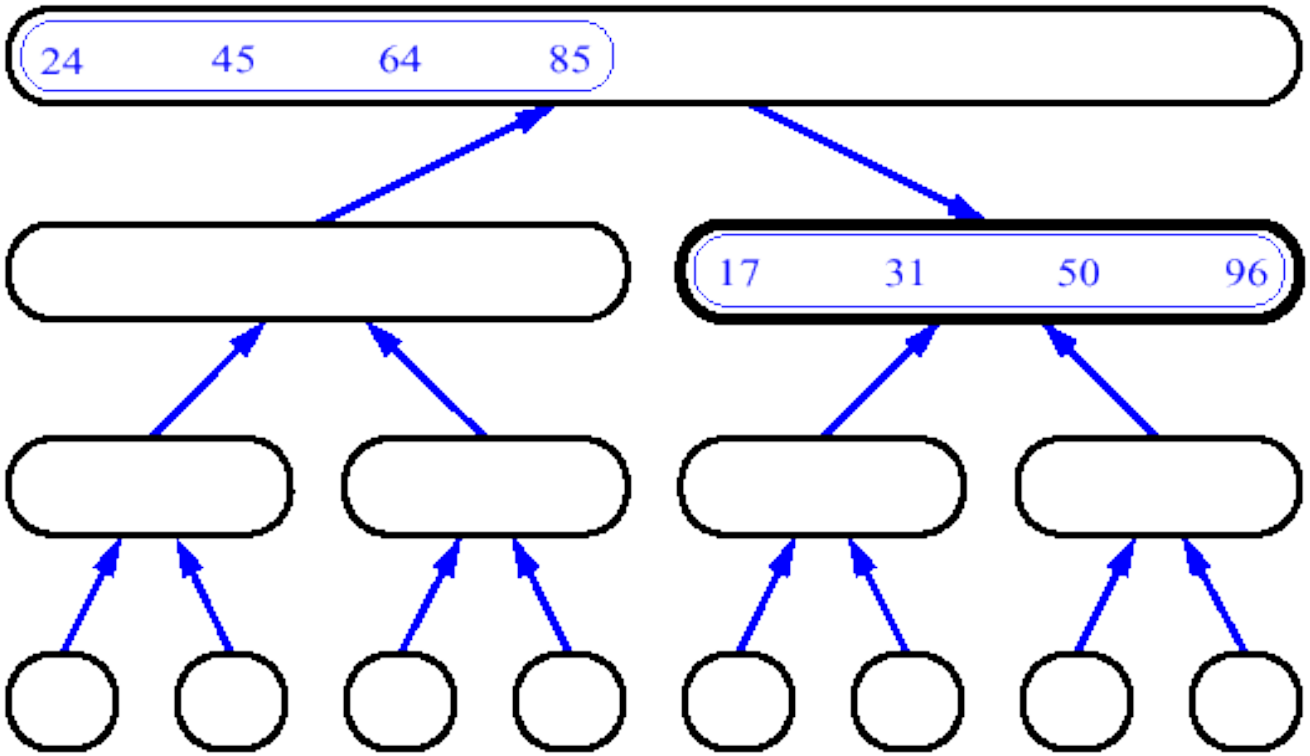
`MergeSort()` running on a array



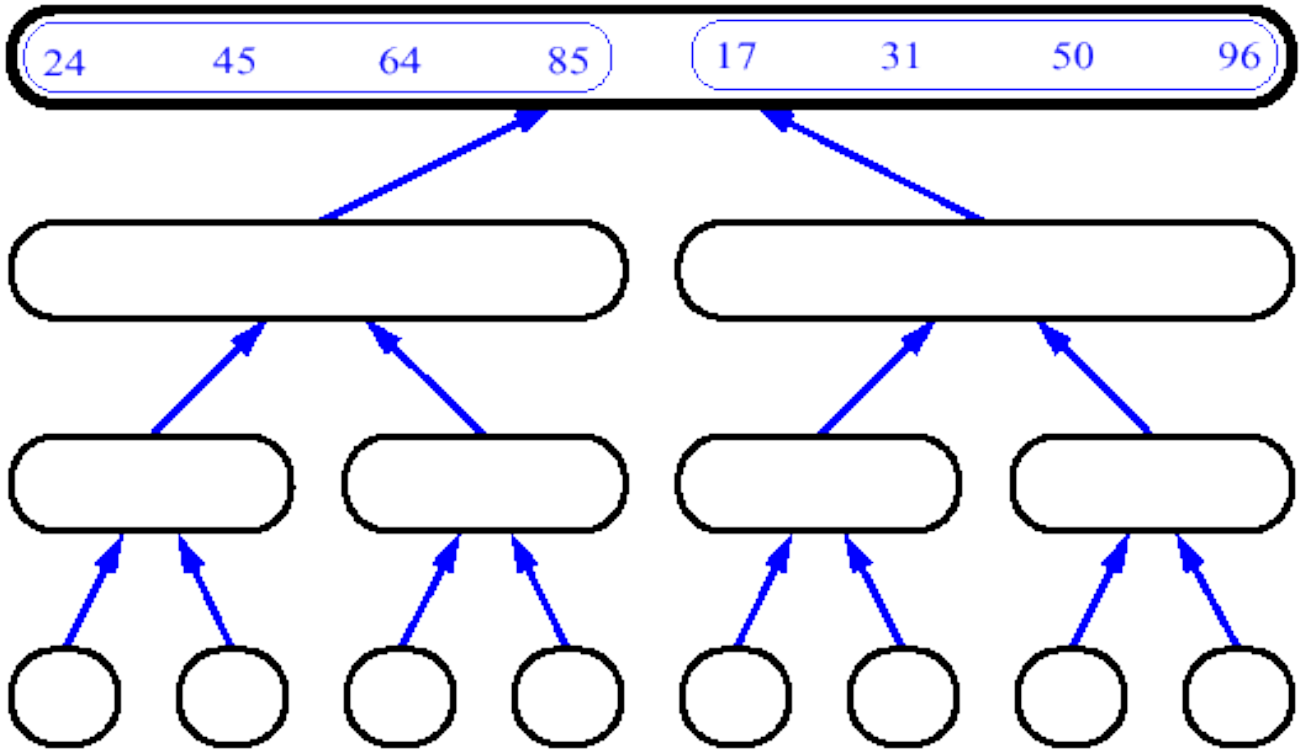
`MergeSort()` running on a array



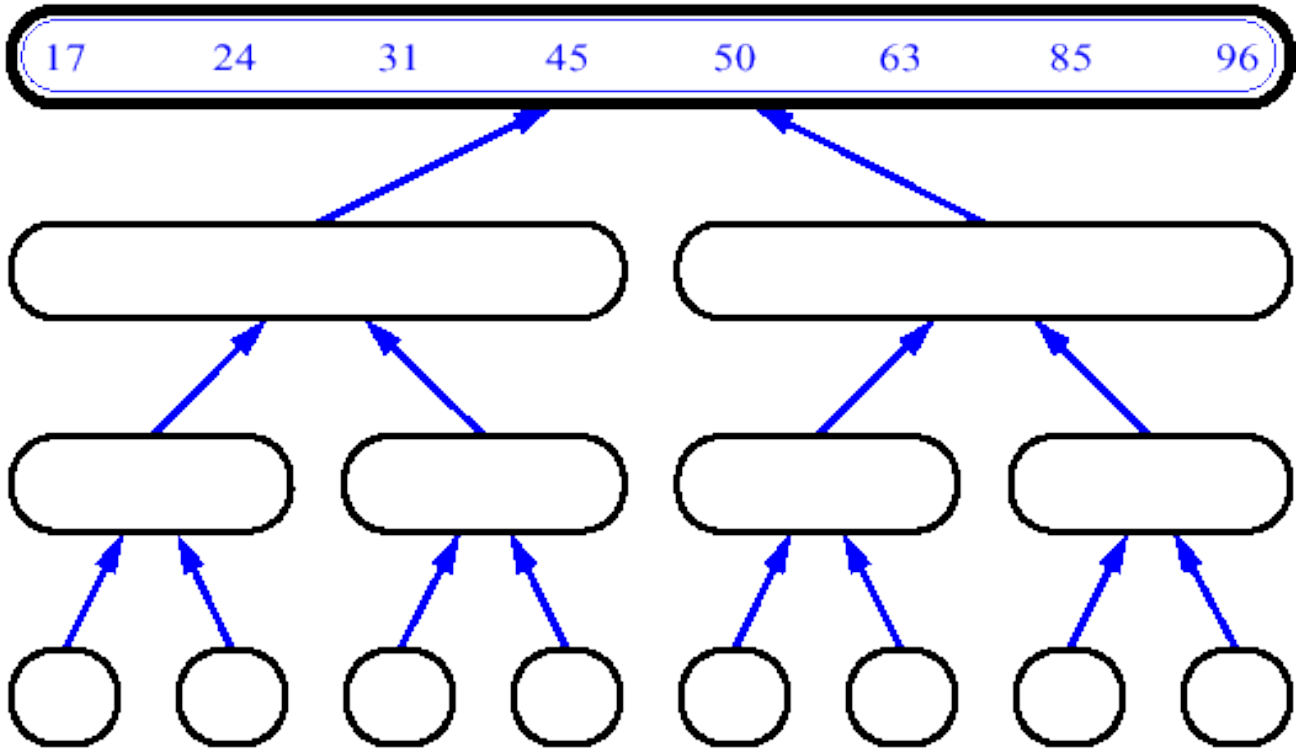
`MergeSort()` running on a array



`MergeSort()` running on a array



`MergeSort()` running on a array



Complexity Analysis of Merge Sort

Statement	Effort

```
MergeSort(A, left, right) {                          T(n)
   if (left < right) {                               Θ(1)
      mid = floor((left + right) / 2);               Θ(1)
      MergeSort(A, left, mid);                       T(n/2)
      MergeSort(A, mid+1, right);                    T(n/2)
      Merge(A, left, mid, right);                    Θ(n)
   }
}
```

- So T(n) = $\Theta(1)$ when n = 1, and
  $2T(n/2) + \Theta(n)$ when n > 1
- So what (more succinctly) is T(n)?

Recurrences

- The expression that represents the **merge sort**:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- is a ***recurrence***.
  - Recurrence: an **equation** or **inequality** that describes a function in terms of its **value on smaller functions**

Recurrences: Factorial

- What is the **recurrence equation** for this algorithm?

```
fac(n) is
   if n = 1 then return 1
   else return fac(n-1) * 1
```

- A recurrence defines **T(n)** in terms of **T** for smaller values

$$T(n) = c \quad \text{if } n=1$$
$$T(n) = T(n-1) + c \quad \text{if } n>1$$

Recurrences: Binary Search

- What is the **recurrence equation** for this algorithm?

```
BinSearch(A[1…n],q)
    if n=1
       then if A[n]=q then return n
                      else return 0

     k←(n+1)/2
     if q < A[k] then BinSearch(A[1…k-1],q)
                 else BinSearch(A[k…n],q)
```

$$T(n) = c \quad \text{if } n=1$$
$$T(n) = c + T(n/2) \quad \text{if } n>1$$

Other Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Solving Recurrences

- Substitution method
- Iteration method
- Master method

Proof by Induction (COMP11120)

- Why is *proof by induction* essential to learn?

```
1 unsigned int N=*;
2 unsigned int i = 0;
3 long double x=2;
4 while( i < N ){
5   x = ((2*x) - 1);
6   ++i;
7 }
8 assert( i == N );
9 assert(x>0);
```

```
1  unsigned int N=*;
2  unsigned int i = 0;
3  long double x=2;
4  if( i < N ){
5    x = ((2*x) - 1);   } k copies
6    ++i;
7  }
8  ...
9  assert( !(i < N) );
10 assert( i == N );
11 assert(x>0);
```

How do we prove this program is **correct**?

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse[1], Lucas

[1] Electronics and Co
[2] Electronic and Information
[3] Department of Comp
e

Abstract. We extended E
symbolic model checking
and multi-threaded ANSI-
verify by induction that th
search for a bounded react

1 Overview

ESBMC is a context-bounded
single- and multi-threaded C c
ESBMC can only be used to fi
prove properties, unless we kn
ever, this is generally not the c
prove safety properties in boun
The details of ESBMC are desc
the differences to the version
on the combination of the k-in

2 Differences to ESBM

Except for the loop handling
version. The main changes co
(where we replaced CBMC's
(where we replaced the name e

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha*, Hussama Ismail†, Lu
*Federal University of Roraima,
E-mail: herbert.rocha@ufrr.b
lucascordeiro@ufam.edu.br,

***Abstract*—We present a proof by induction algorithm, which combines k-induction with invariants to model check embedded C software with bounded and unbounded loops. The k-induction algorithm consists of three cases: in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property ϕ holds in all states reachable within k unwindings; and in the inductive step, we check that whenever ϕ holds for k unwindings, it also holds after the next unwinding of the system. For each step of the k-induction algorithm, we infer invariants using affine constraints (*i.e.*, polyhedral) to specify pre- and post-conditions. Experimental results show that our approach can handle a wide variety of safety properties in typical embedded software applications from telecommunications, control systems, and medical devices; we demonstrate an improvement of the induction algorithm effectiveness if compared to other approaches.**

I. Introduction

The Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) have been applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [1], [2], [3]. The idea behind the BMC techniques is to check the negation of a given property at a given depth, *i.e.*, given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k.

Typically, BMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, *i.e.*, a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BMC techniques

DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs

(Competiti

Williame Rocha[1], Herl
Lucas Cordeiro

[1]Electronic and Information Research
[2]Department of Computer Scien
[3]Department of Computer
[4]Division of Computer Science,

Abstract. DepthK is a software v
duction algorithm that combines k-
to efficiently and effectively verify
DepthK infers program invariants u
sults show that our approach can h
several intricate verification tasks.

1 Overview

DepthK is a software verification tool
and k-induction based on program inv
ing polyhedral constraints. DepthK use
checker that verifies single- and multi-t
tion engine. More specifically, it uses I
a given bound k or to prove correctnes
ever, in contrast to the "plain" ESBMC
polyhedral constraints. It can use the P
PIPS tools [9, 10] to infer these invaria
ers CPAchecker [6] (employed in the
checking verification results.

DepthK pre-processes the C progra
by tracking variables in the loop heade
the C program using either plain BMC
and witness checking. The k-induction
for each step k up to a maximum value,
condition, and inductive step, respective
for a counterexample of the safety pro
forward condition checks whether loop
in all states reachable within k iteratio
for k iterations, then ϕ will also be vali
effectiveness of the k-induction algorit

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Handling Loops in Bounded Model Checking of C Programs via k-Induction

Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro

Electronic and Information Research Center, Federal University of Amazonas, Brazil

Received: date / Revised version: date

Abstract The first attempts to apply the k-induction method to software verification are only recent. In this paper, we present a novel proof by induction algorithm, which is built on the top of a symbolic context-bounded model checker and uses an iterative deepening approach to verify, for each step k up to a given maximum, whether a given safety property ϕ holds in the program. The proposed k-induction algorithm consists of three different cases called *base case*, *forward condition*, and *inductive step*. Intuitively, in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that ϕ holds in all states reachable within k unwindings; and in the inductive step, we check that whenever ϕ holds for k unwindings, it also holds after the next unwinding of the system. The algorithm was implemented in two different ways, a sequential and a parallel one, and the results were compared. Experimental results show that both forms of the algorithm can handle a wide variety of safety properties extracted from standard benchmarks, ranging from reachability to time constraints. And by comparison, the parallel algorithm solves more verification tasks in less time. This paper marks the first application of the k-induction algorithm ulo Theories (SMT) [2] have been successfully applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [3,4,5]. The idea behind the BMC techniques is to check for the violation of a given property at a given depth, i.e., given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k.

Typically, BMC techniques are only able to falsify properties up to the given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BMC techniques limit the visited regions of data structures (e.g., arrays) and the number of loop iterations to a given bound k. This limits the state space that needs to be explored during verification, leaving enough that real errors in applications [3,4,5,6] can be found; BMC tools are, however, susceptible to exhaustion of time or memory limits for programs with loops whose bounds are too large or cannot be determined statically.

Review: Proof by Induction

- Show that a property *P* is true for *n* >= *k*
  - Base case
  - Step case or inductive step
- Suppose that
  - *P(k)* is true for a fixed constant *k*
    - Often *k = 0*
  - *P(n)* ⇛ P*(n+1)* for all *n* >= *k*
- Then *P(n)* is true for all *n* >= *k*

Induction Example: Gaussian Closed Form

- Prove *0+1 + 2 + 3 + … + n = n(n+1) / 2*
  - **Basis:** If *n = 0*, then *0 = 0(0+1) / 2*
  - **Inductive hypothesis:** Assume that *0 + 1 + 2 + 3 + … + k = k(k+1) / 2*
  - **Inductive step:** show that if *P(k)* holds, then also *P(k+1)* holds

    (0+1+2+…+k)+(k+1) = **(k+1)((k+1)+1) / 2.**

    o Using the induction hypothesis that *P(k)* holds:

    k(k+1)/2 + (k+1).

    k(k+1)/2 + (k+1) = [k(k+1)+2(k+1)]/2 = (k^2 +3k+2)/2 = (k+1)(k+2)/2

    = **(k+1)((k+1)+1)/2** **hereby showing that indeed *P(k+1)* holds**

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - **Basis:** show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - **Inductive hypothesis:**
    - Assume $a^0 + a^1 + \ldots + a^k = (a^{k+1} - 1)/(a - 1)$
  - **Inductive step:** show that if *P(k)* holds, then also *P(k+1)* holds:

    $a^0 + a^1 + \ldots + a^{k+1} = a^0 + a^1 + \ldots + a^k + a^{k+1}$

    $= (a^{k+1} - 1)/(a - 1) + a^{k+1} = (a^{k+1+1} - 1)/(a - 1)$

The Substitution Method for Solving Recurrences

- The substitution method
  - A.k.a. the "making a good guess method"
  - Guess the form of the answer, then use **induction** to find the constants and show that solution works
  - Examples:

Recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Guess the solution is:

$T(n) = O(n \log n)$?

Prove that $T(n) \leq cn \log n$ for some $c>0$ using induction and for all $n \geq n_0$

The Substitution Method

- Induction requires us to show that the solution remains valid for the limit conditions
- **Base case**: show that the inequality holds for some *n* sufficiently small
  - If $n = 1 \rightarrow T(1) \leq c * 1 * \log 1 = 0$
  - However, $T(n) = 2T(\lfloor n/2 \rfloor) + n \therefore T(1) = 1$
  - But

$n=2 \rightarrow T(2) = 2T(1) + 2 = 4$ and $cn \log n = c * 2 * \log 2 = 2c$

$n=3 \rightarrow T(3) = 2T(1) + 3 = 5$ and $cn \log n = c * 3 * \log 3$

The Substitution Method

- We can start from $T(2)=4$ or $T(3)=5$ using some $c \geq 2$, given that

$T(n) \leq cn \log n$	$T(2) \leq c2 \log 2$	$T(3) \leq c3 \log 3$

- Base case holds w.r.t. the asymptotic notation:
  - $T(n) \leq cn \log n$ for $n \geq n_0$
- Hint: extend the boundary conditions to make the inductive hypothesis count for small values of n

The Substitution Method

- **Inductive hypothesis:**
  - Assume that $T(n) \le c\, n \log n$ for $c > 0$ holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

$$T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

The Substitution Method

- Induction: Inequality holds for *n*

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Original recurrence

$$\leq 2(c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor)) + n$$

$$\leq cn \log(n/2) + n$$

$$= cn(\log n - \log 2) + n$$

$$= cn \log n - cn + n$$

$$\leq cn \log n$$

The Substitution Method

- Induction: Inequality holds for *n*

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$
$$\leq 2(c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor)) + n$$
$$\leq cn \log(n/2) + n$$
$$= cn(\log n - \log 2) + n$$
$$= cn \log n - cn + n$$
$$\leq cn \log n$$ ***(holds for c ≥1, upper bound analysis)***

Making a good guess

- Guessing a solution takes **experience** / **creativity**
  - Use **heuristics** to help you become a good guesser
  - Use **recursion trees**
- Consider this example: $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$
  - What is your guess here? $T(n) = O(n \log n)$
  - The term “17” cannot substantially affect the solution
- Prove loose upper and lower bounds, e.g.:
  - Start with $T(n) = \Omega(n)$, then $T(n) = O(n^2)$ until we converge to $T(n) = O(n \log n)$

Changing Variables

- A little **algebraic manipulation** can make unknown recurrence similar to one you have seen before

$$T(n) = 2T(\sqrt{n}) + \lg n$$ *m = log n ∴ n = 2^m*

$$T(2^m) = 2T(2^{m/2}) + m$$ *S(m)=T(2^m)*

$$S(m) = 2S(m/2) + m$$ similar to $$T(n) = 2T(n/2) + n$$

$$T(n) = T(2^m) = S(m)$$ *S(m)=O(m log m)*

$$= O(m \log m) = O(\log n \log\log n)$$

Note that *log 2^m = m* and *log $2^{m/2}$ = m/2*

Exercise: Recursive Binary Search

- Given the recurrence equation of the binary search:

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

- Guess the solution is: $T(n) = O(\log n)$
- Prove that $T(n) \leq c \log n$ for some $c>0$ using induction and for all $n \geq n_0$

Summary

- Many useful algorithms are recursive in structure:
  - to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems
- We also analysed recurrences using the substitution method
- We have demonstrated that $T(n)$ of merge sort is $\Theta(n \log n)$, where $\log n$ stands for $\log_2 n$