Types of Algorithms

Basic Steps of the complete development of an algorithm

1- Statement of the problem

2-Development of a model

3-Design of an algorithm

4-Correctness of the algorithm

5-Implementation

6-Analysis and complexity of the algorithm

7-Program testing

8-documentation

Statement of the problem

- 1 Do I understand the vocabulary used in the raw formulation?
- What information has been given?
- What do I want to find out?
- How do I recognize a solution?
- What information is missing.and will any of this be of use?
- Is any of the given information worthless?
- What assumptions have been made?

Development of a model

- Which mathematical structures seem best-suited for the problem?
- Are there any other problems that have been solved which resemble this one?

Documentation

- *Golden rule:*

  Document your program the way you want others to document the program which you read

(a) (divide and conquer)

this technique divides a given problem into smaller instances of the same problem, Solves the smaller problems and combine solutions to solve the given problem.

(b) *Greedy algorithm*: An algorithm which always takes the best immediate or local solution while finding the answer. Greedy algorithms will find the over all or globally optimal solution for some optimization problems but may find less than optimal (suboptimal solutions) for some. These algorithms are very easy to design.

(c) *Dynamic programming Algorithm:* It solves sub problems just once and save the solutions in a table. The solution will be retrieve when the same problem is encountered later on.

(d) Back tracking algorithm: An algorithmic technique by trying one of the several possible choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice.

(e) Brach and bound: it is similar to backtracking.In backtracking we try to find one or all configurations modeled as n- tuples which satisfy certain properties.Branch and bound algorithms are oriented more toward optimization. Here a bounding function is developed to prove incorrect choice.

(f) *Approximate algorithm:* An algorithm to solve an optimization problem that runs in polynomial time in the length of input and outputs the a solution that is guaranteed to be close to the optimal solution.

(g) *Randomized algorithm* :An algorithm is randomized if some of the decisions depend upon the output of a randomizer. A randomized algorithm is said to be Las Vegas algorithm If it always produce the same correct output for the same input. If the output differs from run to run for the same input we call it Monte Carlo algorithm.

(h) Genetic Algorithm: It is an effective method for optimization problems.It was introduced by J.H.Holland in 1975..It manipulates bit strings analogously to DNA evolution. Here we represent a possible solution as a chromosome. This technique creates a population of chromosome and applies genetic operators such as mutation and crossover to evolve the solutions in order to find the best.

(i) Parallel algorithm: A parallel algorithm is an algorithm that has been specially written for execution on a computer with two or more processors.

Divide and Conquer algorithms

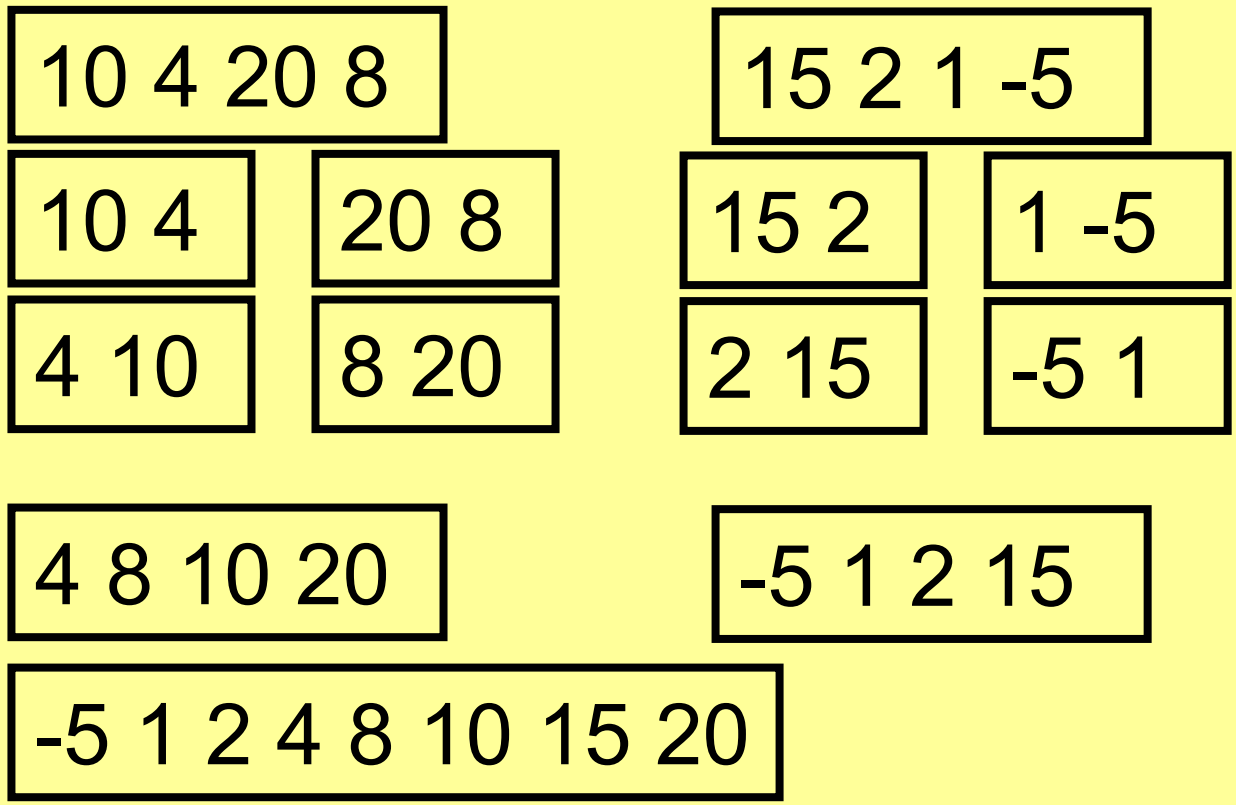
Motivation

- Given a large data set
- (DIVIDE): Partition the data set into smaller sets
- Solve the problems for the smaller sets
- (CONQUER): Combine the results of the smaller sets

Merge Sort

- Sort: 10, 4, 20, 8, 15, 2, 1, -5

- DIVIDE:
- DIVIDE:
- Sort (Conquer)
- COMBINE:
- COMBINE:



- Brute Force: n^2, Merge Sort: $n \log n$

What did we do?

- We are breaking down a larger problem into smaller sets (DIVIDE).
- After solving the smaller set problems (CONQUER), we combine the answers.

General Template

- Divide_and_Conquer (x)

  if *x* is small or simple

  return solve(*x*)

  else

  Decompose *x* into smaller sets x_0, x_1, … x_n

  for (i=0; i<n; i++)

  y_i = Divide_and_Conquer(x_i)

  Recombine y_i's. (We have solved *y* for *x*)

  return *y*

Three Conditions that make D&C worthwhile

1. It must be possible to decompose an instance into sub-instances.

2. It must be efficient to recombine the results.

3. Sub-instances should be about the same size.

MERGE SORT

The procedure merge sort sorts the elements in the sub array A[low,high].

If low≥high the sub array has at most one element and is therefore already sorted.

Otherwise we divide it into A[low,mid] and array[mid+1..high]

Merge sort(A,low,high)

```
1       if low <high
2           then mid←[(low+high)/2]
3           merge sort(A,  low,  mid)
4           merge sort(A,  mid+1,   high)
5           merge (low,  mid,   high)
```

Merging Two Sequences

- Pseudo-code for merging two sorted sequences into a unique sorted sequence

```
Algorithm merge (S1, S2, S):
    Input: Sequence S1 and S2 (on whose elements a
    total order relation is defined) sorted in nondecreas
    ing order, and an empty sequence S.
    Ouput: Sequence S containing the union of the ele
    ments from S1 and S2 sorted in nondecreasing order;
    sequence S1 and S2 become empty at the end of the
    execution
    while S1 is not empty and S2 is not empty do
      if S1.first().element() ≤ S2.first().element() then
        {move the first element of S1 at the end of S}
        S.insertLast(S1.remove(S1.first()))
      else
        { move the first element of S2 at the end of S}
        S.insertLast(S2.remove(S2.first()))
    while S1 is not empty do
      S.insertLast(S1.remove(S1.first()))
      {move the remaining elements of S2 to S}
    while S2 is not empty do
      S.insertLast(S2.remove(S2.first()))
```

```
Algorithm Merge (low,mid,high)
{  h:= low;   i:=low;   j:=mid+1;
while((h ≤   mid) and (j ≤ high)) do
{
if (a[h] ≤ a[j] then
{
   b[I]:= a[h]; h:=h+1;
}
Else
{b[I]:=a[j];j:=j+1;
}
I:=I+1;
}
```

```
If(h> mid) then
    for k:=j to high do
{
B[I]:=a[k];I:=I+1;
}
Else
    for k:= h to mid do
{
B[I] :=a[k]; I:= I+1;
}
For k:= low to high do a[k]:= b[k];
}
```

Time for merging is proportional to n.

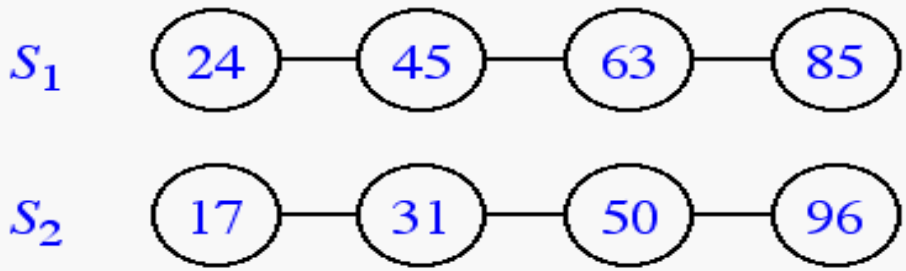
Merge-Sort

- Algorithm:
  - **Divide**: If S has at leas 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.
  - **Recur**: Recursive sort sequences S_1 and S_2.
  - **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

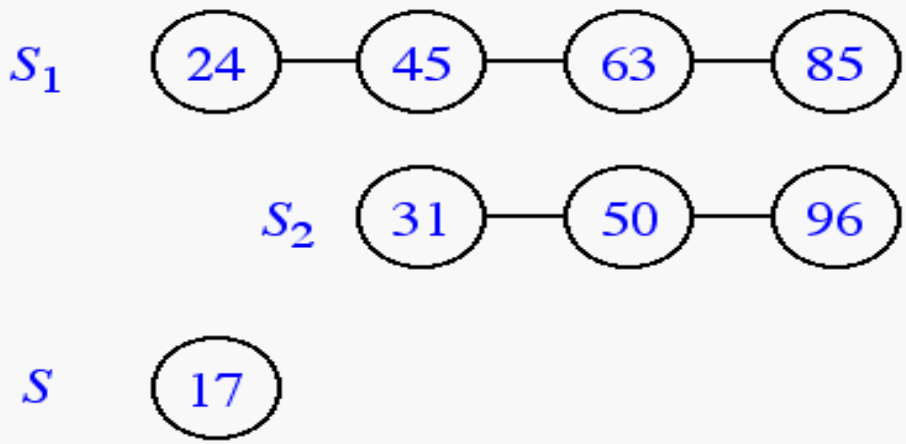
Merging Two Sequences (cont.)

- Some pictures:

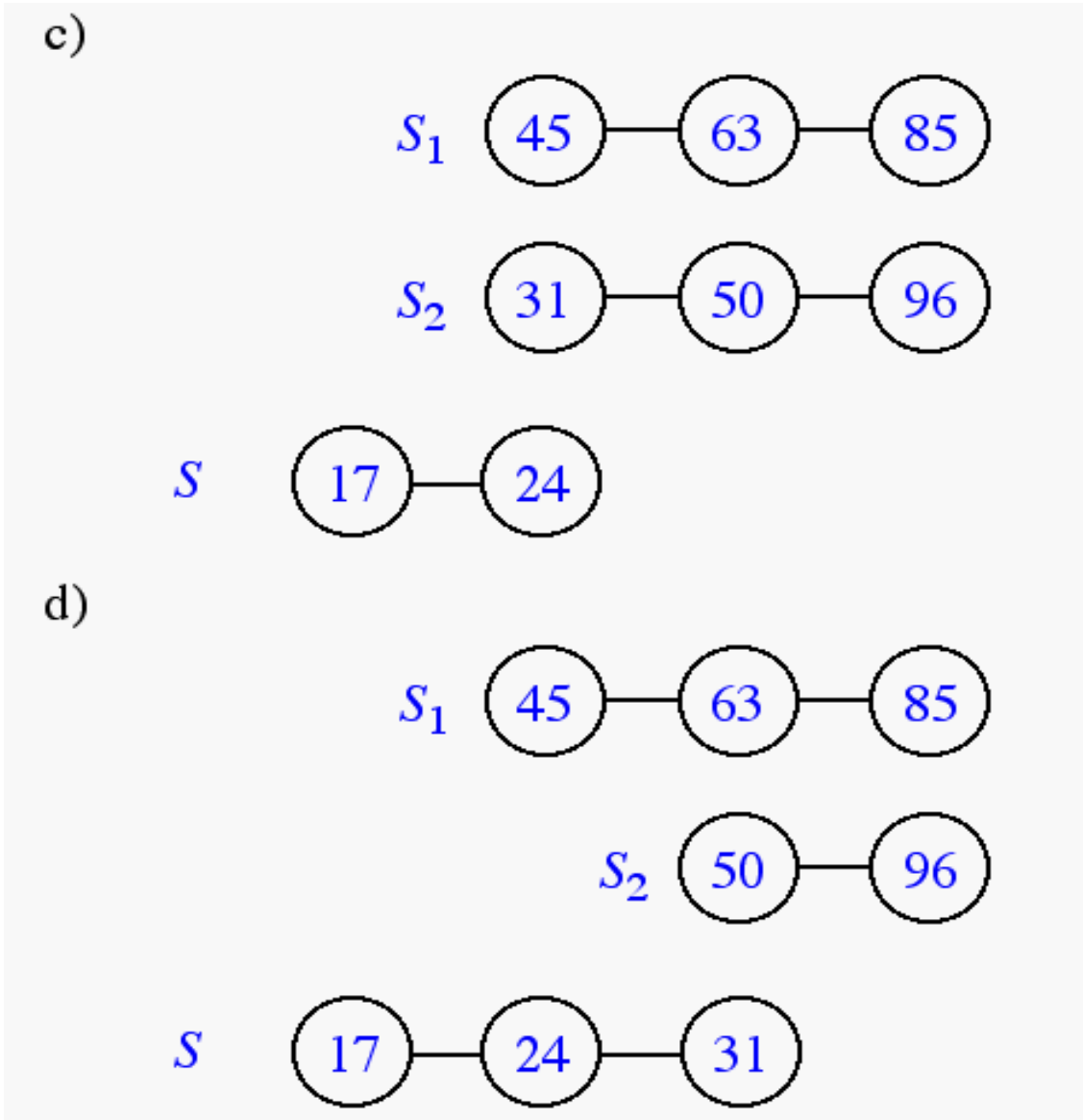
a)



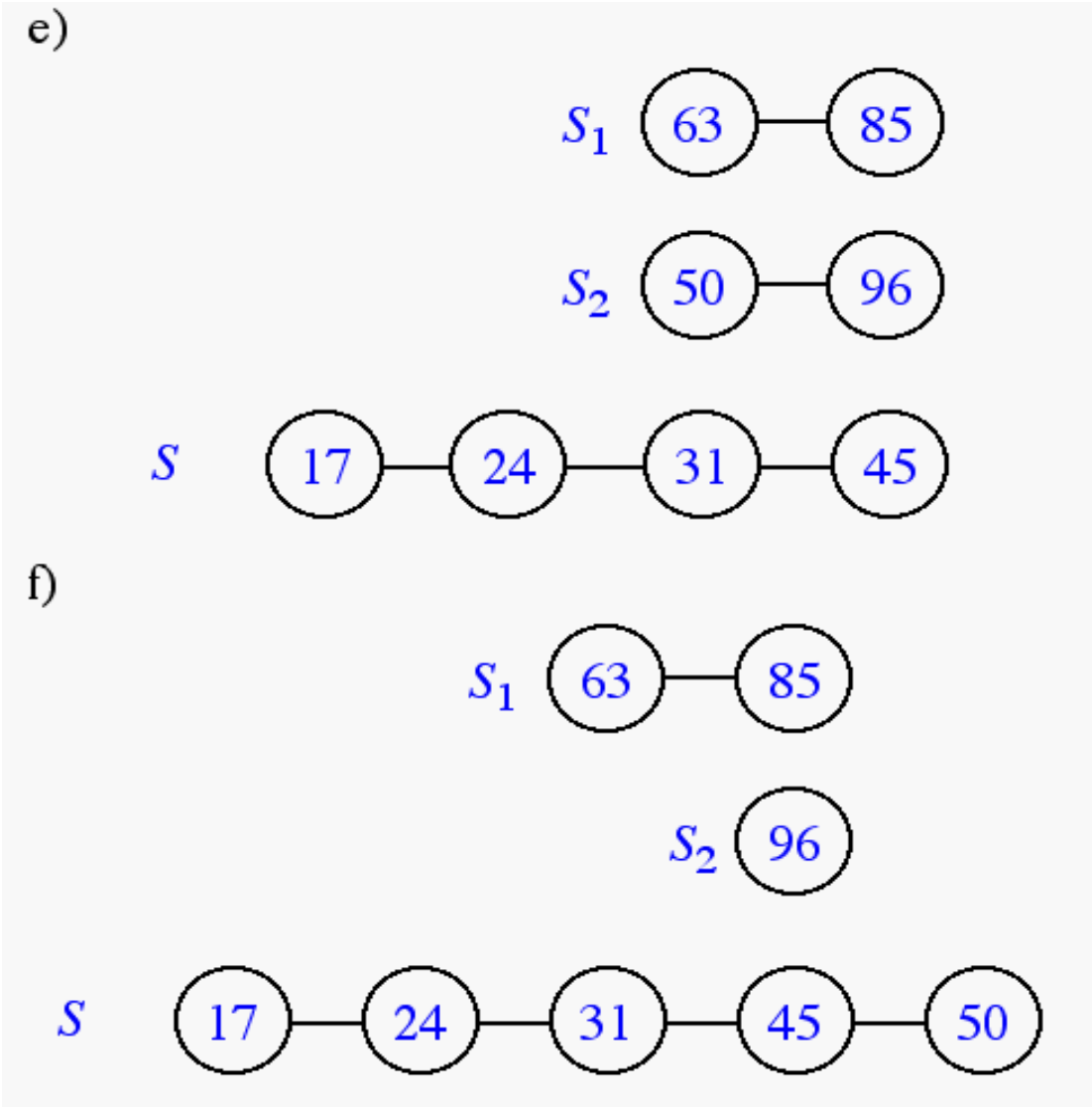
b)



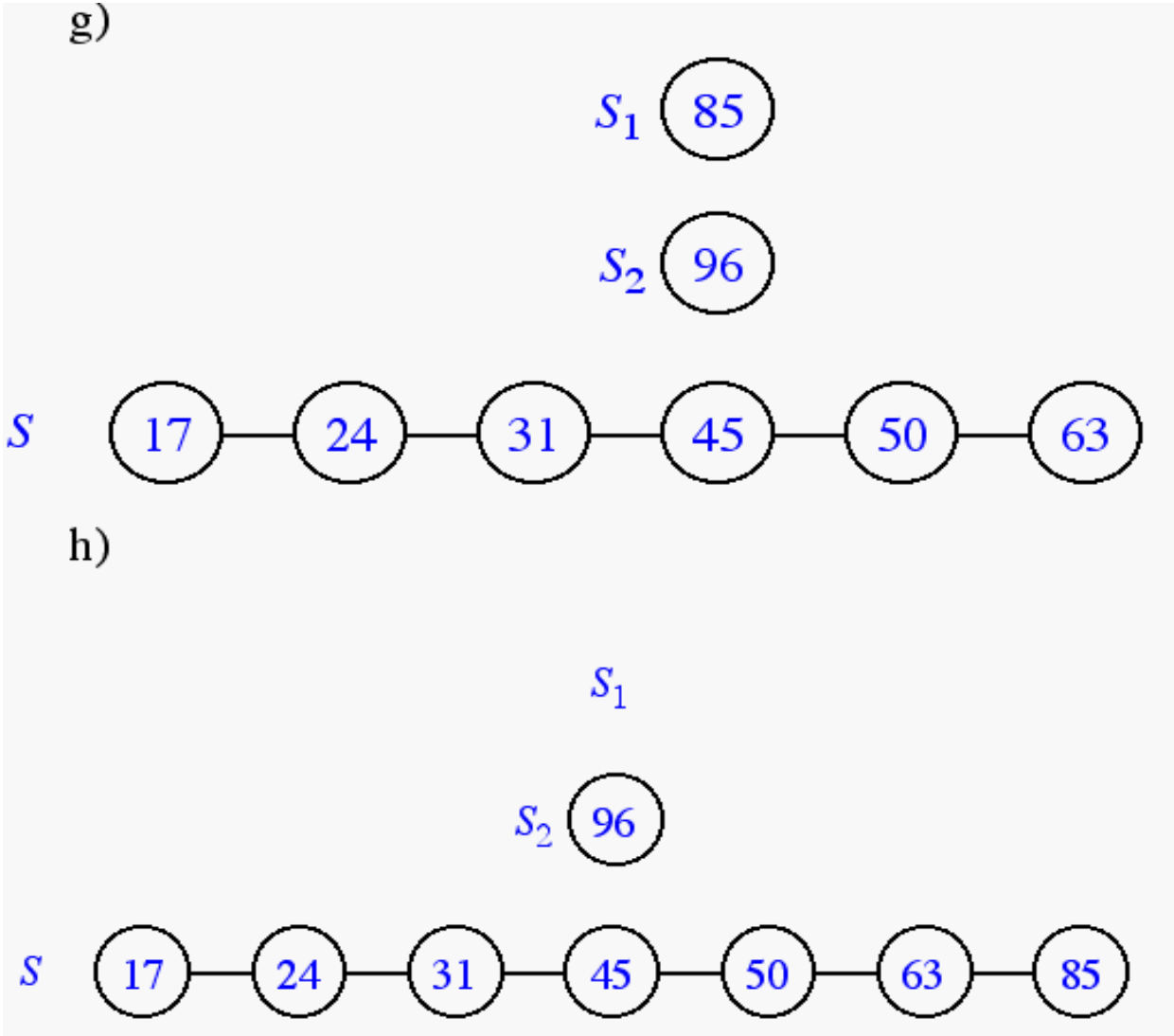
Merging Two Sequences (cont.)



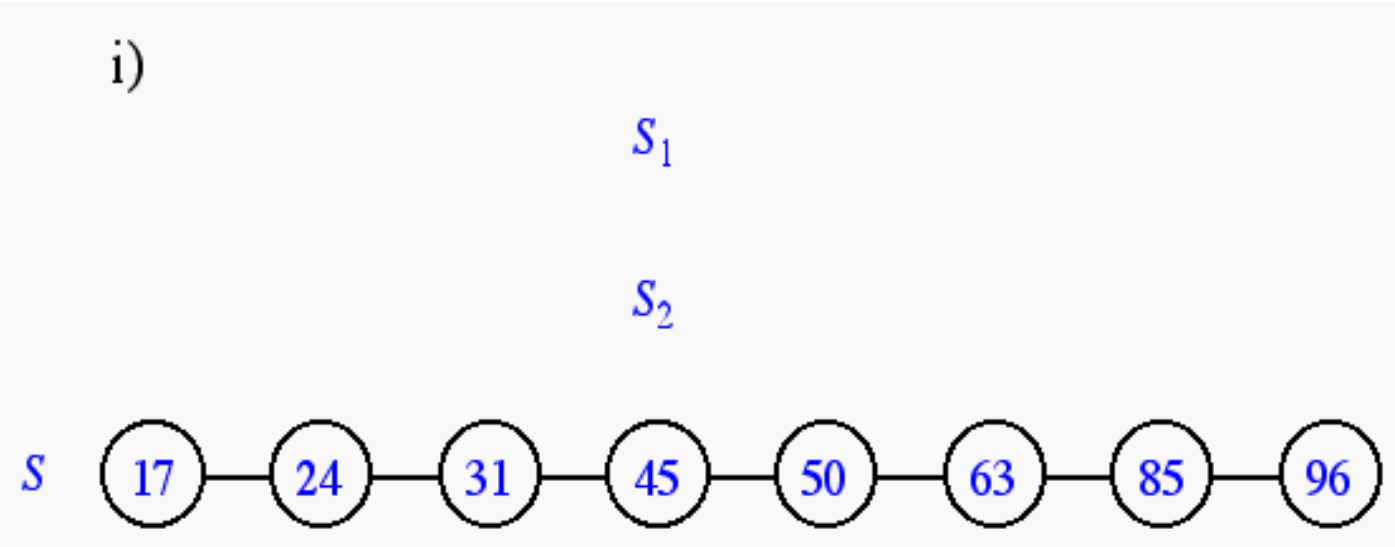
Merging Two Sequences (cont.)



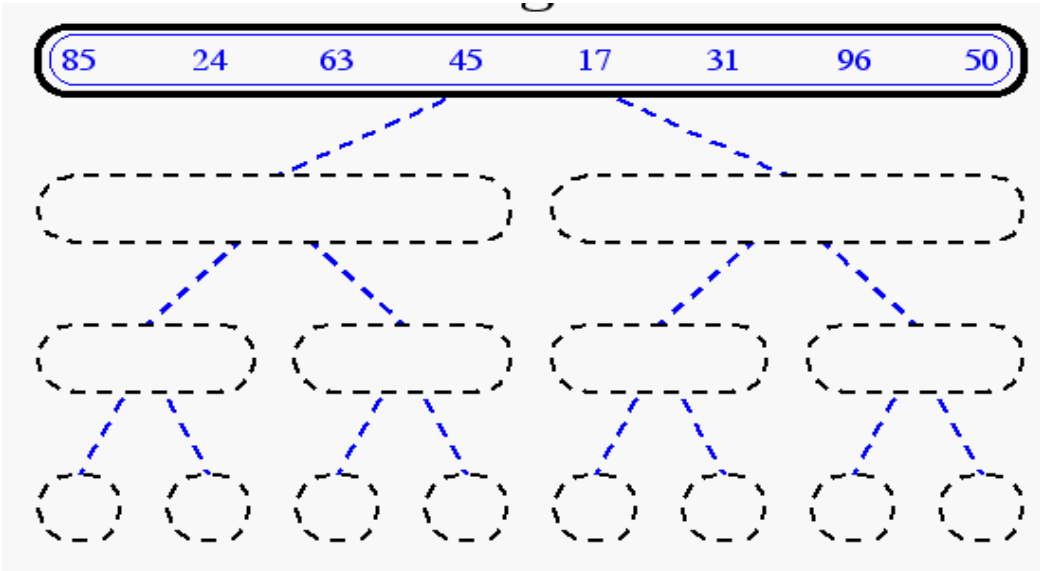
Merging Two Sequences (cont.)

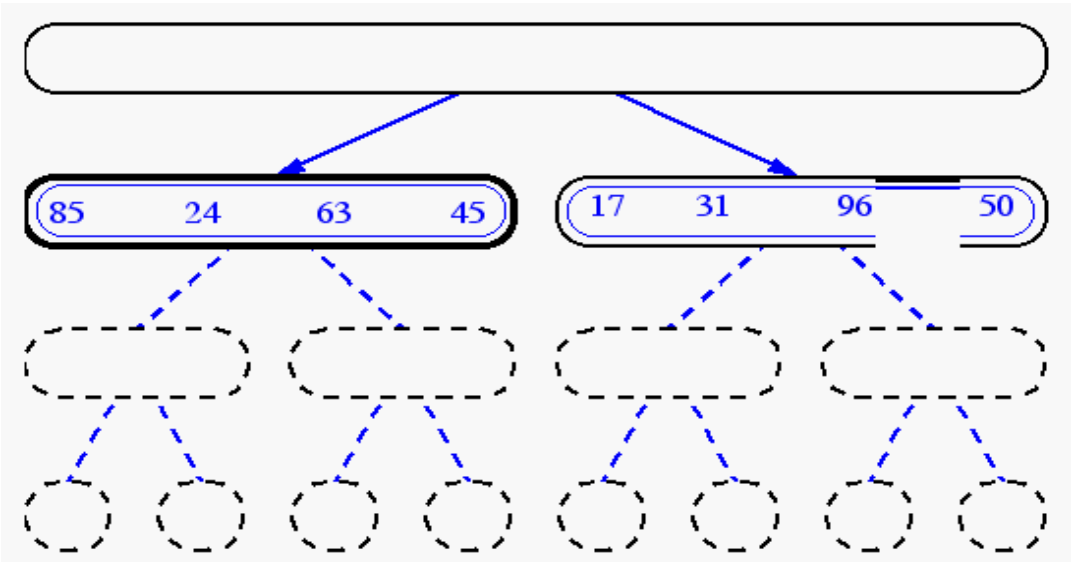


Merging Two Sequences (cont.)

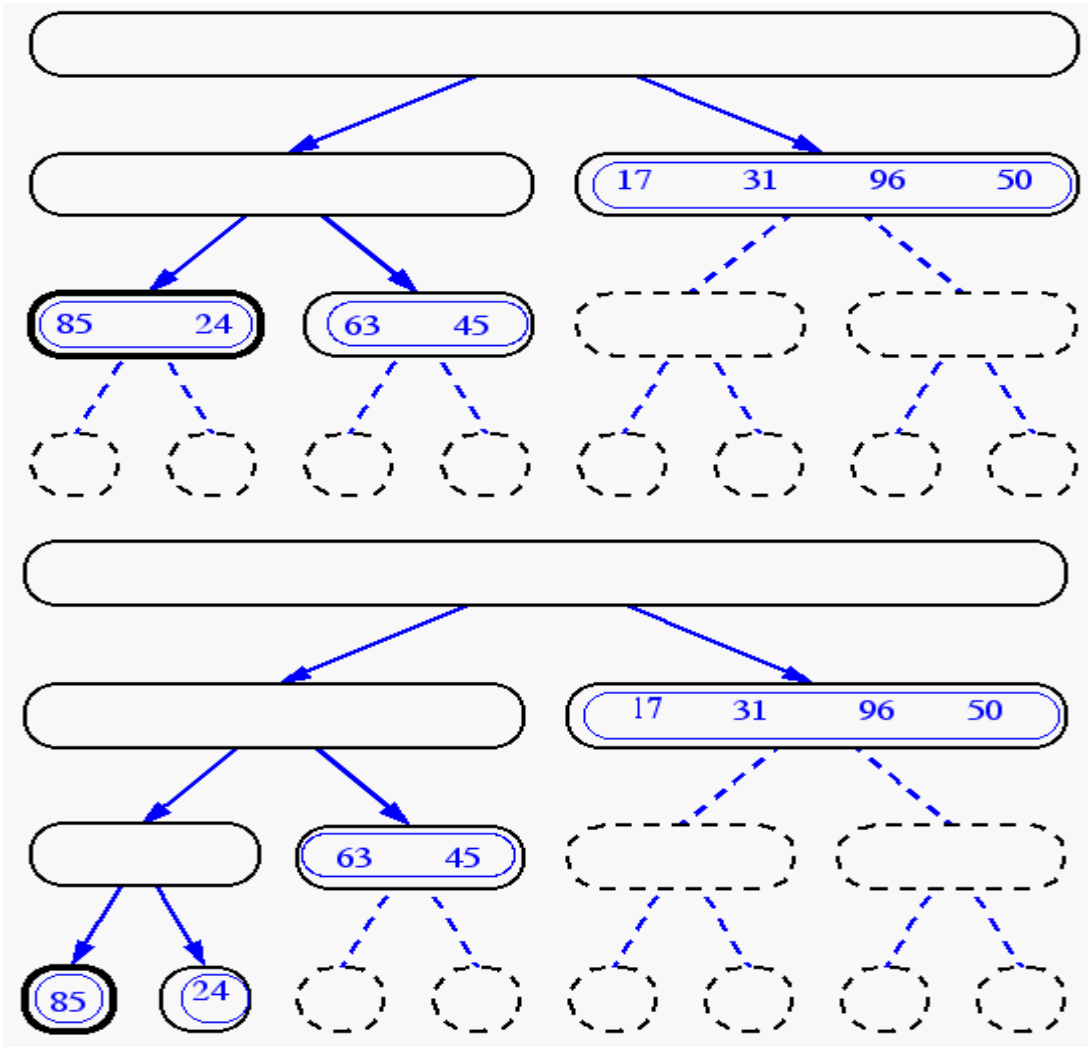


Merge-Sort

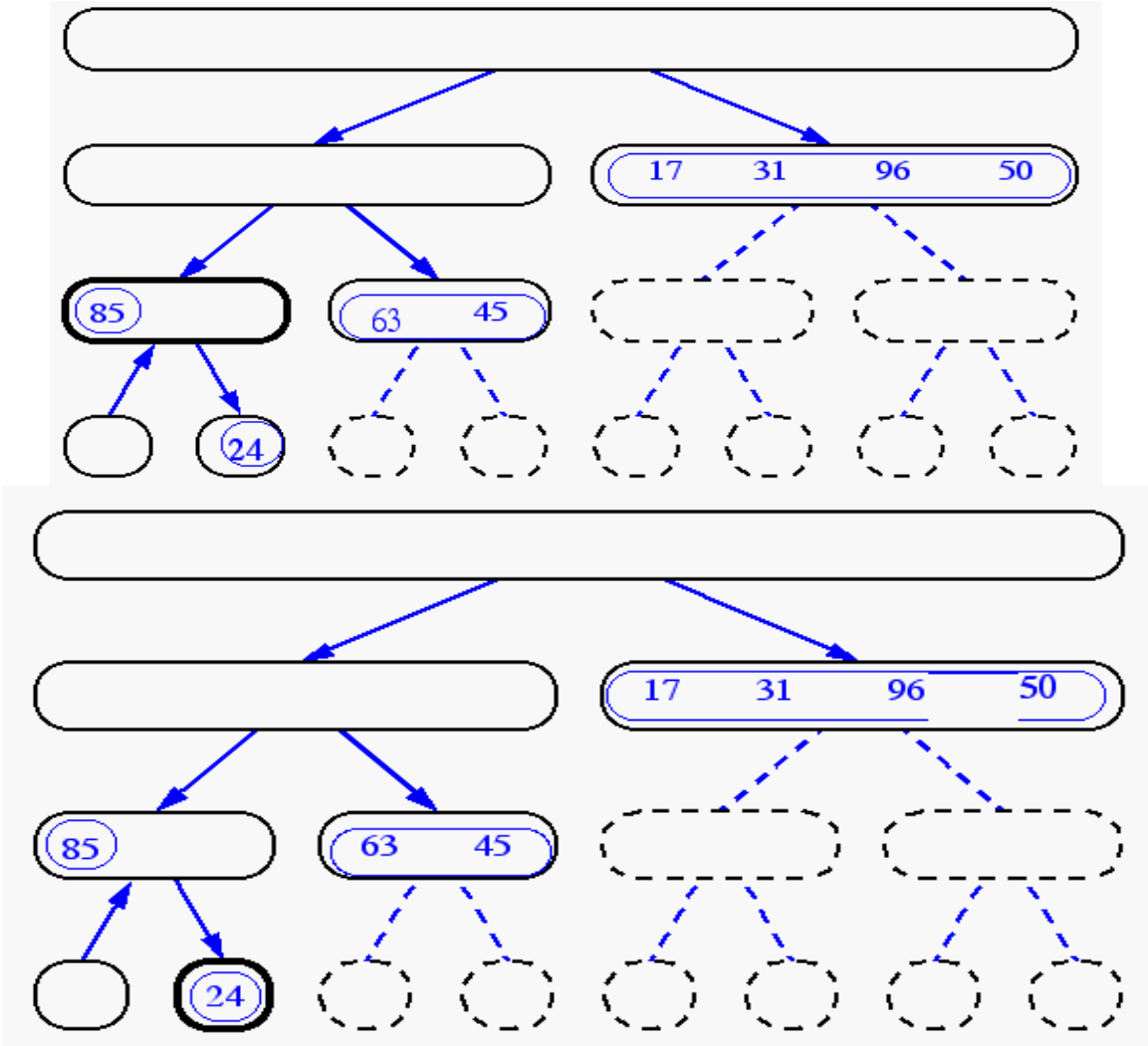




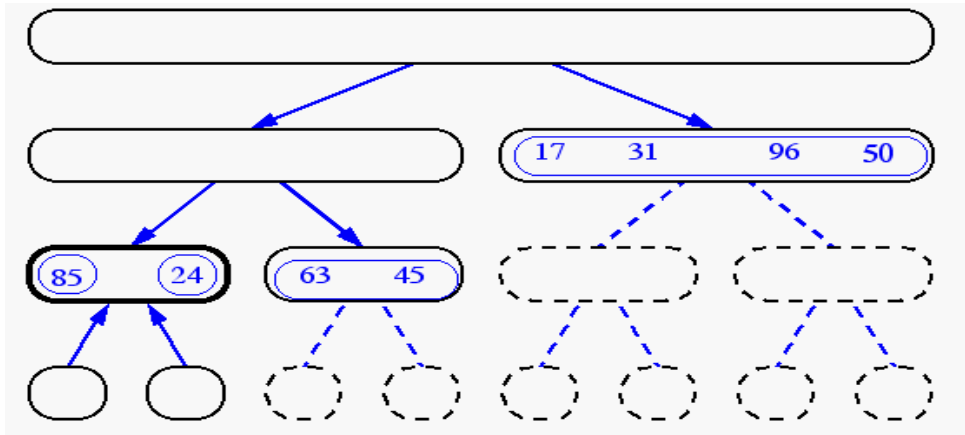
Merge-Sort(cont.)

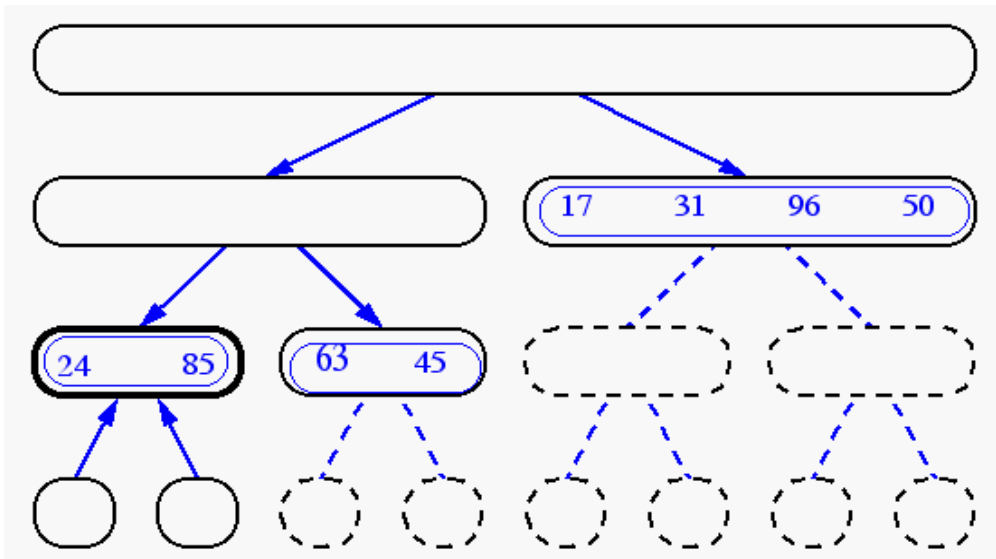


Merge-Sort (cont.)

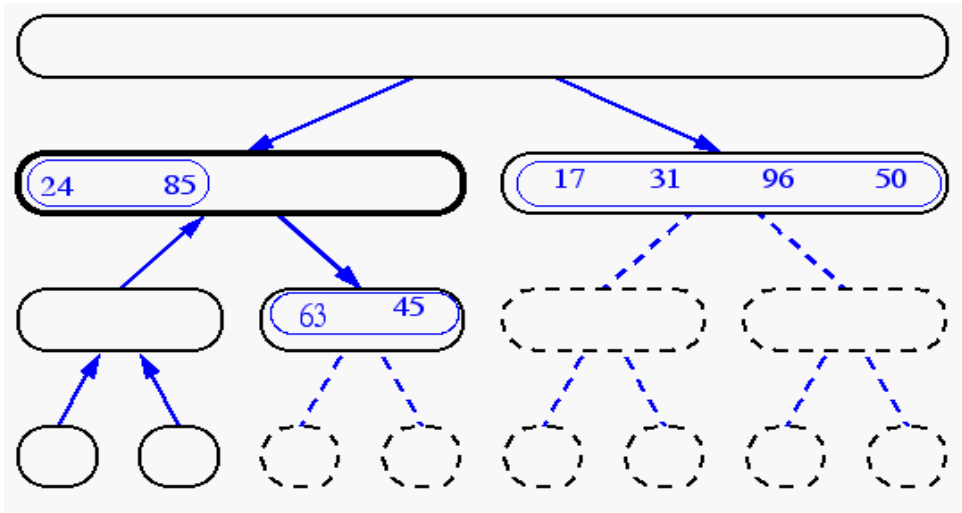


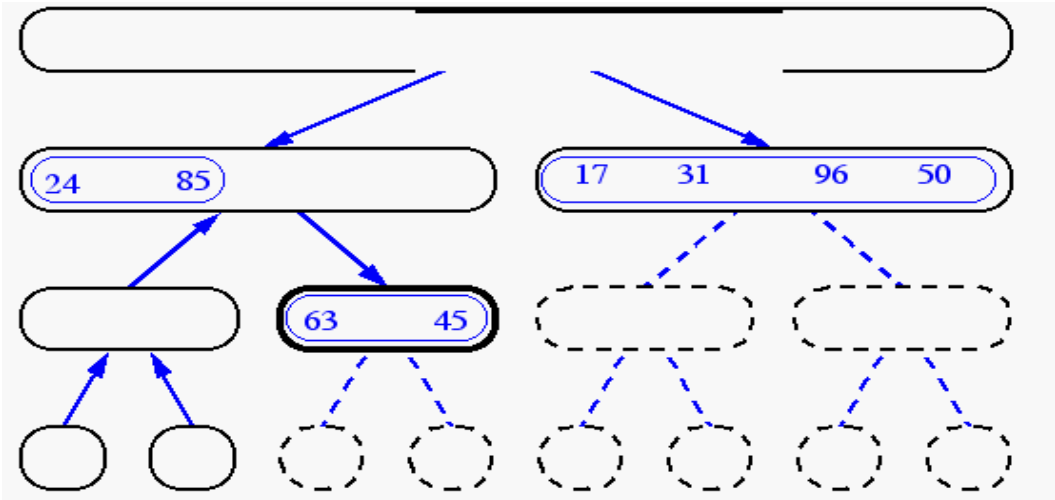
Merge-Sort (cont.)



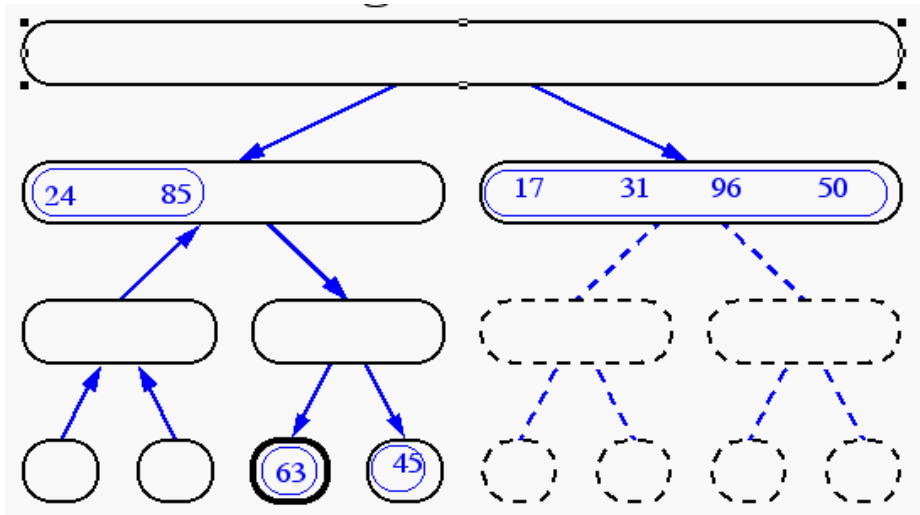


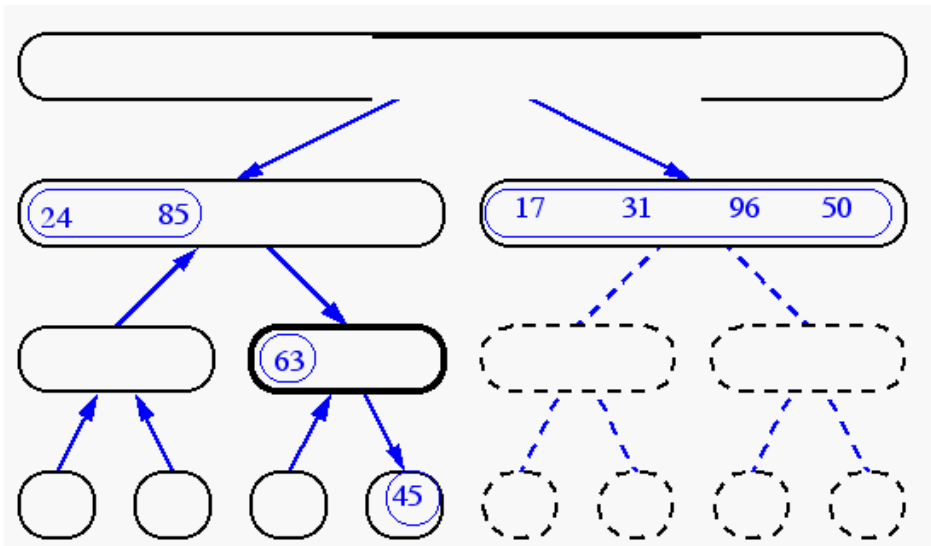
Merge-Sort (cont.)



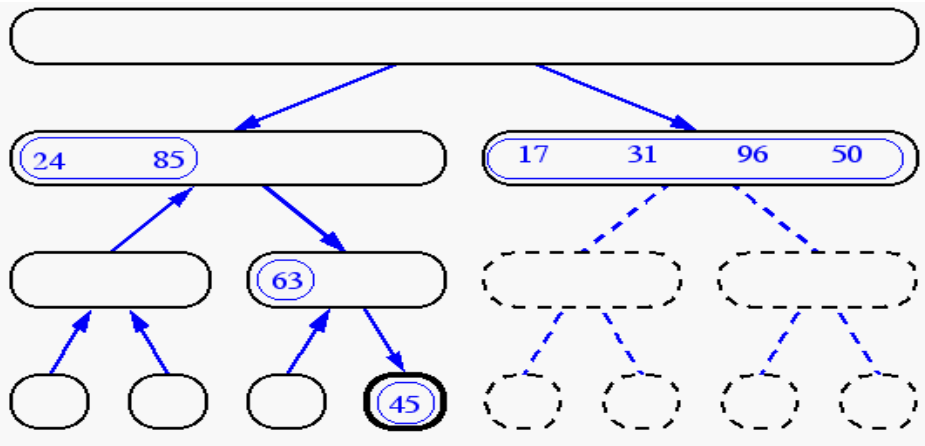


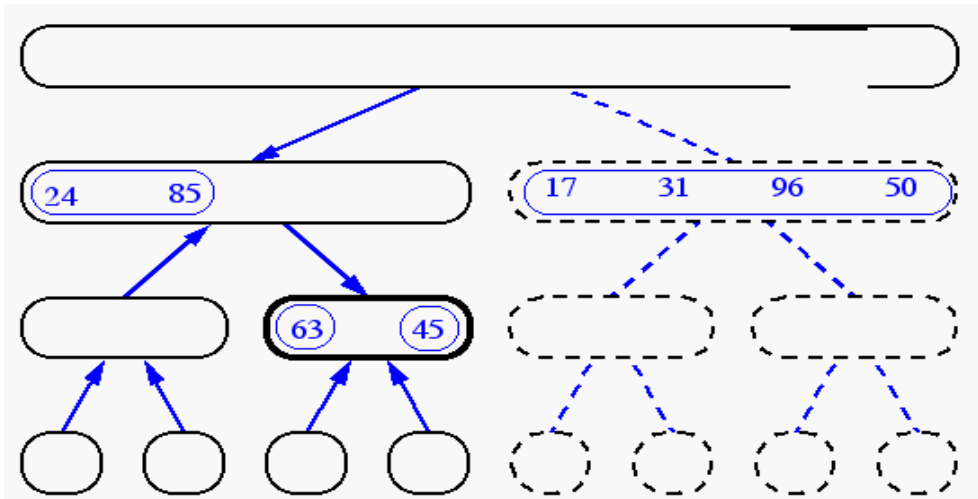
Merge-Sort (cont.)



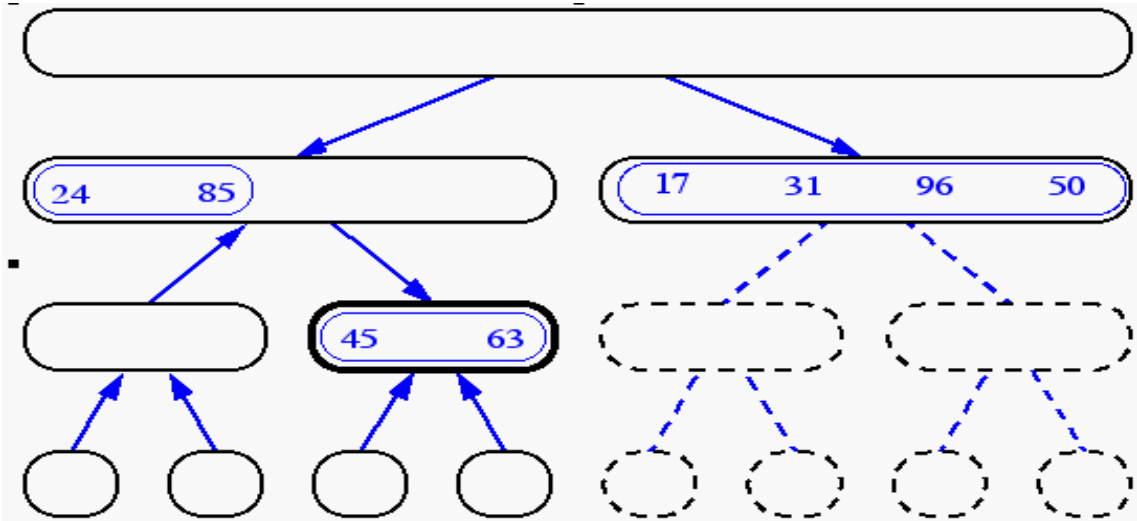


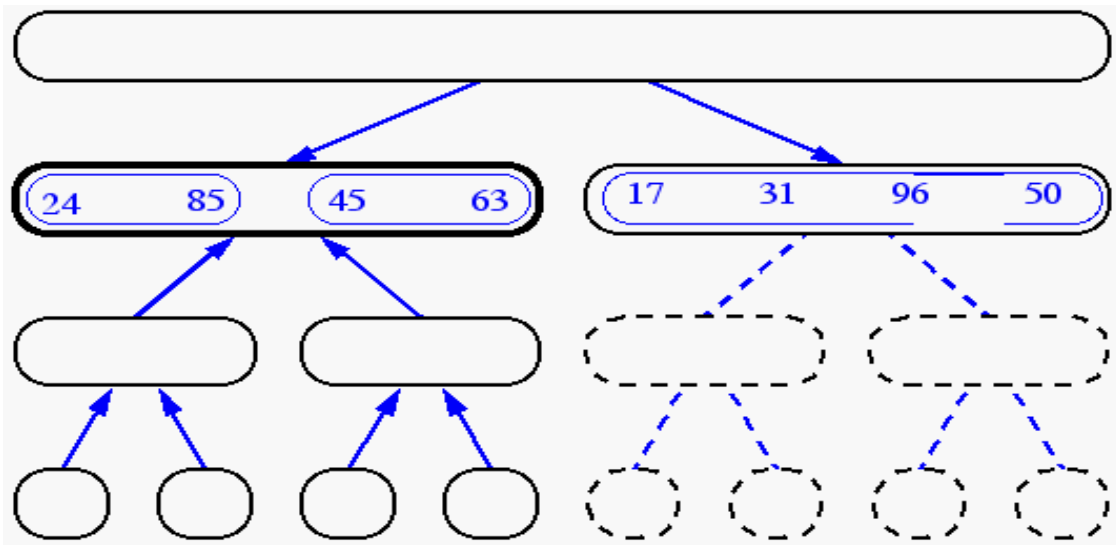
Merge-Sort (cont.)



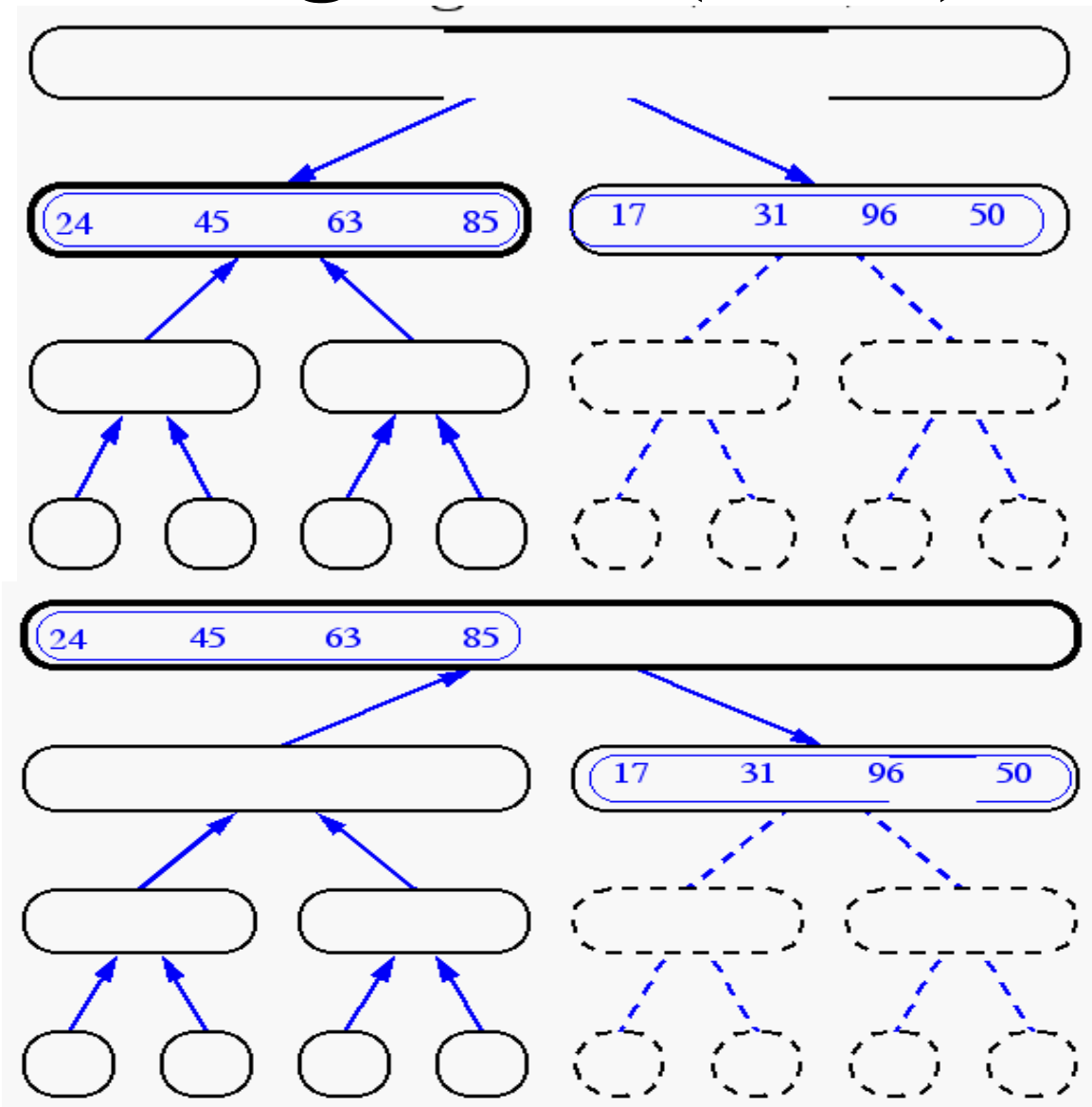


Merge-Sort(cont.)

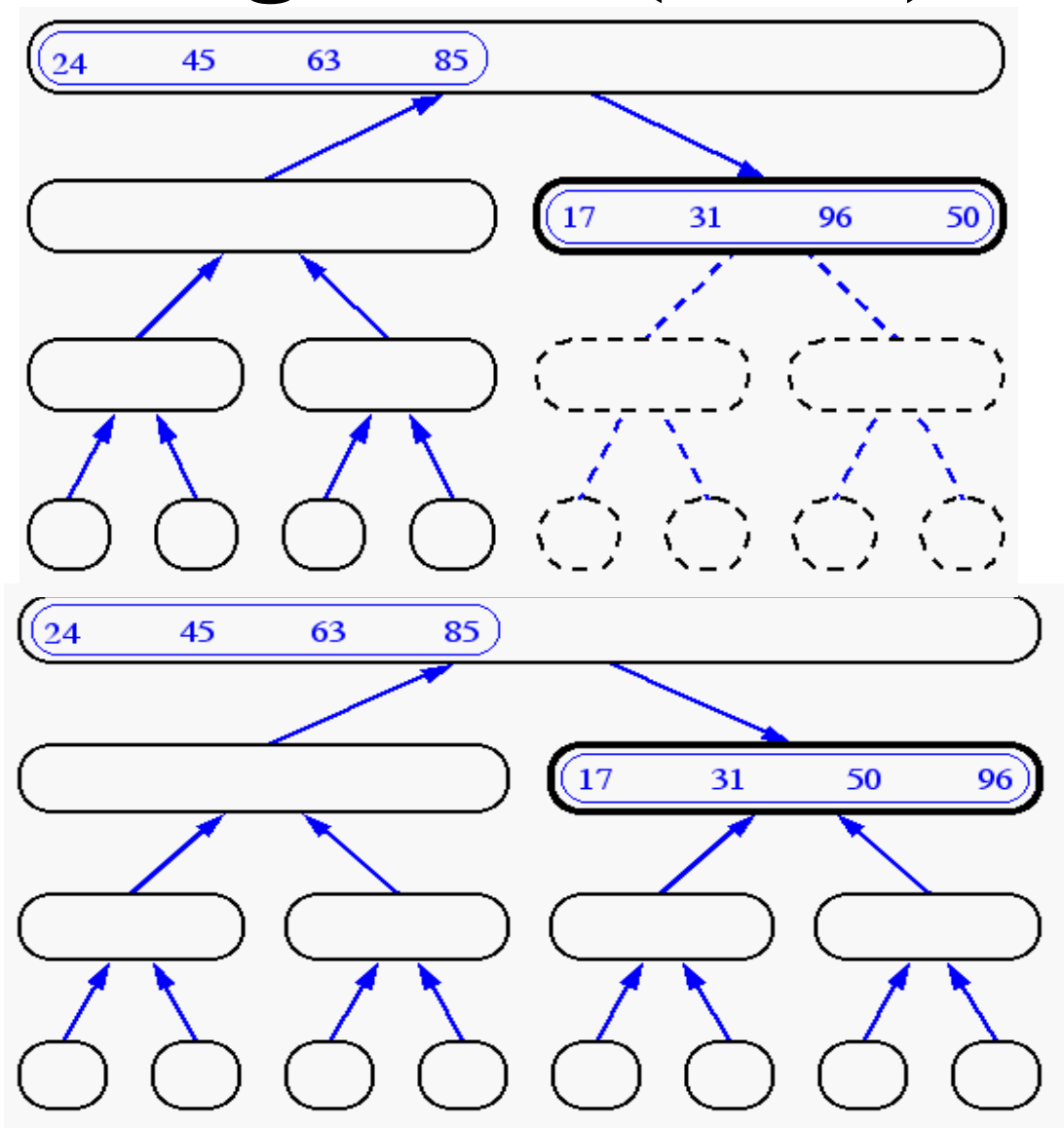




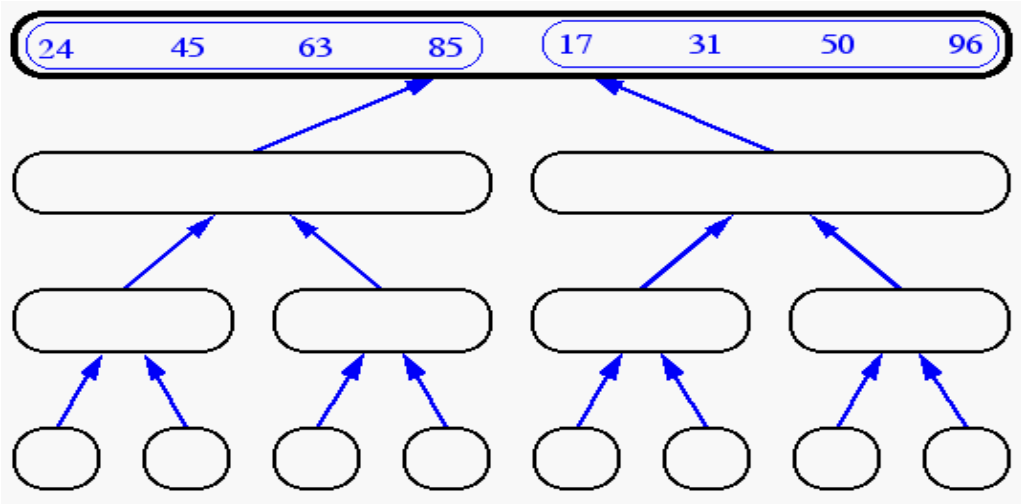
Merge-Sort (cont.)

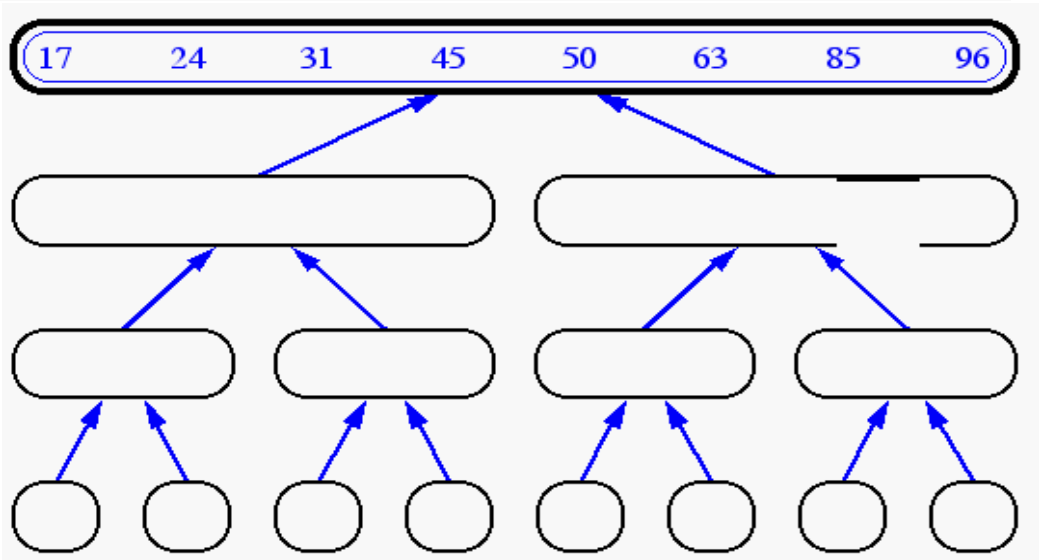


Merge-Sort (cont.)



Merge-Sort (cont.)



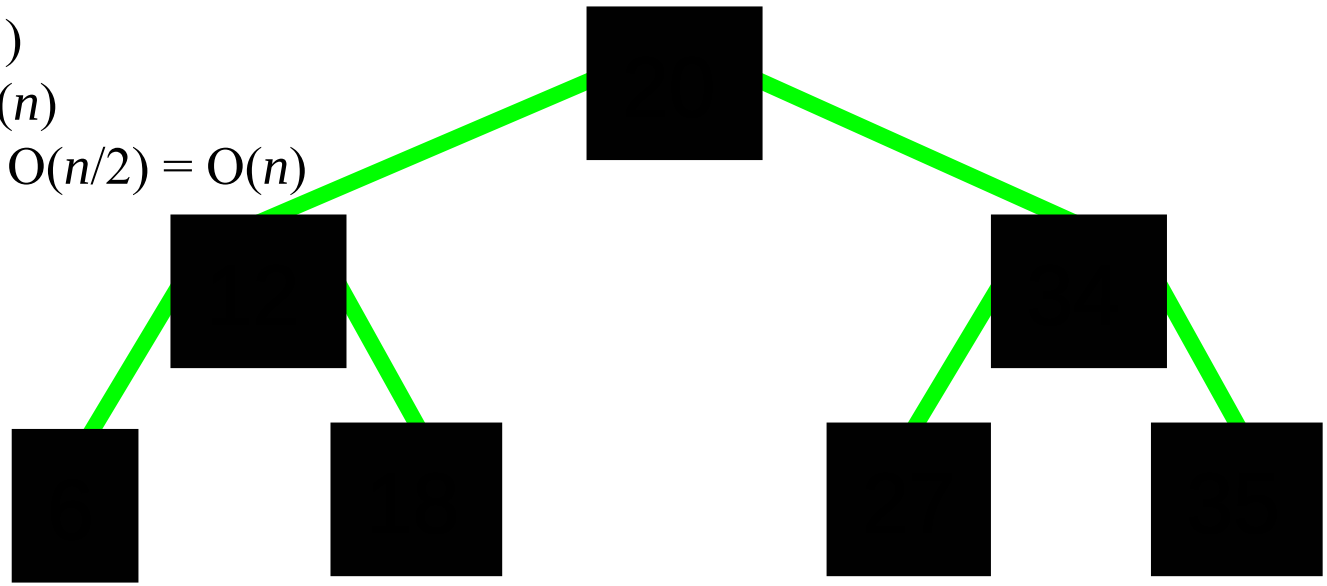


Binary search

- Given a sorted array T and an item x.
- We want to find out the index location of x in T.
- If x does not exist in T, we want the index location of where it would be inserted.
- Find i, such that $T[i\text{-}1] < x < T[i]$

Binary Search

- Search: (6, 12, 18, 20, 27, 34, 35)
- Brute Force:
  - Best Case: O(1)
  - Worst Case: O(n)
  - Average Case: O(n/2) = O(n)



- Binary Search Tree:
  - Best Case: O(1)
  - Worst Case: O(log n)
  - Average Case: O(~log n - 1) = O(log n)

What are some examples similar to Binary Search?

- How do you look up a name in the phone book?
- How do you look up a word in the dictionary?
- How can we make it more efficient?
  - Look up something based on the first letter
  - What is this analogous to in the divide and conquer scheme?

Analyzing divide and conquer algorithms

- Total time of an algorithm based on divide and conquer approach is based on three timings:
- Let T(n) be the running time on a problem of size n. If the problem size is small straight forward solution takes constant time say $\theta(1)$.
- Suppose we divide the problem into number of sub problems each of size 1/b the size of original.
- If it takes D(n) time to divide the problem into sub problems and
- C(n) time to combine the solutions into the solution of original problem,we get the relation as:

$T(n) = \theta(1)$ if $n \leq c$

$= a\,T(n/b) + D(n) + C(n)$ otherwise

For merge sort:

$D(n) = \theta(1)$; $C(n) = \theta(n)$

And we divide the problem into two of size half so

$$T(n) = \theta(1) \quad \text{If } n=1$$

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

Recurrence

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.For example:

- $T(n) = \theta(1)$ if $n = 1$

  $= 2\,T(n/2) + \theta(n)$ if $n > 1$

  There are three main methods to solve –that is for obtaining asymptotic θ or O bounds on the solution.

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

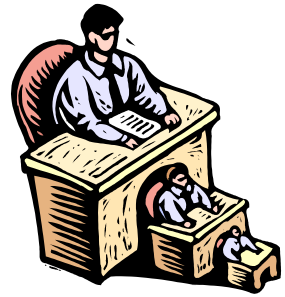
Iterative Substitution

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2 T(n/2^2) + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.
- So, $T(n) = bn + bn \log n$
- Thus, T(n) is O(n log n).

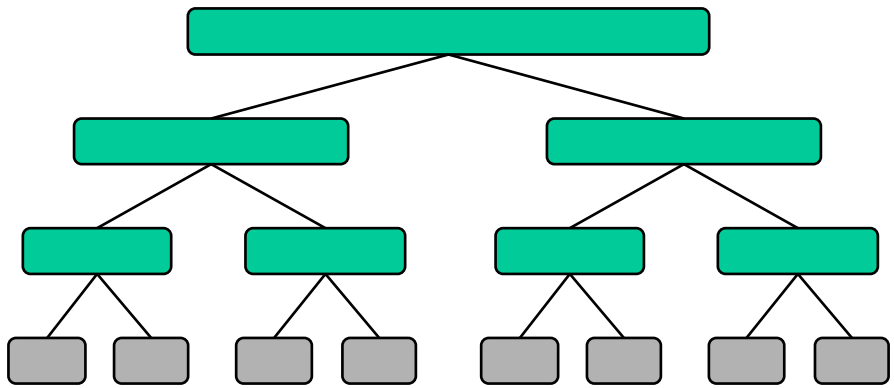
The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

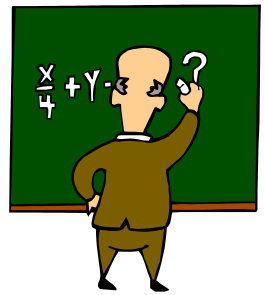
depth	T's	size	time
0	1	***n***	***bn***
1	2	***n***/2	***bn***
i	2^i	$n/2^i$	***bn***
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method



- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

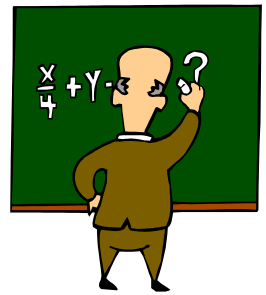
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than cn log n

Guess-and-Test Method



- Recall the recurrence equation:

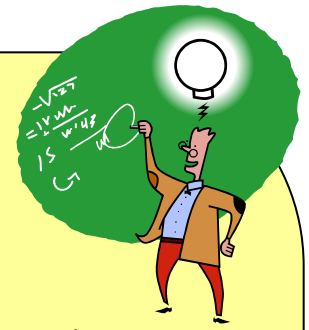
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: $T(n) < cn\log^2 n$.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \end{aligned}$$

  - if $c > b$.
- So, $T(n)$ is $O(n\log^2 n)$.
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method



- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Master Method, Example 1

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

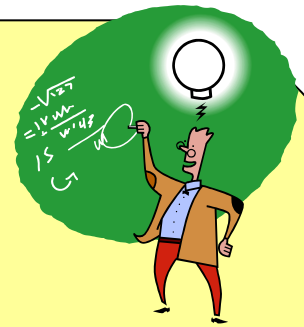
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution: $\log_b a = 2$, since $f(n) = n = O(n^{\log_b a - 1})$

Here $\in = 1$ so case 1 says T(n) is = $\theta(n^2)$.

Master Method, Example 2



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution:

Here a =2 , b= 2 ,$\log_b a = 1$, f(n) = n log n , k = 1

F(n) = n log n = θ (n log n) so case 2 says T(n) is $\theta(n \log^2 n)$.

Master Method, Example 3

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution:

b= 3, a=1, $\log_b a = 0$, $f(n) = n \log n = \Omega(n^{\log a + 1})$

so case 3 says T(n) is $\theta(n \log n)$.

Regularity condition

$a\, f(n/b) <= \delta f(n)$ for some $\delta < 1$

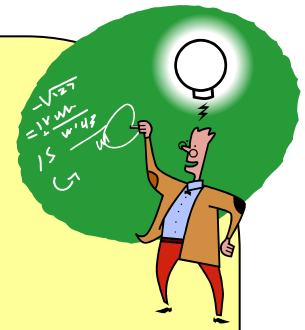
For this case

$a = 1, b = 3$

$1(n/3 \log(n/3)) = (1/3)(n\log(n/3))$

$<= 1/3(n \log n) = \delta f(n)$

Master Method, Example 4



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution: b=2; a = 8 ; $\log_b a = 3$,
$f(n) = n^2 = O(n^{3-1})$; case 1 says T(n) is $\theta(n^3)$.

Master Method, Example 5

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution: $\log_b a = 2$, $f(n) = n^3 = \Omega(n^{2+1})$

$af(n/b) = 9(n/3)^3 = n^3/3 <= (1/3)f(n)$ so case 3 says $T(n)$ is $\theta(n^3)$.

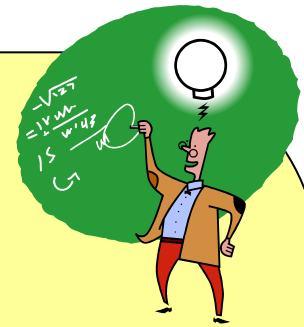
Master Method, Example 6

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/2) + 1 \quad \text{(binary search)}$$

Solution: $\log_b a = 0$, so case 2 says T(n) is O(log n).

Master Method, Example 7



- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a = 1$, so case 1 says T(n) is O(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

Strategy

- 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

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

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

also $f(n) = 1,\ f(n) = \Theta(1)$

$\Rightarrow$ 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)
```

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

$a = 9, b = 3;$

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

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

Examples (2)

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

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

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

$\Rightarrow$ Case 3:

Regularity condition

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

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

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

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

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

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

$\Rightarrow$ neither Case 3 nor Case 2!

Examples (3)

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

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

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

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

Checking the regularity condition

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

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

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

$c = 3/4 < 1$

Examples (3)

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

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

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

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

Checking the regularity condition

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

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

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

$c = 3/4 < 1$

Repeated Substitution Method

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

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

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

- Iteration method(expansion method)
- Substitution method
- Master method

Iteration method

- Idea is to expand(iterate) the recurrence and express it as a summation of terms depending only on n and the initial conditions.
- Ex
- $T(n) = k + T(n/2)$ if $n>1$

  $= c$ if $n=1$

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

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

$T(n) = k + k + T(n/4)$

Repeating this process we get

$T(n) = k + k + k + T(n/8)$

And repeating over and over we get

$T(n) = k + k + \ldots K + T(1)$

$= k + k + \ldots K + c$

How many k's are there? This is no. of times we can divide n by 2 to get down 1 that is log n.Thus

$T(n) = (\log n)*k + c$ where k and c are both constants

Thus

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

Substitution

It involves guessing or informally recognizing the proper answer, then using proof by induction to prove it formally.

Example:

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

Based on some mysterious prompting from our subconscious, we might hypothesize that

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

base T(1) = c constant(log 1)+c’

Inductive assumption: Assume that

T(I) <= cons*(log I) +c for all I <n

Now consider T(n) = k + T(n/2)

Since n/2 is less than n we can use assumption

T(n) <= k + cons *(log n/2) + c

i.e.

$T(n) <= k + cons * ((\log n) - 1) + c$

$< = k - cons + cons * (\log n) + c$

Now what we need to show is that $T(n) <= cons(\log n) + c'$

If we choose cons = k

$T(n) <= cons * \log(n) + c$

Which is exactly what we wanted

Example

- $T(n) = 4\ T(n/2) + n^3$
- $A = 4$, $b = 2$ and $f(n) = n^3$
- Is $n^3 = \Omega(n^{2+\epsilon})$?
- Yes, for $0 < \epsilon < 1$, so case 3 might apply.
- Is $4(n/2)^3 <= c.n^3$?
- Yes , for $c >= ½$ so there exists a $c < 1$ to satisfy the condition for regularity, so case 3 applies and $T(n) = \Theta(n^3)$

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

$$n^{\log_b a} = n^{\log_3 1} = n^0 = 1$$

$$F(n) = \Omega(n^{0+\in}) \ for \in = 1$$

$$we\ still\ need\ to\ check\ af\ (n/b) = n/3 = 1/3(\ f\ (n)$$

$$T(n) = \theta(\ f\ (n)) = \theta(n)(case\ 3)$$

- Ex

$$T(n) = 9T(n/3) + n^{2.5}$$

$$a = 9,\ b = 3,\ and\ \ f(n) = n^{2.5}$$

$$so\ \ n^{\log_b a} = n^{\log_3 9} = 2$$

$$f(n) = \Omega(n^{2+\epsilon})\ with\ \ \epsilon = 1/2$$

$$case3\ \ 3\ if\ \ af(n/b) < cf(n)\ \ for\ some\ c < 1$$

$$f(n/b)\ 9(n/3)^{2.5} = (1/3)^{0.5}\ f(n)$$

$$u \sin g\ c = (1/3)^{0.5}\ case\ 3\ applies\ and$$

$$T(n) = \Theta(n^{2.5)}$$

Suppose $T(n) = aT(n/b) + cn^k$ for n>1 and n a power of b

$T(1) = d$

Where b >= 2 and k>=0 are integers a>0, c>0 d>= 0

Then

$$T(n) = \Theta(n^k) \text{ if } a < b^k$$

$$= \Theta(n^k \lg n) \text{ if } a = b^k$$

$$= \Theta(n^{\log_b a}) \text{ if } a > b^k$$

A tricky example

- T(n) = 2*T(sqrt(n)) + log n
- This can not be solved with the master theorem since sqrt(n) does not match the n/b pattern.
- One way to simplify this problem is to replace n by another variable which is defined as a function of n
- For example suppose let
- M = log n
- Or
- N = 2**m

- Thus
- T(2**m) = 2*T(sqrt(2**m)) + m
  - » = 2*T(2**(m/2)) +m
  - » Now simplifying further by defining S(x) == T(2**x)
- S(m) = 2*S(m/2) +m
- So S(m) = O(m*log m)
- T(2**m) = O(m* log m)
- Or finally
- T(n) =O((log n)*log(log n))

Tower of Hanoi

$= 2M(n-1)+1$

$=2M'(m/b)+1$

Now we can use master theorem

As

$a = 2$; $f(n) = n^0$

So

$M'(m) = \theta(m^{(\log_b 2)}) = \theta(2 ** \log_b m) =$

$\theta(2**n)$

Divide-and-Conquer

- Matrix multiplication and Strassen's algorithm

$\sum_{k \le n} A(i,k)B(k,j)$

.Strassen's matrix multiplication

- Let A and B be two n by n matrices.
- The product C = AB can be obtained as
- C(i,j) = $\sum_{1 \le k \le n} A(i,k)B(k,j)$

For all i and j between 1 and n.

To compute C(i,j) we need n multiplications.As C has n*n elements we can say matrix multiplication algorithm is $\theta(n^3)$

Tower of Hanoi

{// replace all the n plates from peg1 to peg3 using peg2
Hanoitower(peg1, peg3,peg2,n-1)
//replace first n-1 plate from peg1 to peg2 using peg 3
Hanoitower(peg1, peg2, peg3, 1)
Hanoitower(peg2, peg1, peg3, n-1)
}

- $M(n) = 2M(n-1) + 1$

We can not use master theorem directly.

A trick called reparameterization can be used

As: $m = b^n$ for some fixed base b and exponential parameter m.

$M(n) = M(\log_b m) = M'(m)$ so

$M'(m) = M(\log_b m)$

$= M(n)$

Matrix Multiplication

- How many operations are needed to multiply two 2 by 2 matrices?

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Traditional Approach

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

- $r = ae + bg$
- $s = af + bh$
- $t = ce + dg$
- $u = cf + dh$
- 8 multiplications and 4 additions

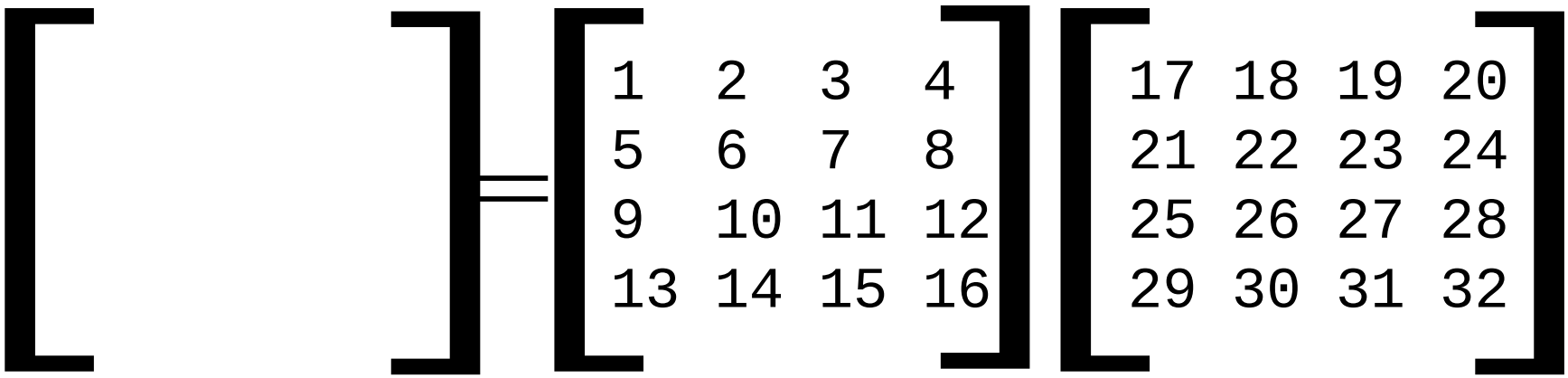
Extending to n by n matrices

$$\begin{bmatrix} R & S \\ T & U \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

- Each letter represents an n/2 by n/2 matrix
- We can use the breakdown to form a divide and conquer algorithm

- $R = AE + BG$
- $S = AF + BH$
- $T = CE + DG$
- $U = CF + DH$
- 8 multiplications of n/2 by n/2 matrices
- $T(n) = 8\ T(n/2) + \Theta(n^2)$
- $T(n) = \Theta(n^3)$

Example



- R = AE + BG
- S = AF + BH
- T = CE + DG
- U = CF + DH

- What are A, B, …, H?
- What happens at this level of the D&C multiplication?

Strassen's Approach

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

- p1 = a(f – h)
- p2 = (a+b)h
- p3 = (c+d)e
- p4 = d(g-e)
- p5 = (a+d)(e + h)
- p6 = (b-d)(g+h)
- p7 = (a-c)(e+f)

- r = p5 + p4 - p2 + p6
- s = p1 + p2
- t = p3 + p4
- u = p5 + p1 – p3 – p7

- 7 multiplications
- 18 additions

Extending to n by n matrices

$$\begin{bmatrix} R & S \\ T & U \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

- Each letter represents an n/2 by n/2 matrix
- We can use the breakdown to form a divide and conquer algorithm

- 7 multiplications of n/2 by n/2 matrices
- 18 additions of n/2 by n/2 matrices
- $T(n) = 7\ T(n/2) + \Theta(n^2)$
- $T(n) = \Theta(n^{\lg 7})$

Example

$$\begin{bmatrix} & \\ & \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \begin{bmatrix} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{bmatrix}$$

- p1 = a(f – h)
- p2 = (a+b)h
- p3 = (c+d)e
- p4 = d(g-e)
- p5 = (a+d)(e + h)
- p6 = (b-d)(g+h)
- p7 = (a-c)(e+f)
- r = p5 + p4 - p2 + p6
- s = p1 + p2
- t = p3 + p4
- u = p5 + p1 – p3 – p7

- What are A, B, …, H?
- What happens at this level of the D&C multiplication?

Observations

- Comparison: n= 70
  - Direct multiplication: $70^3 = 343{,}000$
  - Strassen: $70^{\lg 7}$ is approximately 150,000
  - Crossover point typically around n = 20
- Hopcroft and Kerr have shown 7 multiplications are necessary to multiply 2 by 2 matrices
  - But we can do better with larger matrices

- $T(n) = b$ for $n <= 2$
  - $= 7\,T(n/2) + a\,n^2$ For $n > 2$

Where a and b are constants.

Solving it

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

The problem of multiplying long integers

X,Y two n-bit integers.

Simple multiplication is $O(n^2)$ algorithm counting single bit or digit multiplication as one step.

X: | A | B |

$X = A\,2^{n/2} + B$

Y: | C | D |

$Y = C\,2^{n/2} + D$

$XY = AC\,2^{n} + (AD + BC)\,2^{n/2} + BD$

Four multiplications of n/2 bit integers (AC,AD,BC and BD) three additions with at most 2n bits (corresponding to three + signs) and two shifts (multiplications by 2^{n} and $2^{n/2}$). As these additions and shifts take O(n) steps ,we can write recurrence relation for it as:

$T(1) = 1$

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

Solving it

$T(n)$ is $O(n^{2})$

Which is no improvement over ordinary method.

- We can get asymptotic improvement if we decrease the number of sub problems.

$$XY = AC2^{n} + [(A-B)(D-C) + AC + BD]2^{n/2} + BD$$

It requires only three multiplications of n/2 bit integers AC , BD, and (A-B)(D-C) , six additions or subtractions and two shifts.

All but multiplications take O(n) steps the time T(n) is given as:

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

Solution is $T(n) \in O(n^{\log_2 3}) = O(n^{1.59})$

Worst case

- $T(n) = P(n) + T(0) + T(n-1)$

$= c.n + 0 + T(n-1)$

$= c.n + c.(n-1) + T(n-2)$

$= c.n + c.(n-1) + c.(n-2) + T(n-3)$

.

.

$= c\,(1+2+3+ \ldots + n) + T(0)$

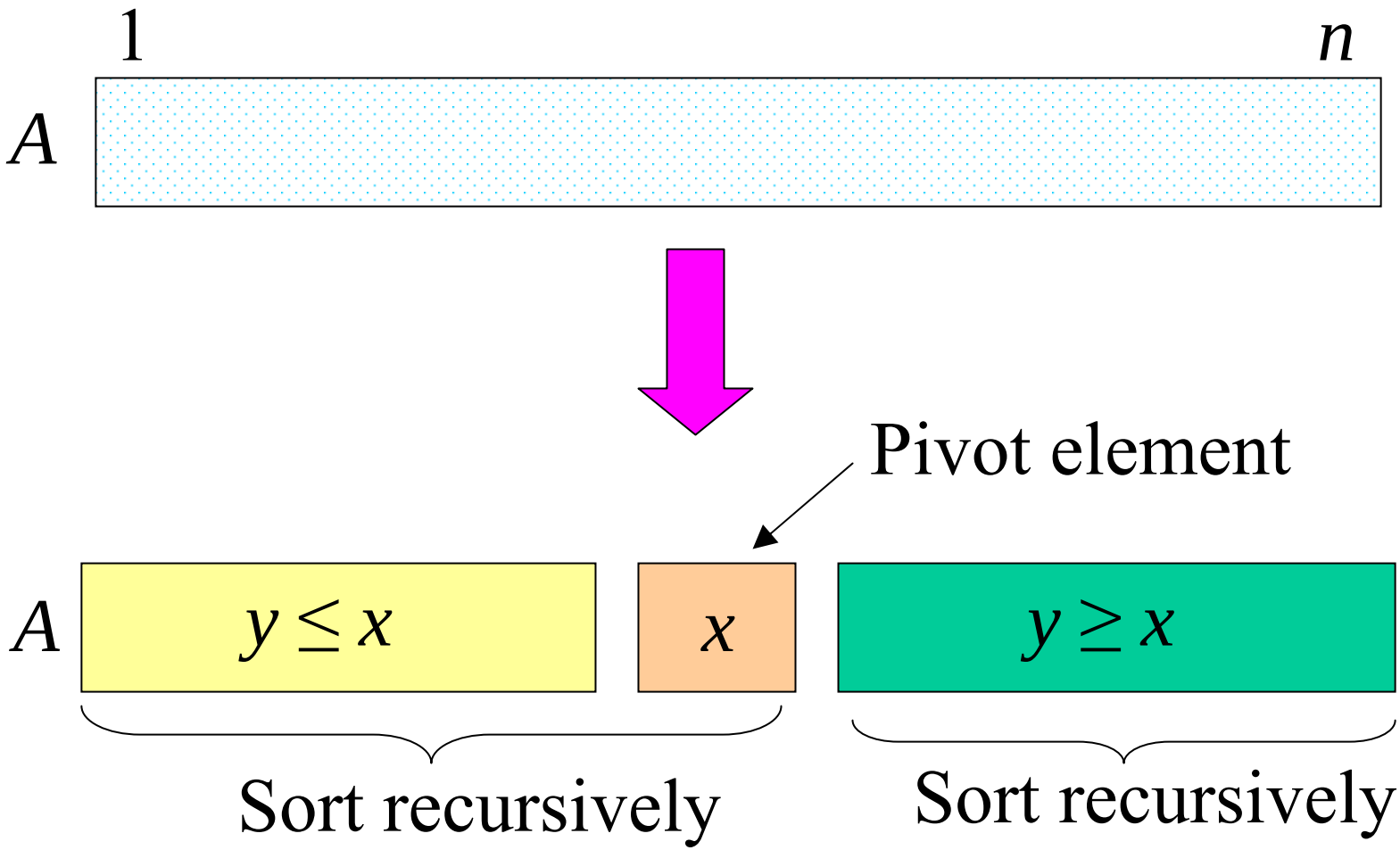
$= c.n.(n+1)/2$

$= O(n^2)$

Quick Sort

- Similar to Merge Sort but the partitioning of the array is not into halves, but by choosing a pivot point, with lesser numbers on left, greater on right.
- Quick sort partitions the array of numbers to be sorted, but does not need to keep a temporary array.
- Quick sort is faster than Merge sort because it does not go through the merge phase.

Quick Sort Approach



Complexity analysis for Quick sort

- The best case occurs when the pivot partitions the set into halves. The complexity is then O(N log N).
- The worst case occurs when the pivot is the smallest element. The complexity is then $O(N^2)$.
- The average case is also O(N log N) because the problem is halved at each step.

Quick Sort Algorithm

Input: Unsorted sub-array $A[p..r]$

Output: Sorted sub-array $A[p..r]$

```
QUICKSORT (A, p, r)
    if p < r
      then q ← PARTITION(A, p, r)
           QUICKSORT (A, p, q)
           QUICKSORT (A, q+1, r)
```

Partition Algorithm

Input: Sub-array $A[p..r]$

Output: Sub-array $A[p..r]$ where each element of $A[p..q]$ is $\leq$ to each element of $A[(q+1)..r]$; returns the index q

PARTITION (A, p, r)

```
1   x ← A[p]
2   i ← p -1
3   j ← r +1
4   while TRUE
5           repeat j ← j - 1
6               until A[j] ≤ x
7           repeat i ← i + 1
8               until A[i] ≥ x
9           if i < j
10              then exchange A[i] ↔ A[j]
11              else return j
```

$\Theta(n)$ where $n = r-p+1$

Picking a pivot

- Choosing the first element in the array for pivot will give us $O(N^2)$ complexity if the array is sorted.
- Choosing the middle point is safe; it will give average complexity, but does not necessarily give the fastest running time.
- The median would give us the best running time but it is time-consuming to compute the median.
- We approximate the median by taking the median of 3 numbers in the array, namely the first, the middle and the last elements.

Quick Sort

- **Each time the function recurses, a "key" value of the structure is selected to be positioned.**
- **Values less than the key are passed to the left side of the structure and values that are greater than the key are passed to the right side of the structure.**
- **These "left to right" and "right to left" scans and swaps continue until a flag condition tells them to stop.**
- **The function then repeatedly scans through the structure from two directions.**

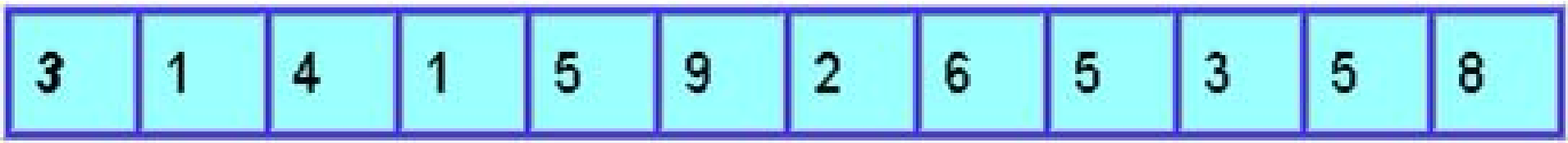
- **This occurs when the structure can be divided into two sorted partitions.**
- **The key is then "switched into its final position".**
- **Next the quick sort function is called to sort the partition that contains the values that are less than the key.**
- **Then the quick sort function is called to sort the partition that contains the values that are greater than the key.**
- **The above algorithm continues until a condition indicates that the structure is sorted.**

Quick Sort

A Step Through Example

1. **This is the initial array that you are starting the sort with**

2. **The array is pivoted about its first element *p = 3***



3. **Find first element larger than pivot (underlined) and the last element not larger than pivot (italicised)**

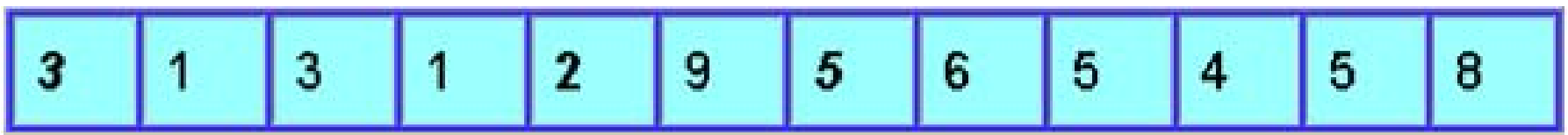
Quick Sort

A Step Through Example

4. **Swap those elements**

5. **Scan again in both directions**

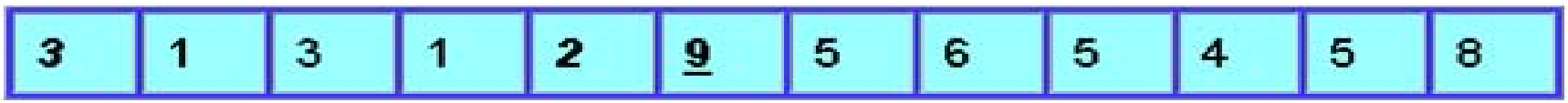
6. **Swap**



Quick Sort

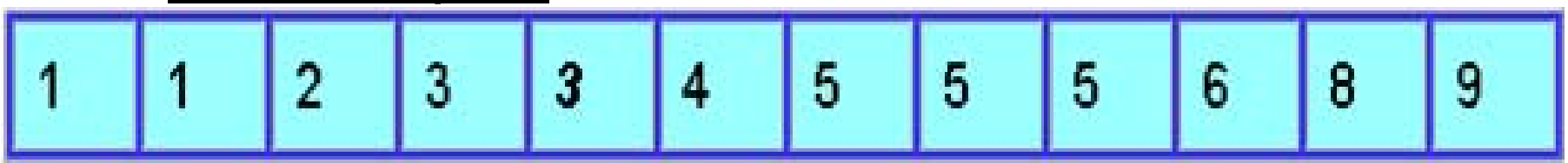
A Step Through Example

7. **Scan**



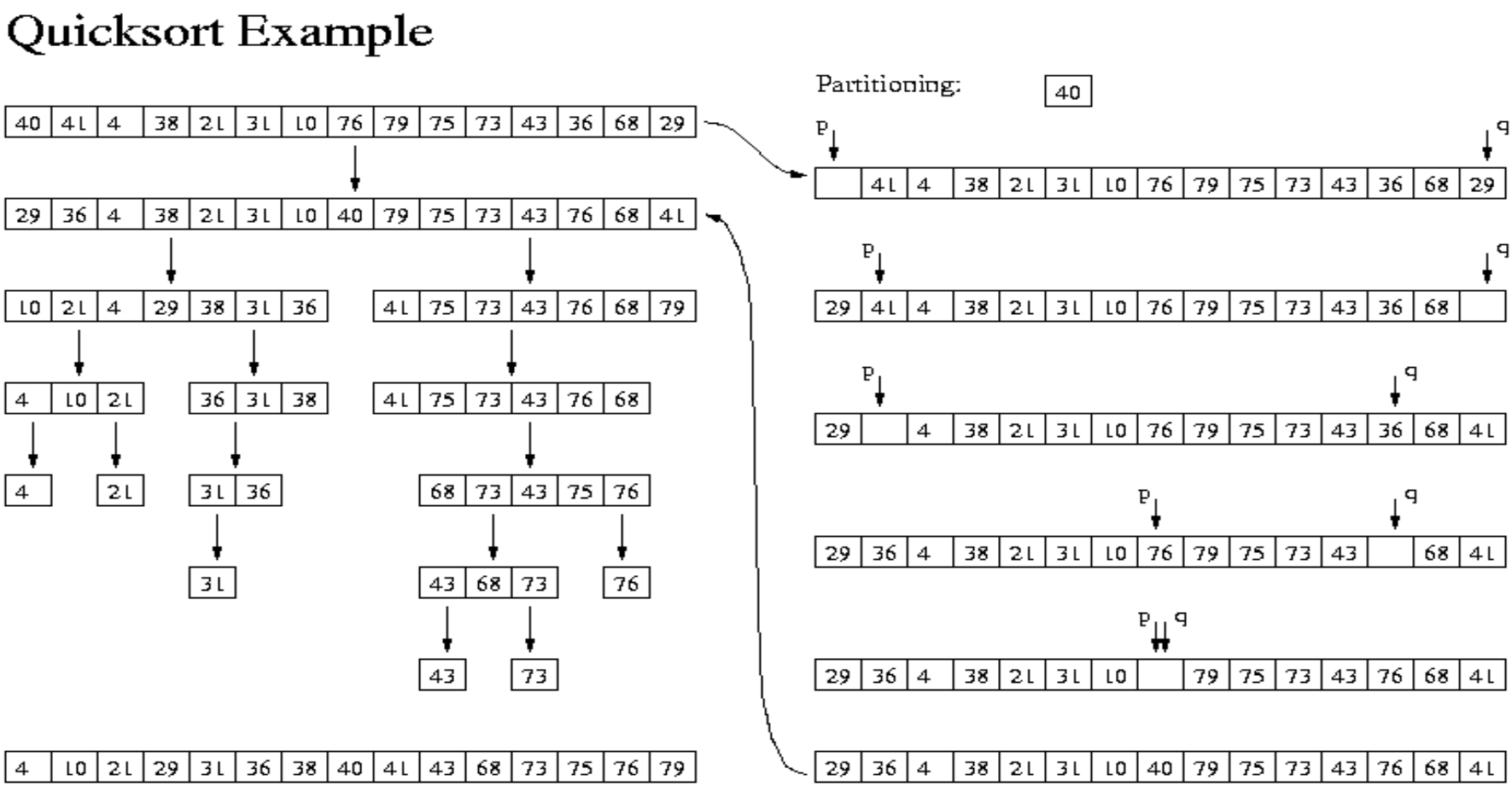
8. **The pointers have crossed: swap pivot with italicised.**

9. **Pivoting is now complete. Recursively sort subarrays on each side of pivot.**



The array is now sorted.

Quick Sort


Quicksort Example
Partitioning:
40
p
q
In this example, the left-most array element is used for the pivot during partitioning.

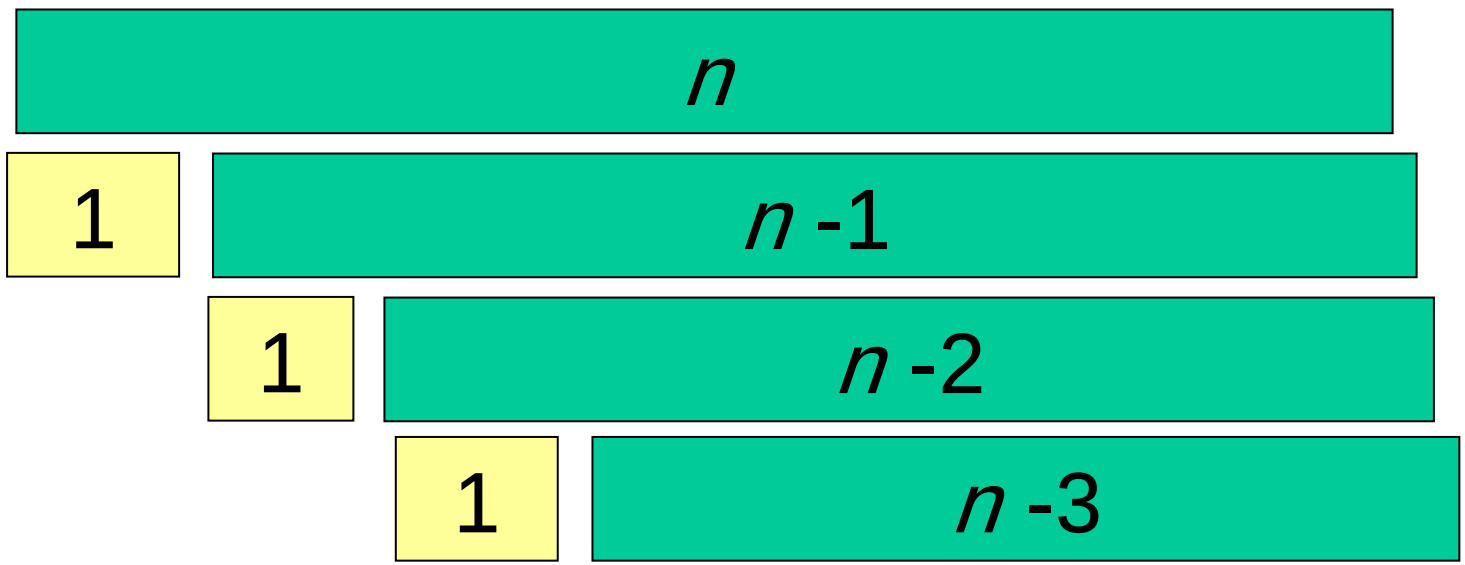
Treatment of elements equal to pivot

• If we ignore elements equal to pivot, then i & j do not stop. This creates wildly uneven subsets in the recursion, & can lead to a complexity of $O(N^2)$.

• If i & j both stop at elements equal to the pivot, & if the array is made up of all duplicates, then everyone swaps. In this case all subsets in the recursion are even & the complexity is O(N log N).

Analysis of Quick Sort

- *Worst-case*: if unbalanced partitioning
  - One region with 1 element and the other with n-1 elements
  - If this happens in every step $\Rightarrow \Theta(n^2)$
  - $T(n) = T(n\text{-}1) + T(1) + \Theta(n) \Rightarrow \Theta(n^2)$



- Worst-case occurs
  - When array is already sorted (increasingly)

- *Best-case*: if balanced partitioning
  - Two regions, each with $n/2$ elements
  - $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \lg n)$

- *Average case closer to the best case than the worst case*