COMP20010


# Data Structures and Algorithms I
# 08 - Recursion II

*Dr. Aonghus Lawlor*
*aonghus.lawlor@ucd.ie*

# Recursion

# Divide & Conquer

# Recursion

- We are going to look at some sorting algorithms
- Interesting to consider the complexity of these algorithms
- We will consider iterative and recursive solutions
- Many of the recursive solutions fit into the category of divide & conquer

# Selection Sort

- works by repeatedly sorting elements
- first find the smallest in the array and exchange it with the element in the first position
- then find the second smallest element and exchange it with the element in the second position
- continue in this way until the entire array is sorted.

# Selection Sort

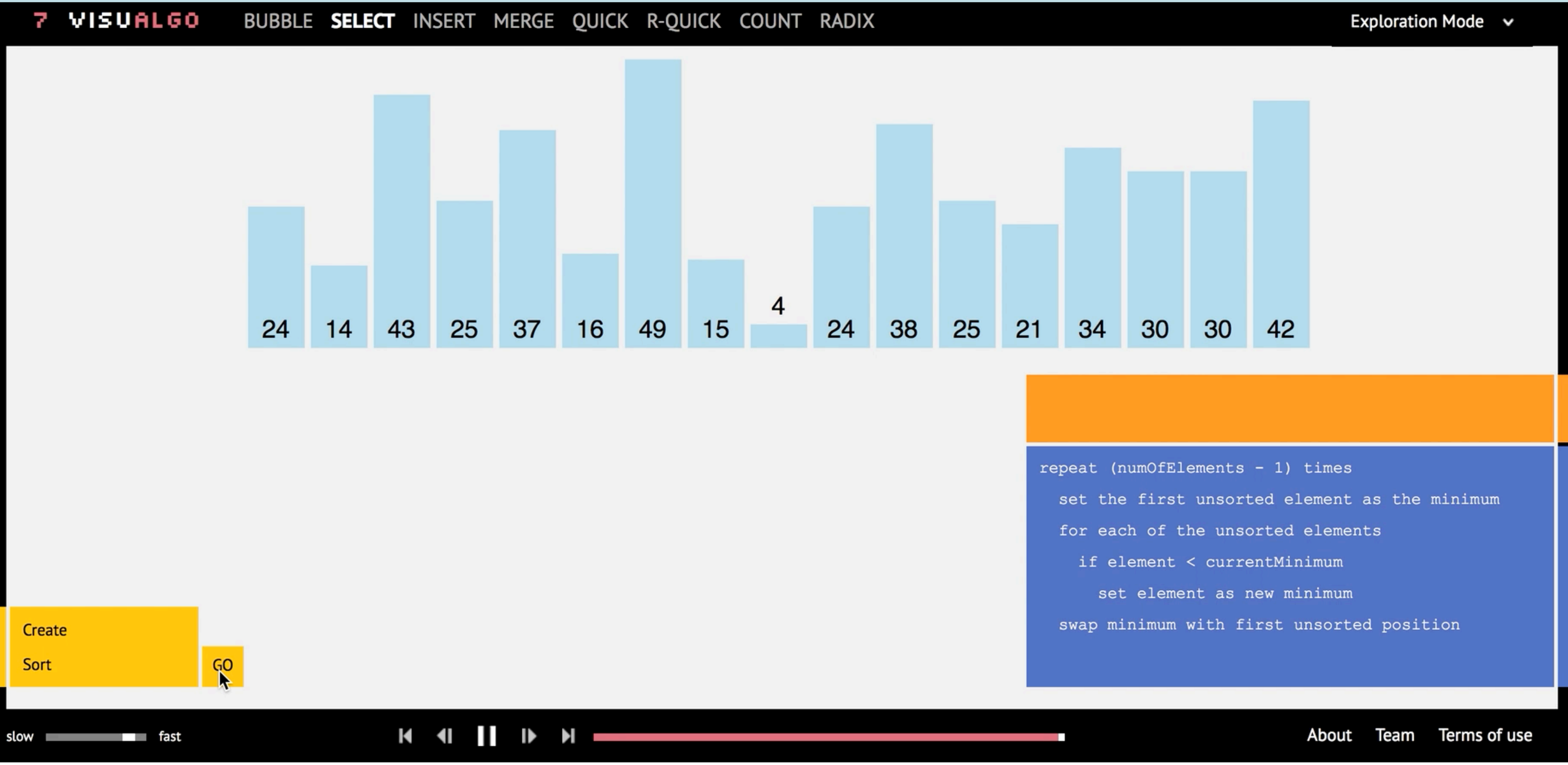
```
Algorithm: Selection-Sort (A)
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ←i + 1 to n do
        if A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A [i]
    A[i] ← min x
```

Selection sort is possibly the simplest of sorting techniques and works very well for small files.

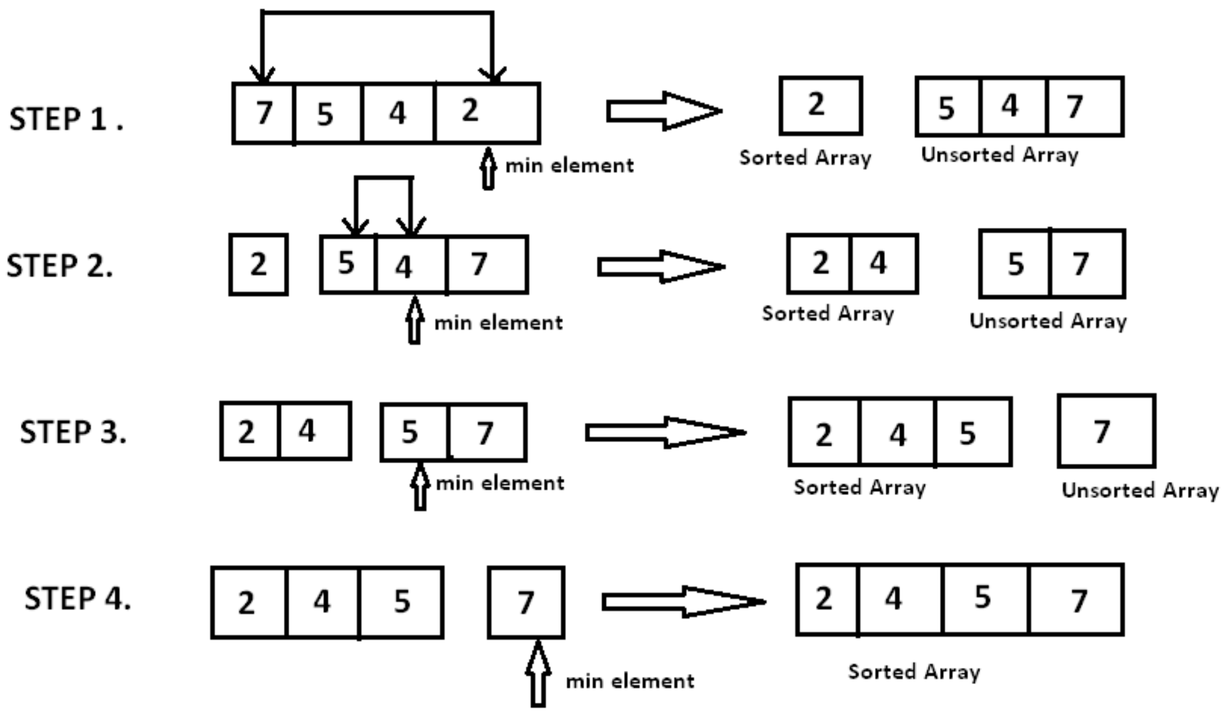
It has a quite important application as each item is actually moved at the most once.

Section sort is a good method for sorting files with very large objects (records) and small keys.

The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.


7 VISUALGO
BUBBLE SELECT INSERT MERGE QUICK R-QUICK COUNT RADIX
Exploration Mode
24 14 43 25 37 16 49 15 4 24 38 25 21 34 30 30 42
repeat (numOfElements - 1) times
set the first unsorted element as the minimum
for each of the unsorted elements
if element < currentMinimum
set element as new minimum
swap minimum with first unsorted position
Create
Sort
GO
slow
fast
About
Team
Terms of use

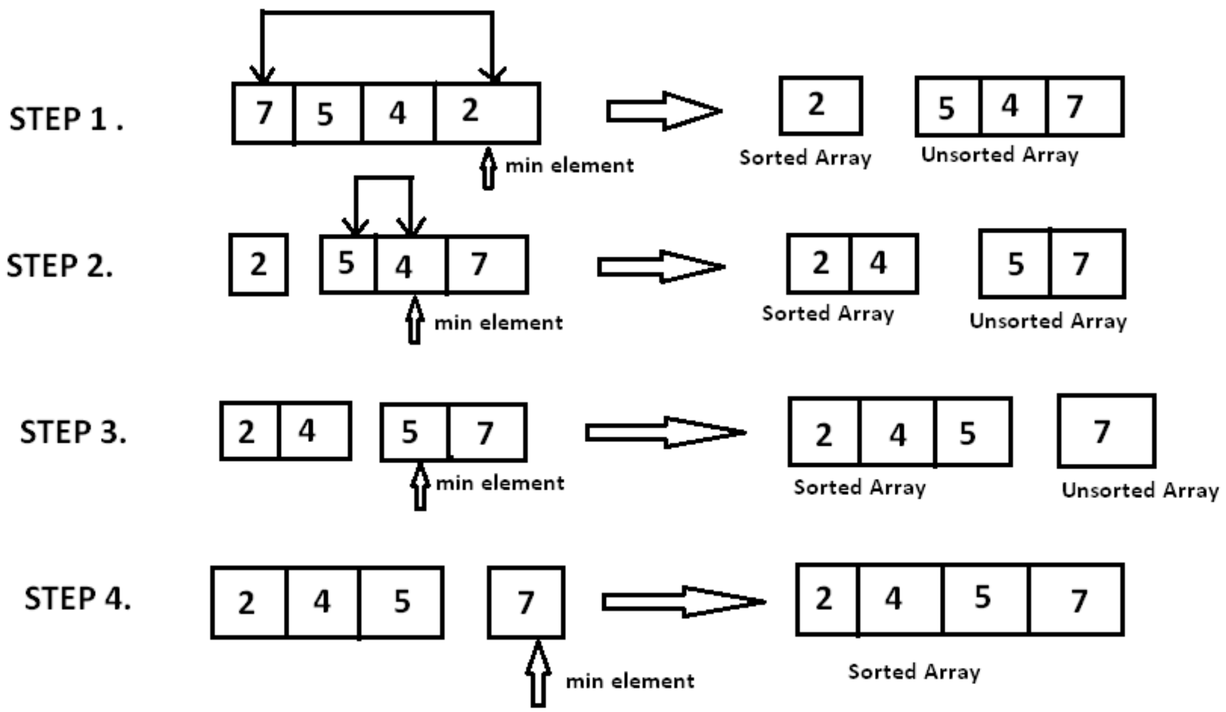
# Selection Sort




**Algorithm: Selection-Sort (A)**

```
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ←i + 1 to n do
        if A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A [i]
    A[i] ← min x
```

# Selection Sort




To find the minimum element from the array of N elements, N-1 comparisons are required.

After putting the minimum element in its proper position, the size of an unsorted array reduces to N-1 and then N-2 comparisons are required to find the minimum in the unsorted array.

$$(N-1)+(N-2)+(N-3)+\ldots = N(N-1)/2 + N = O(N^2)$$

comparisons swaps time complexity

# Bubble Sort

The bubble-sort algorithm has the following properties:

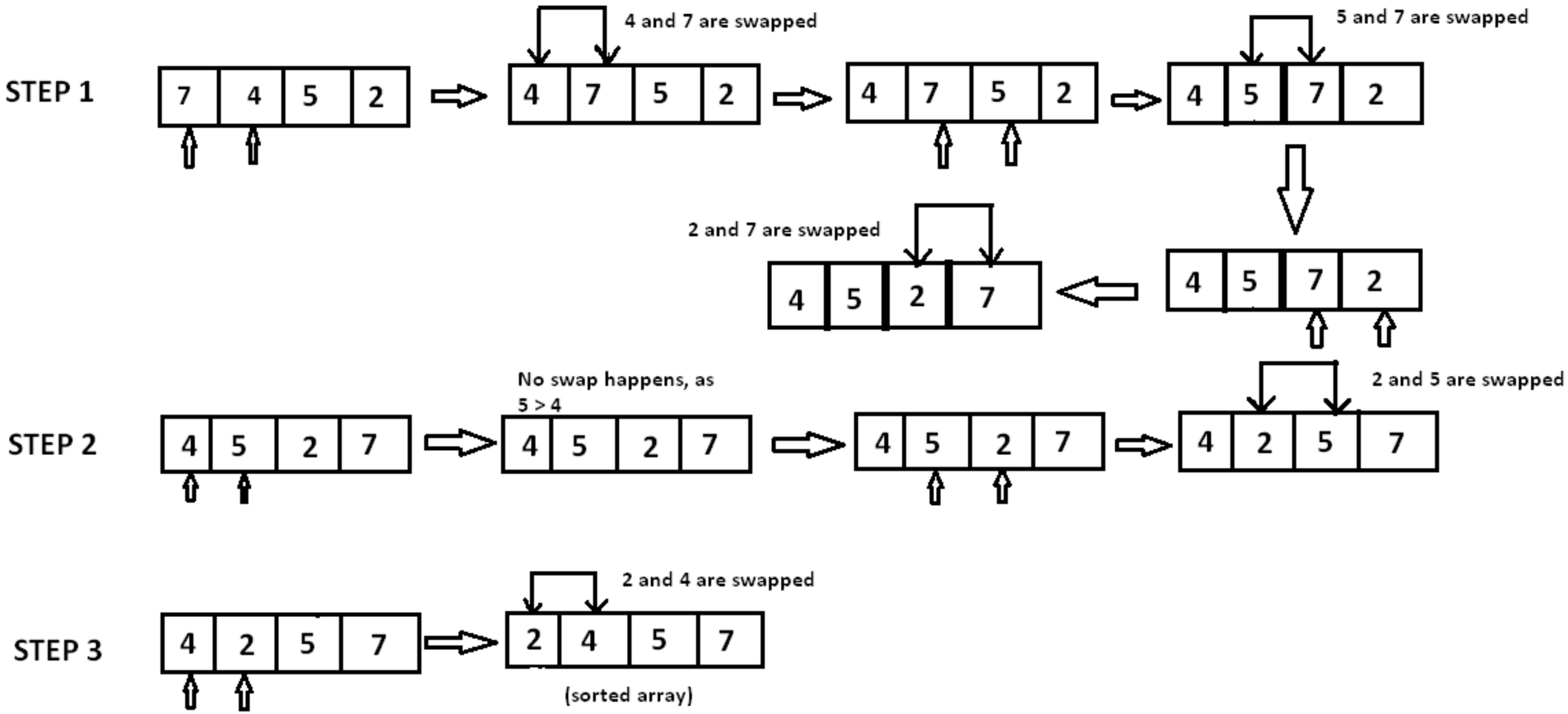
- In the first pass, once the largest element is reached, it keeps on being swapped until it gets to the last position of the sequence.
- In the second pass, once the second largest element is reached, it keeps on being swapped until it gets to the second-to-last position of the sequence.
- In general, at the end of the *i*th pass, the right-most *i* elements of the sequence (that is, those at indices from $n - 1$ down to $n - i$) are in final position.

# Bubble Sort

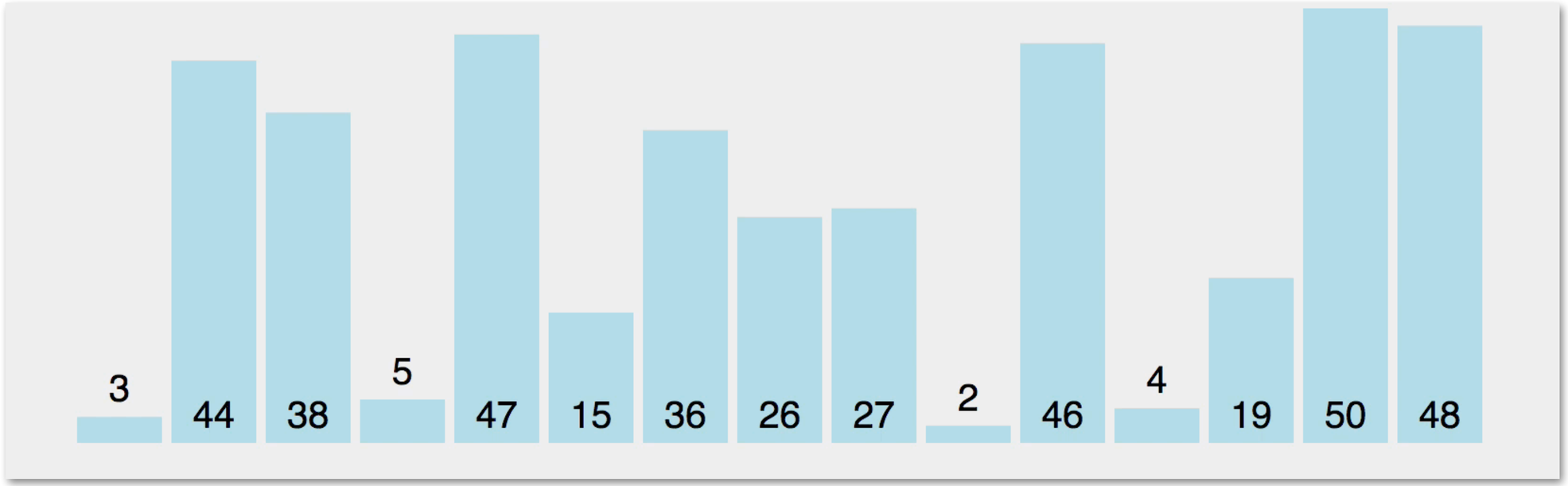
```
1:  function BubbleSort(A)
2:      Input array A of length n
3:      for i = 1 to n do
4:          for j = 0 to n − 1 do
5:              if A[j] > A[j + 1] then
6:                  swap(A[j], A[j + 1])
7:              end if
8:          end for
9:      end for
10: end function
```

# Bubble Sort

```
A [ ] = { 7, 4, 5, 2}
```



# Bubble Sort




```
1:  function BubbleSort(A)
2:      Input array A of length n
3:      for i = 1 to n do
4:          for j = 0 to n − 1 do
5:              if A[j] > A[j + 1] then
6:                  swap(A[j], A[j + 1])
7:              end if
8:          end for
9:      end for
10: end function
```

# Bubble Sort Analysis

- Assume that the implementation of the sequence is such that the accesses to elements and the swaps of elements performed by bubble-sort take *O*(1) time each. That is, the running time of the *i*th pass is O(n-i+1), then the overall running time is:

$$O\left(\sum_{i=1}^{n}(n-i+1)\right)$$

- and we can sum this to:

$$n + (n-1) + \ldots + 2 + 1 = \sum_{i=1}^{n} i$$

$$= \frac{n(n+1)}{2}$$

# Bubble Sort Analysis

- Assume that the implementation of the sequence is such that the accesses to elements and the swaps of elements performed by bubble-sort take *O*(1) time each. That is, the running time of the *i*th pass is O(n-i+1), then the overall running time is:

- The time complexity of bubble sort is:

$$O(n^2)$$

# Sorting

- In a sorting problem we are given a sequence of $n$ objects, stored in a linked list or an array, together with some comparator defining a total order on these objects
- we want to produce an ordered representation of these objects.
- To allow for sorting of either representation, we describe our sorting algorithm at a high level for sequences and explain the details needed to implement it for linked lists and arrays.
- To sort a sequence $S$ with $n$ elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

# Merge Sort

***Divide:*** If $S$ has zero or one element, return $S$ immediately; it is already sorted. Otherwise ($S$ has at least two elements), remove all the elements from $S$ and put them into two sequences, $S1$ and $S2$, each containing about half of the elements of $S$; that is, $S1$ contains the first $\lceil n/2 \rceil$ elements of $S$, and $S2$ contains the remaining $\lfloor n/2 \rfloor$ elements.

2. ***Recur:*** Recursively sort sequences $S1$ and $S2$.

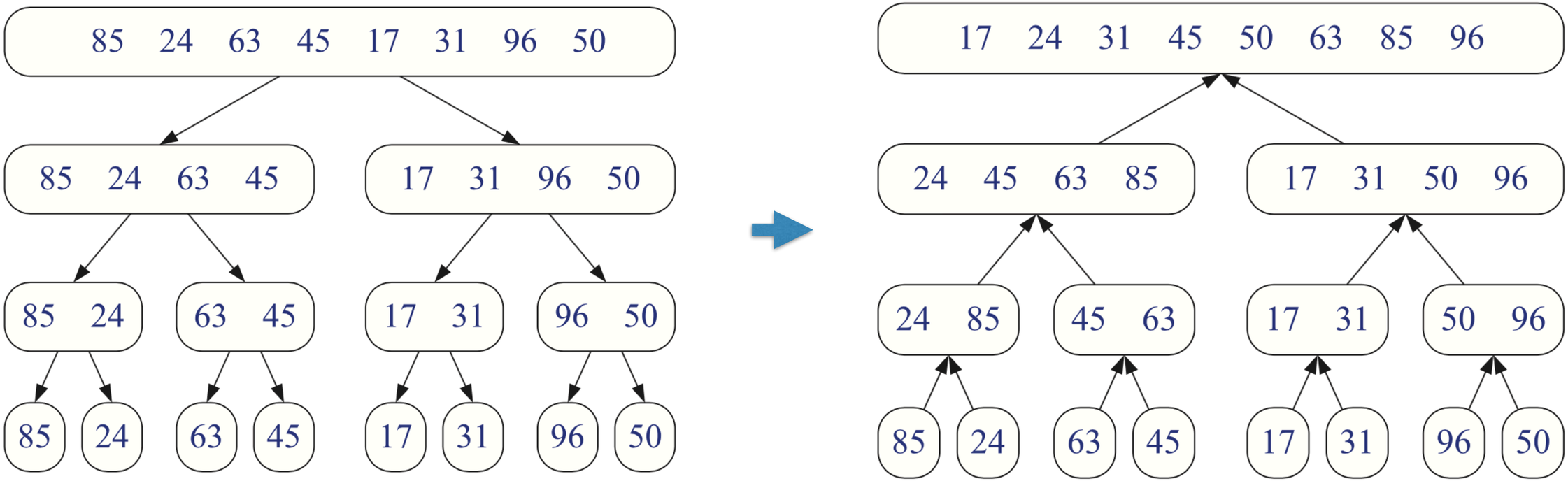
3. ***Conquer:*** Put back the elements into $S$ by merging the sorted sequences $S1$ and $S2$ into a sorted sequence.

# Merge Sort

## recursive merge sort:

```
1:  function MergeSort(A, i, j)
2:      Input array A, positive integers i, j between 0 and n − 1
3:      if j − i ≤ 1 then                    ▷ length 0 or 1, so already sorted
4:          return
5:      end if
6:      mid = (i + j)/2
7:      mergeSort(A, i, mid)                 ▷ first recursive call on left side
8:      mergeSort(A, mid, j)                 ▷ second recursive call on right side
9:      merge(A, i, mid, j)
10: end function
```

# Merge Sort


85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
85
24
63
45
17
31
96
50
17 24 31 45 50 63 85 96
24 45 63 85
17 31 50 96
24 85
45 63
17 31
50 96
85
24
63
45
17
31
96
50

# Merge Sort

how does the merge work?

```
function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

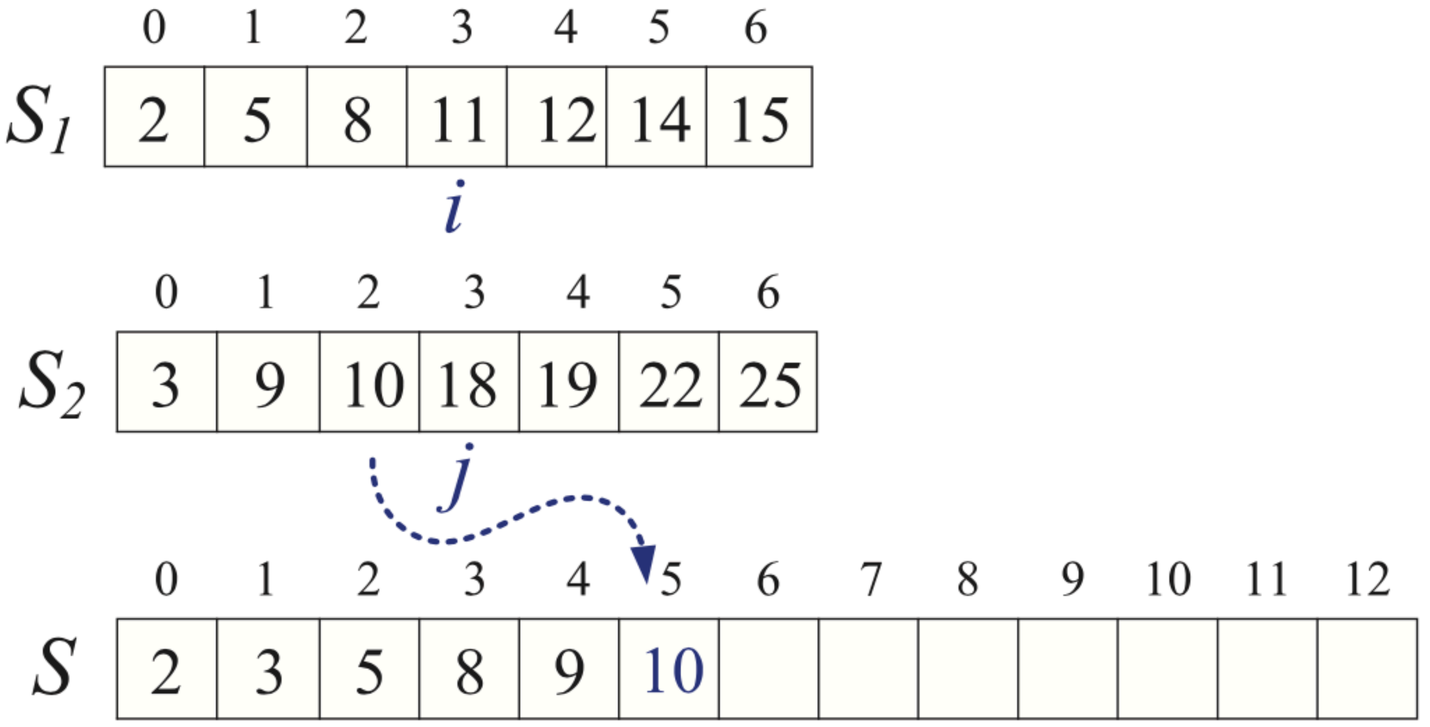
# Merge Sort

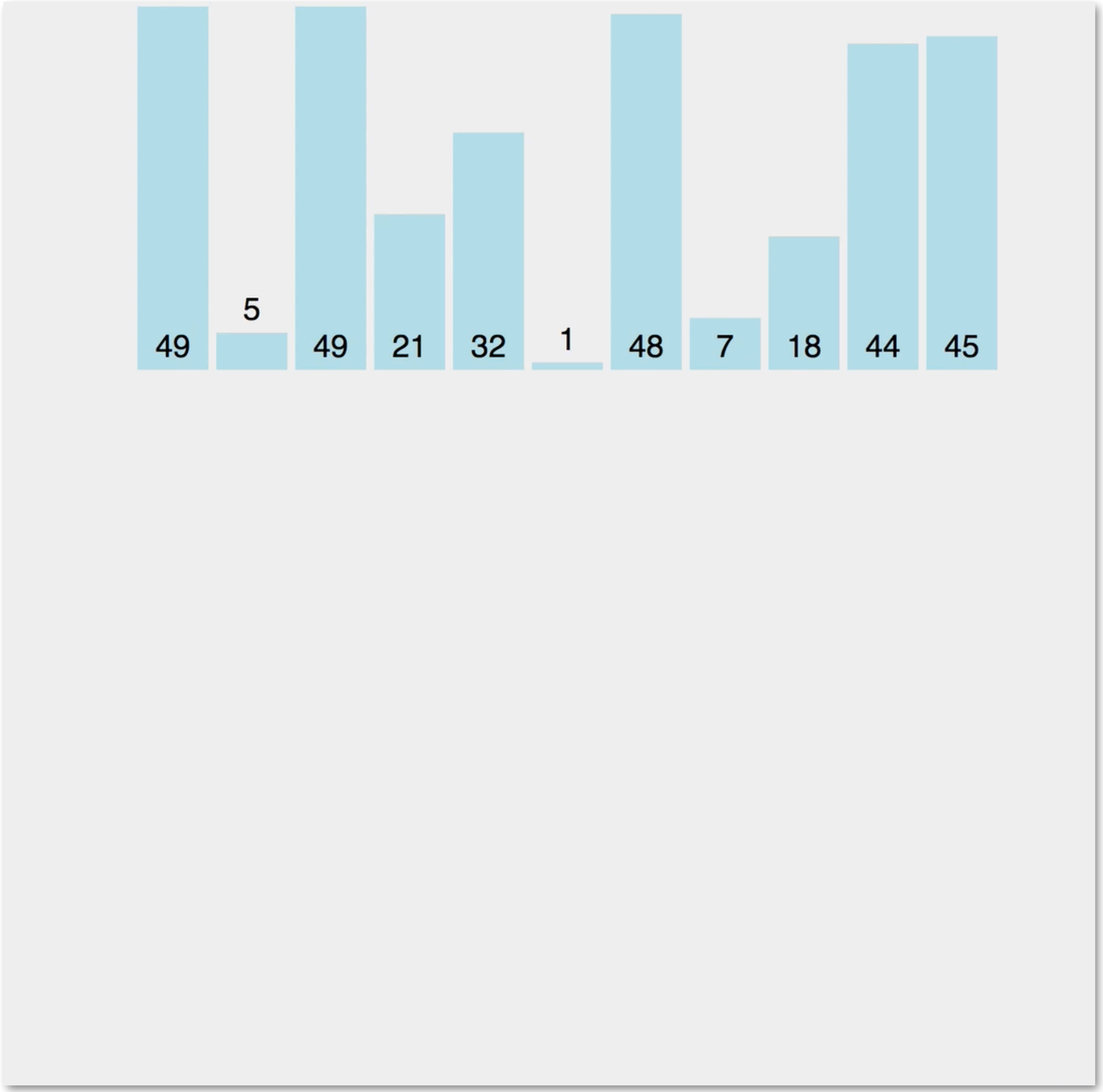
## how does the merge work?

```
1:  function MERGE(A, l, m, r)
2:      Input array A, positive integers l, m, r
3:      L ← A[l : m]                          ▷ new temporary subarrays
4:      R ← A[m : r]
5:      i ← 0                                 ▷ initial indexes of subarrays
6:      j ← 0
7:      k ← l                                 ▷ initial index of merged subarray
8:      while i < m − l + 1 and j < r − m do
9:          if L[i] ≤ R[j then
10:             A[k] ← L[i]
11:             i ← i + 1
12:         else
13:             A[k] ← R[j]
14:             j ← j + 1
15:         end if
16:         k ← k + 1
17:     end while
18:     while i < m − l + 1 do        ▷ Copy remaining elements of L if any
19:         A[k] ← L[i]
20:         i ← i + 1
21:         k ← k + 1
22:     end while
23:     while j < r − m do            ▷ Copy remaining elements of R if any
24:         A[k] ← R[j]
25:         j ← j + 1
26:         k ← k + 1
27:     end while
28: end function
```

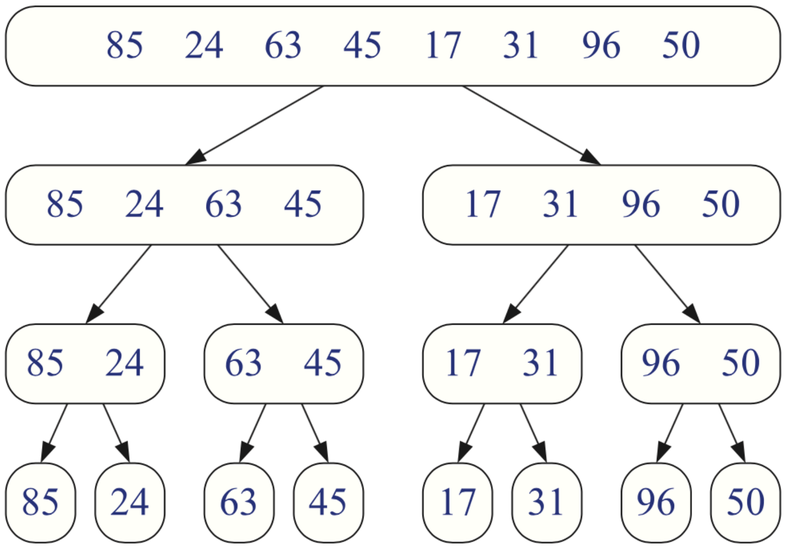
# Merge Sort

```
1:  function MERGE(A, l, m, r)
2:      Input array A, positive integers l, m, r
3:      L ← A[l : m]                                  ▷ new temporary subarrays
4:      R ← A[m : r]
5:      i ← 0                                         ▷ initial indexes of subarrays
6:      j ← 0
7:      k ← l                                         ▷ initial index of merged subarray
8:      while i < m − l + 1 and j < r − m do
9:          if L[i] ≤ R[j then
10:             A[k] ← L[i]
11:             i ← i + 1
12:         else
13:             A[k] ← R[j]
14:             j ← j + 1
15:         end if
16:         k ← k + 1
17:     end while
18:     while i < m − l + 1 do                        ▷ Copy remaining elements of L if any
19:         A[k] ← L[i]
20:         i ← i + 1
21:         k ← k + 1
22:     end while
23:     while j < r − m do                            ▷ Copy remaining elements of R if any
24:         A[k] ← R[j]
25:         j ← j + 1
26:         k ← k + 1
27:     end while
28: end function
```




49
5
49
21
32
1
48
7
18
44
45

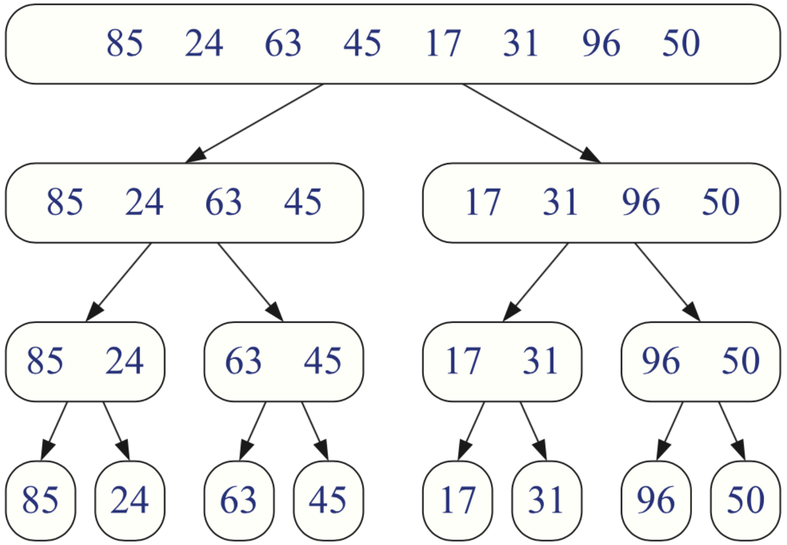
# Complexity Analysis




we assume that the input sequence S and the auxiliary sequences S1 and S2, created by each recursive call of merge-sort, are implemented by either arrays or linked lists (the same as S), so that merging two sorted sequences can be done in linear time.

the time spent at level *v* is proportional to the size at level v. Merging also takes linear time

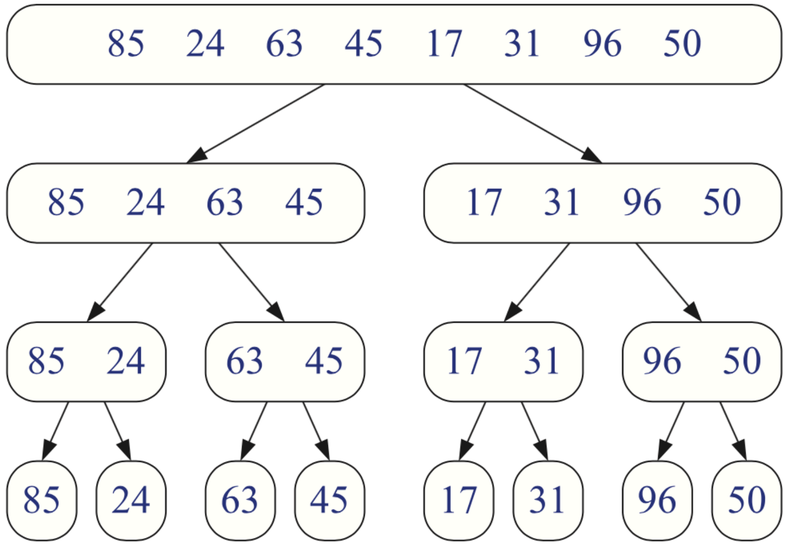
# Complexity Analysis

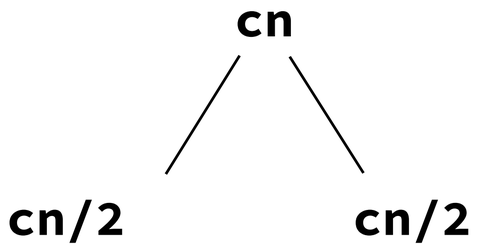



for the original problem, we have a cost of $cn$,

plus the two subproblems, each costing T($n$/2).

# Complexity Analysis

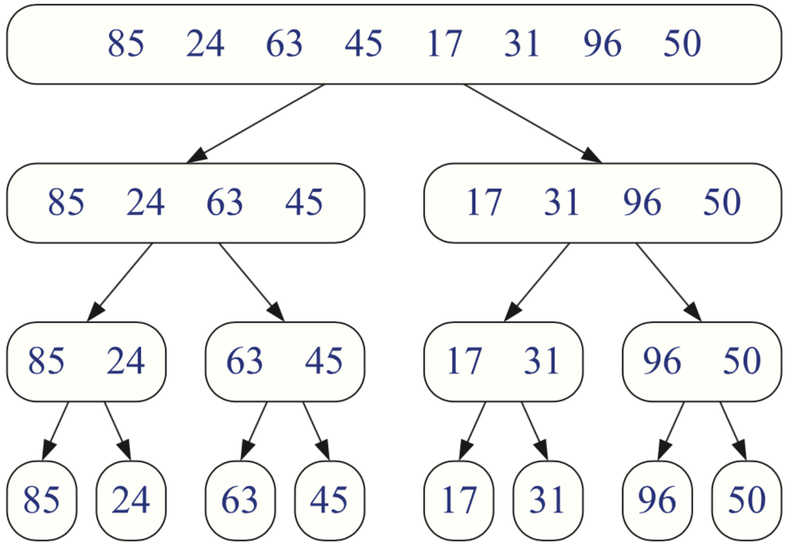


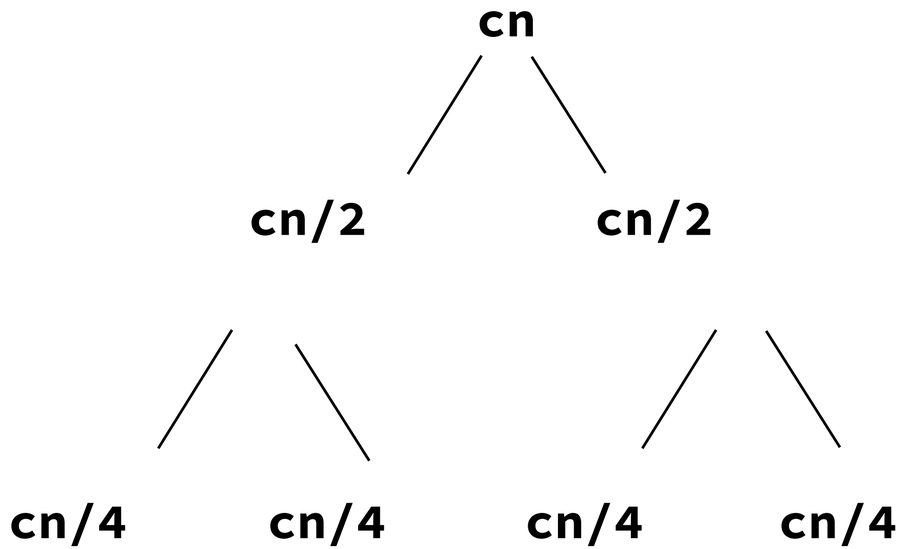




for the original problem, we have a cost of *cn*,

plus the two subproblems, each costing T(*n*/2).

# Complexity Analysis

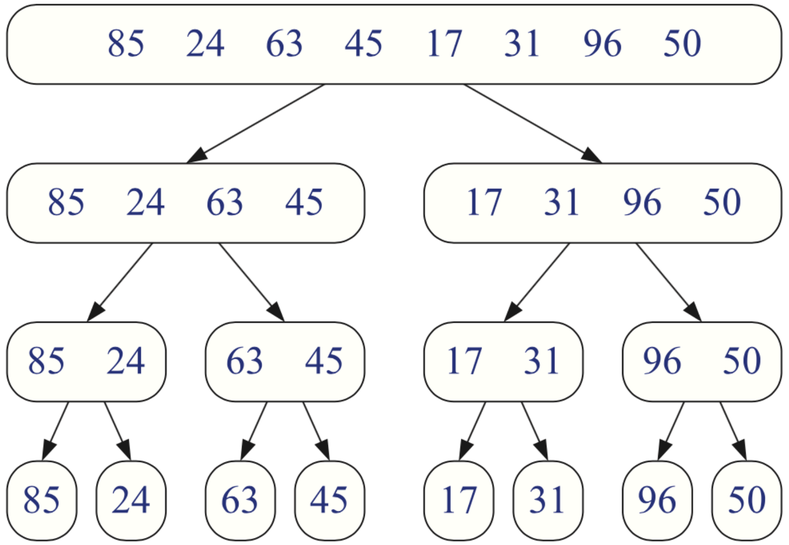



for the original problem, we have a cost of *cn*,

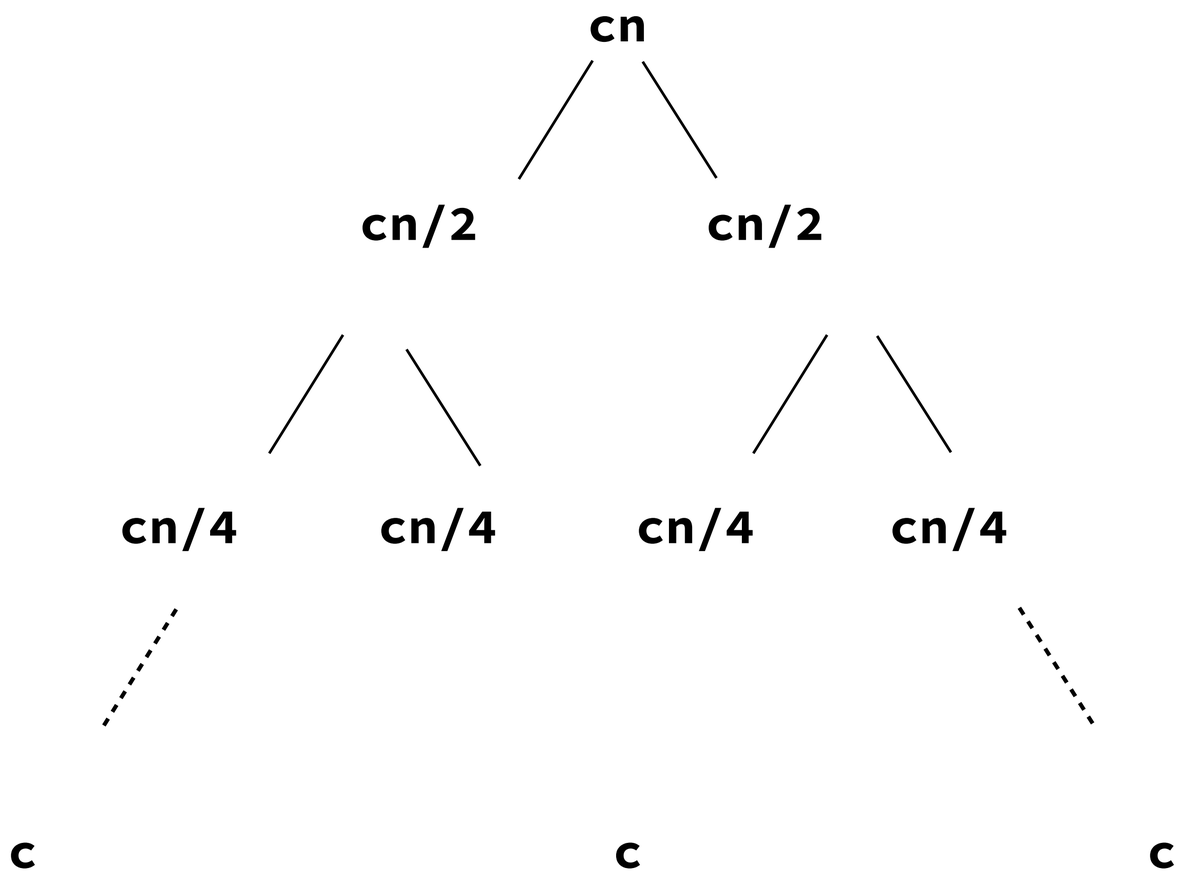
plus the two subproblems, each costing T(*n*/2).

# Complexity Analysis



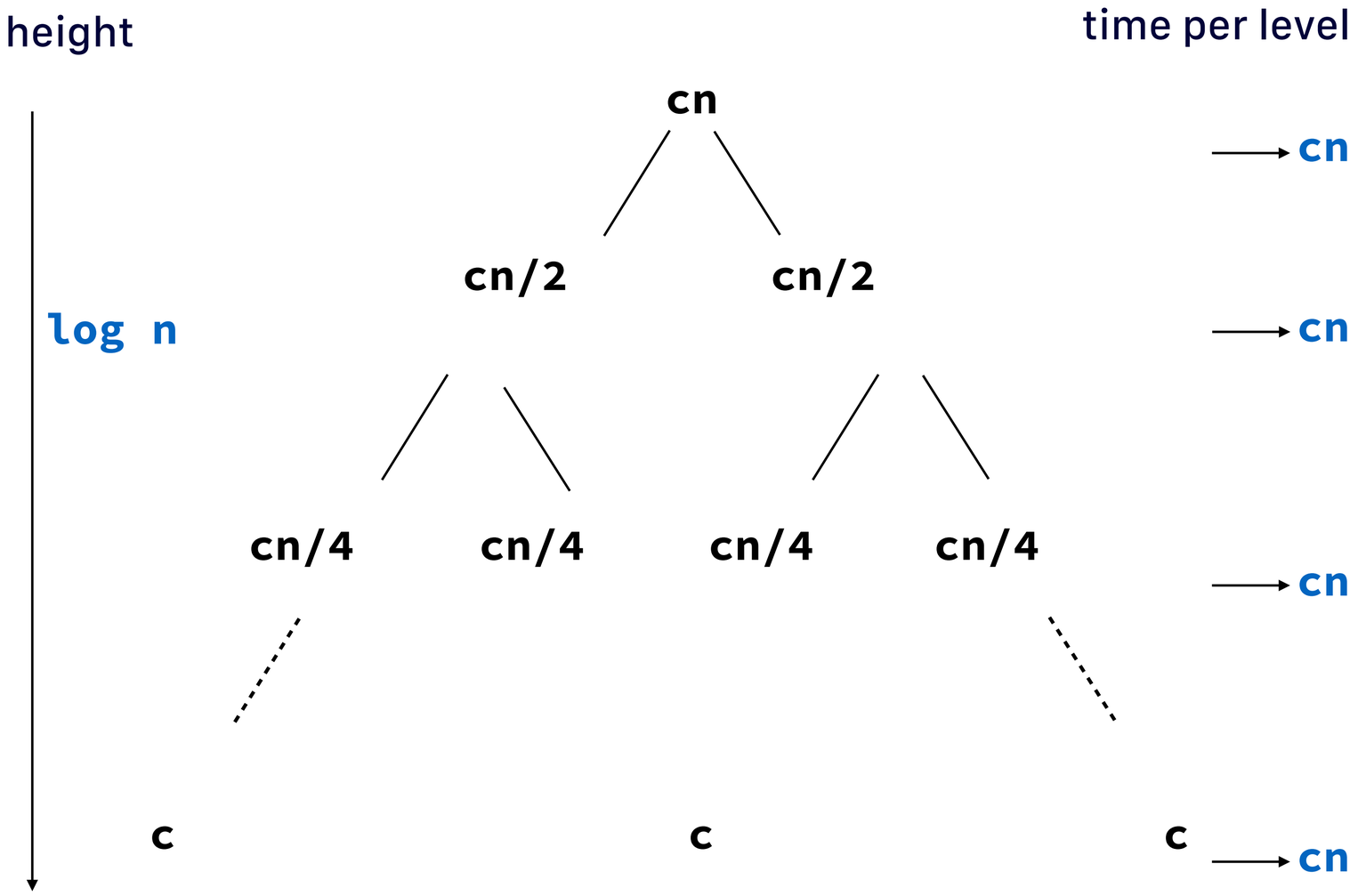

for the original problem, we have a cost of *cn*,

plus the two subproblems, each costing T(*n*/2).



# Complexity Analysis

- The top level has cost *cn*.
- The next level down has 2 subproblems, each contributing cost *cn*/2.
- The next level has 4 subproblems, each contributing cost *cn*/4.
- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves.
- Therefore, cost per level stays the same.
  The height of this recursion tree is log *n* and there are log *n* + 1 levels
- Total cost is sum of costs at each level of the tree. Since we have log *n* +1 levels, each costing *cn*, the total cost is
- O(n log n)



# Complexity Analysis

the running time of merge-sort is equal to the sum of the times spent at the nodes of $T$ .

$T$ has exactly $2i$ nodes at depth $i$.

this implies that the overall time spent at all the nodes of $T$ at depth $i$ is:,

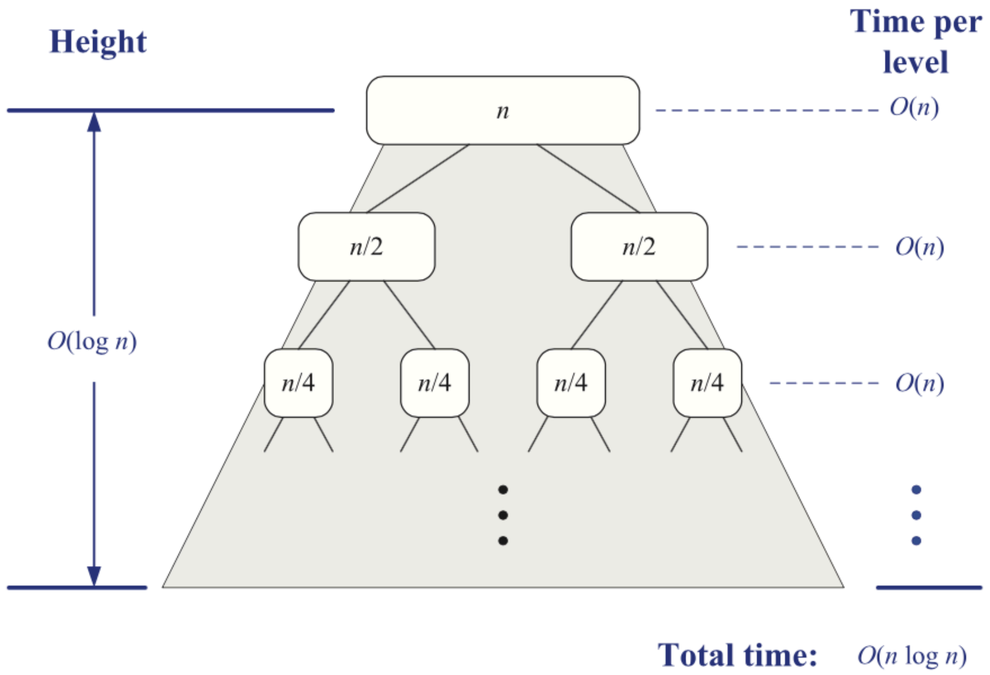
$$O\left(\frac{2^i n}{2^i}\right)$$

which is $O(n)$. The height of $T$ is $\lceil \log n \rceil$.

Thus, since the time spent at each of the $\lceil \log n \rceil + 1$ levels of $T$ is $O(n)$, we have the time complexity of merge sort is:

$$O\left(n \log n\right)$$

# Complexity Analysis




consider the recursive nature of the merge-sort algorithm.

***recurrence equation*** are useful here

Let the function $T(n)$ denote the worst-case running time of merge-sort on an input sequence of size $n$. Since merge-sort is recursive, we can characterise T($n$) in recursive terms of itself. In order to simplify our characterisation of $T(n)$, we restrict to cases where $n$ is a power of 2.

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

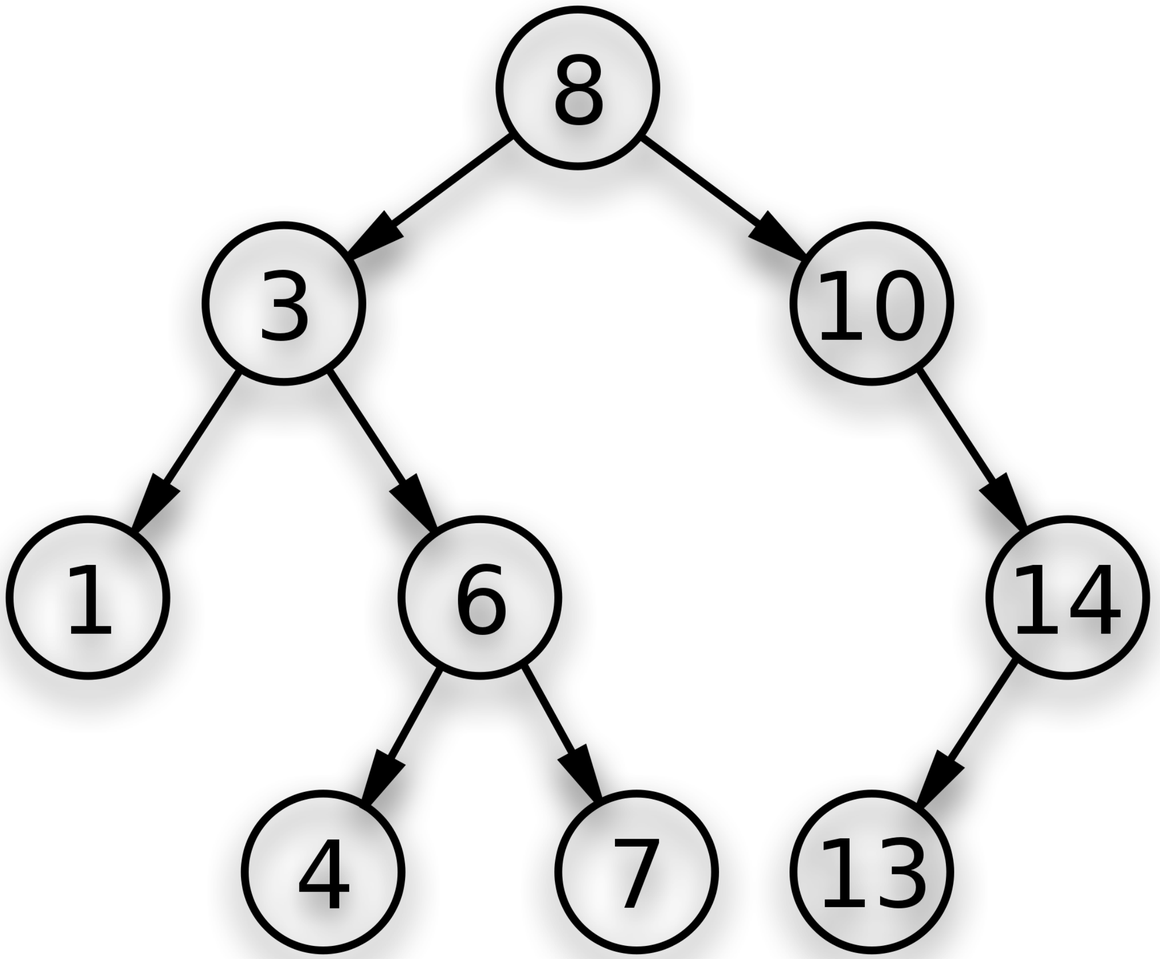
$$O\left(n \log n\right)$$

# Binary Search

A classic divide-and-conquer algorithm is binary search

Problem: find a key *k* in an array z[0, 1, . . . , n – 1] in sorted order.

We first compare k with z[n/2], and depending on the result we recurse either on the first half of z[0, . . . , n/2 – 1], or on the second half, z[n/2, . . . , n – 1].



# Binary Search

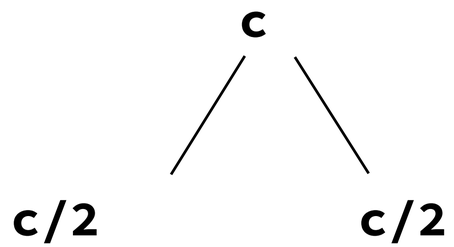
```
// initially called with low = 0, high = N-1
BinarySearch(A[0..N-1], value, low, high) {
    if (high < low)
        return not_found
    mid = (low + high) / 2
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid
```

# Binary Search

Binary Search is called on a subarray of length approximately n/2

there are c comparisons in the worst case before a recursive call is needed again. So the recurrence relation is

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

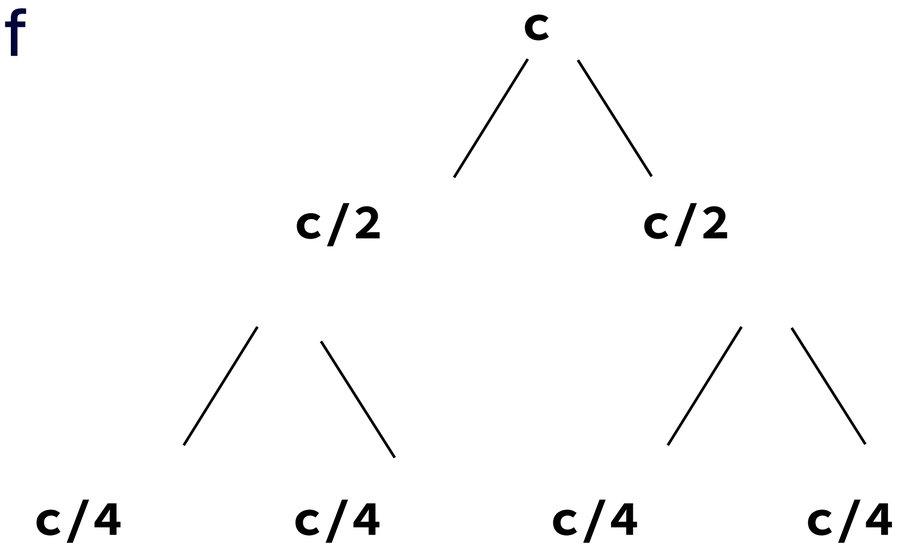


# Binary Search

Binary Search is called on a subarray of length approximately n/2

there are c comparisons in the worst case before a recursive call is needed again. So the recurrence relation is

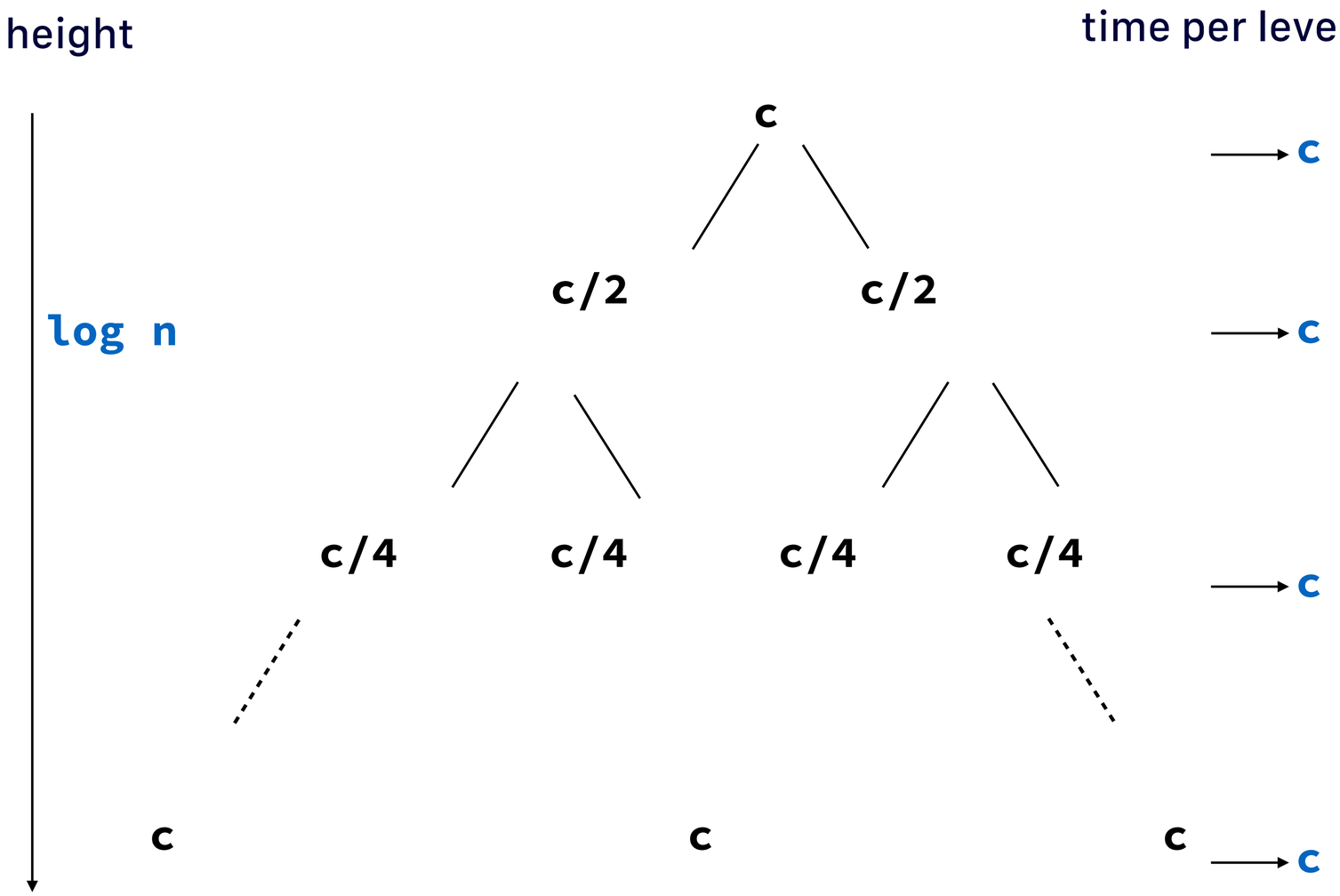
$T(n) = T(n/2) + c$



# Binary Search

$$T(n) = T(\lceil n/2 \rceil) + O(1)$$

$$O(\log n)$$



# Multiplication

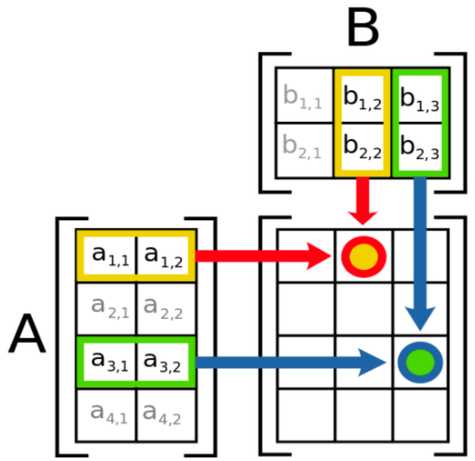
# Matrix Multiplication

Given two (2 dimensional) matrices, $A$ and $B$

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

the matrix product $A \cdot B$ is defined as:

$$C = \begin{bmatrix} (a_{00}b_{00} + a_{01}b_{10}) & (a_{00}b_{01} + a_{01}b_{11}) \\ (a_{10}b_{00} + a_{11}b_{10}) & (a_{10}b_{01} + a_{11}b_{11}) \end{bmatrix}$$



# Matrix Multiplication

**Algorithm 2**

```
1:  function MultiplyMatrix(A[n, n], B[n, n])
2:      Returns a new matrix which is matrix multiplication of A × B
3:      C = newmatrix[n][n]
4:      for i = 0 to n do
5:          for j = 0 to n do
6:              sum ← 0
7:              for k = 0 to n do
8:                  sum ← sum + A[i][k] × B[k][j]
9:              end for
10:             C[i][j] ← sum
11:         end for
12:     end for
13:     return C
14: end function
```

# Matrix Multiplication

**Algorithm 3**

```
1:  function MultiplyMatrix(A[n, n], B[n, n])
2:      Returns a new matrix which is matrix multiplication of A × B
3:      C = newmatrix[n][n]
4:      for i = 0 to n do                                   ▷ O(n)
5:          for j = 0 to n do                               ▷ O(n)
6:              sum ← 0                                     ▷ O(1)
7:              for k = 0 to n do                           ▷ O(n)
8:                  sum ← sum + A[i][k] × B[k][j]           ▷ O(1)
9:              end for
10:             C[i][j] ← sum                               ▷ O(1)
11:         end for
12:     end for
13:      return C                                           ▷ Total: O(n^3)
14: end function
```

# Matrix Multiplication

Volker Strassen first published this algorithm in 1969 and proved that the $n_3$ general matrix multiplication algorithm wasn't optimal.

The **Strassen algorithm** is only slightly better, but its publication resulted in much more research about matrix multiplication that led to faster approaches

The Strassen algorithm has complexity $O(n^{2.807355})$

One example is Coppersmith-Winograd algorithm which is $O(n^{2.375477})$

# Matrix Multiplication

each of these 4 is an n/2 block

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

divide-and-conquer strategy: compute the size-n product XY, recursively compute eight size-n/2 products AE, BG, AF, BH, CE, DG, CF, DH, and then do a few $O(n^2)$- time additions. The total running time is described by the recurrence relation

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

# Matrix Multiplication

Strassen discovered how to improve the efficiency with the help of some clever algebra. It turns out XY can be computed from just seven n/2 × n/2 subproblems. How did Strassen discover it?

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$P_1 = A(F - H)$$
$$P_2 = (A + B)H$$
$$P_3 = (C + D)E$$
$$P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H)$$
$$P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

# Matrix Multiplication

There are 7 multiplications, so the new running time is:

$$T(n) = 7T(n/2) + O(n^2)$$

write out a few values of the recurrence:

$$\begin{aligned}
T(1) &= 1 \\
T(2) &= 7T(2/2) = 7 \\
T(4) &= 7T(4/2) = 7T(2) = 7^2 \\
T(8) &= 7T(8/2) = 7T(4) = 7^3 \\
T(16) &= 7T(16/2) = 7T(8) = 7^4
\end{aligned}$$

# Matrix Multiplication

There are 7 multiplications, so the new running time is:

$$T(n) = 7T(n/2) + O(n^2)$$

and we realise that:

$$T(n) = 7^{\log n}$$

*use induction to prove this is true*

# Matrix Multiplication

There are 7 multiplications, so the new running time is:

$$T(n) = 7T(n/2) + O(n^2)$$

and we realise that:

$$7^{\log n} = n^{\log 7}$$
$$T(n) = n^{\log 7} \sim n^{2.81}$$

This is the complexity of Strassens multiplication algorithm.

# Caveats

in practice, the $O(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide.

If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time.

If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big O notation is great to get you started, and tells us to throw away very inefficient algorithms.

But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

# Caveats

When is Strassen's worth it?

If we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1000$, assuming no communication costs

Higher communication costs drive up the n at which Strassen's becomes useful very quickly.

Even at n = 1000, naive matrix-multiply requires 1e9 operations; we can't really do much more than this with a single processor.

Strassen's is mainly interesting as a theoretical idea.

# Multiplication

**long multiplication**, or **grade-school multiplication**, sometimes called **Standard Algorithm:**

multiply the multiplicand by each digit of the multiplier and then add up all the properly shifted results.

[2,3,9,5,8,2,3,3] x [0,0,0,0,5,8,3,0]

```
   23958233
×      5830
-----------------
    00000000 (=     23,958,233 ×     0)
    71874699 (=     23,958,233 ×    30)
   191665864 (=     23,958,233 ×   800)
+ 119791165  (=     23,958,233 × 5,000)
-----------------
 139676498390 (= 139,676,498,390       )
```

```
1:  function MULTIPLY(a[1...p], b[1...p], base)
2:      product ← [1...p + q]
3:      for b_i = 1 doq                                          n
4:          for a_i dop                                          n
5:              product[a_i + b_i − 1] += carry + a[a_i] * b[b_i]
6:              carry = product[a_i + b_i − 1]/base
7:              product[a_i + b_i − 1] = product[a_i + b_i − 1]  mod  base
8:          end for
9:          product[b_i + p] += carry
10:     end for
11:     return product
12: end function                                                 O(n²)
```

# Multiplication

Let's look at the method for numbers with one, two, or four digits, and then for numbers of any length.

The simplest case is multiplication of two numbers consisting of one digit each (n = 1). Multiplying them needs a single basic operation, namely one multiplication of digits, which immediately gives the result.

$$a = p \times 10 + q$$

$$b = r \times 10 + s$$

# Multiplication

The next case we look at is the case n = 2, that is, the multiplication of two numbers a and b having two digits each. We split them into halves, that is, into their digits:

$$a = p \times 10 + q$$
$$b = r \times 10 + s$$

For example, we split the numbers a = 78 and b = 21 like this:

p=7, q=8, and r=2, s=1.

# Multiplication

We can now rewrite the product a × b in terms of the digits:

$$\begin{aligned} a \times b &= (p \times 10 + q) \times (r \times 10 + s) \\ &= (p \times r) \times 100 + (p \times s + q \times r) \times 10 + q \times s \end{aligned}$$

Continuing the example a = 78 and b = 21, we get

$78 \cdot 21 = (7 \cdot 2) \cdot 100 + (7 \cdot 1 + 8 \cdot 2) \cdot 10 + 8 \cdot 1 = 1638.$

# Multiplication

The product of complex numbers seemed to involve four multiplications

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

Gauss realised you do it with just 3, since:

$$(bc + ad) = (a + b)(c + d) - ac - bd$$

# Multiplication

Writing the product a×b of the two-digit numbers a and b shows how it can be computed using four multiplications of one-digit numbers, followed by additions. This is precisely what long multiplication does.

Karatsuba had an idea that enables us to multiply the two-digit numbers a and b with just three multiplications of one-digit numbers. These three multiplications are used to compute the following auxiliary products:

$$u = p \times r$$

$$v = (q - p) \times (s - r)$$

$$w = q \times s$$

# Karatsuba

Why does all this help to multiply a and b? The answer comes from this formula:

$$u + w - v = p \times r + q \times s - (q - p) \times (s - r)$$
$$= p \times s + q \times r.$$

Karatsuba's trick consists in using this formula to express the product a × b in terms of the three auxiliary products u, v, and w:

$$a \times b = u \times 10^2 + (u + w - v) \times 10 + w.$$

$$u = 7 \times 2 \quad = 14$$
$$v = (8 - 7) \times (1 - 2) \quad = -1$$
$$w = 8 \times 1 \quad = 8$$

$$78 \times 21 = 14 \times 100 + (14 + 8 - (-1)) \times 10 + 8 \quad = 1400 + 230 + 8$$
$$= 1638$$

We have used two subtractions of digits, three multiplications of digits, and several additions and subtractions of digits to combine the results of the three multiplications.

# Karatsuba

The general form of Karatsuba's method is this:

Two numbers a and b, each consisting of $n=2\times2\times2\times\cdots\times2=2^k$ digits, are split into

$$a = p \times 10^{n/2} + q$$

$$b = r \times 10^{n/2} + s$$

$$a \times b = p \times r \times 10^n + (p \times r + q \times s - (q - p) \times (s - r)) \times 10^{n/2} + q \times s$$

# Multiplication

division

$$x = \boxed{x_L}\boxed{x_R}$$

$$y = \boxed{y_L}\boxed{y_R}$$

multiply the smaller numbers, recursive call

$$x_L, x_R = leftmost\lceil n/2 \rceil, rightmost\lfloor n/2 \rfloor \quad \text{bits of x}$$

$$y_L, y_R = leftmost\lceil n/2 \rceil, rightmost\lfloor n/2 \rfloor \quad \text{bits of y}$$

$$P_1 = multiply(x_L, y_L)$$

$$P_2 = multiply(x_R, y_R)$$

$$P_3 = multiply(x_L + x_R, y_L + y_R)$$

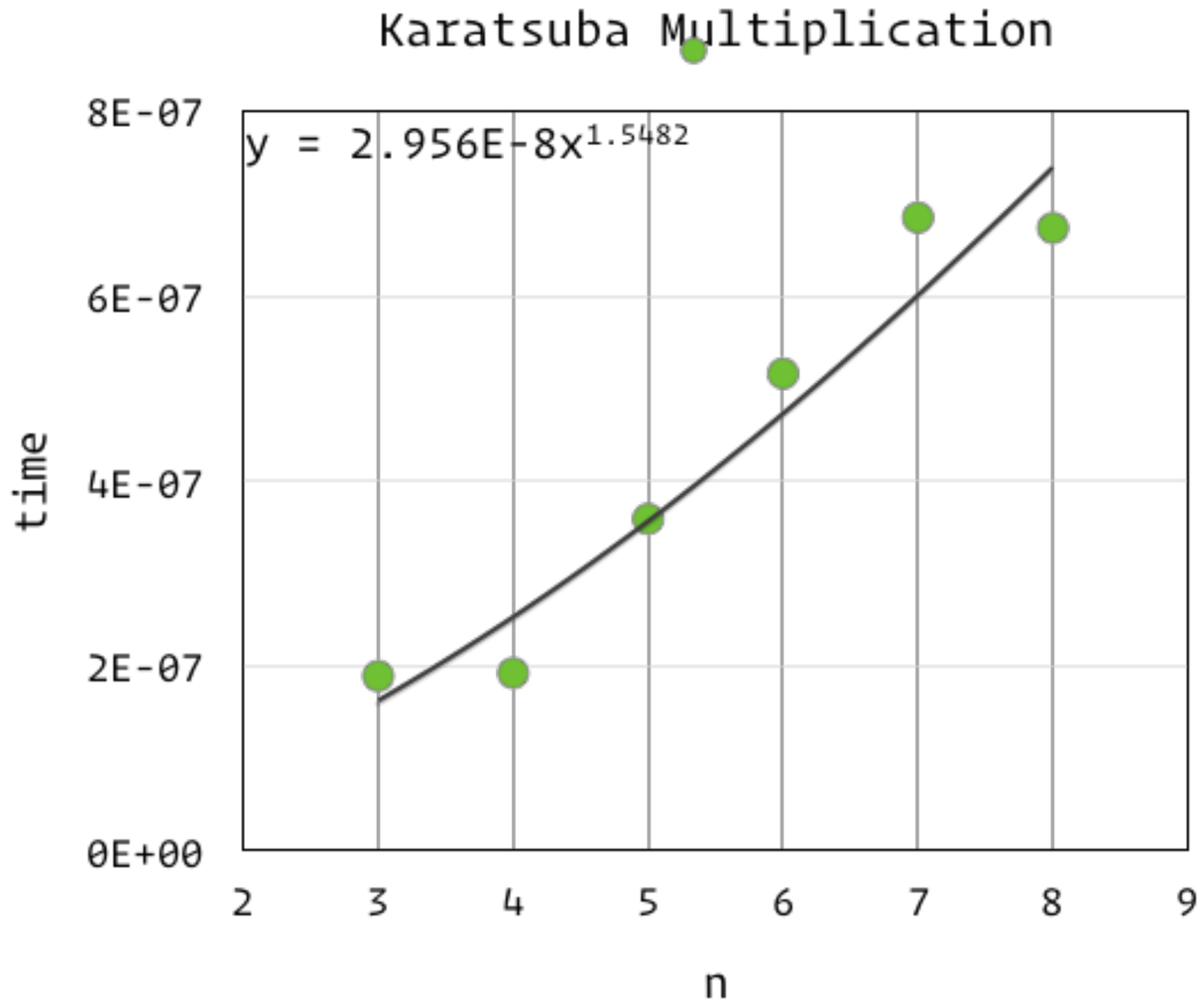
combine

$$P_1 10^{2m} + (P_3 - P_1 - P_2)10^m + P_2$$

# Assignment

(30marks)

An alternative multiplication algorithm is that of Karatsuba. The basic step of this algorithm is a formula that allows you to compute the product of two large numbers $num_1$ and $num_2$ using three multiplications of smaller numbers, each with about half as many digits as $num_1$ or $num_2$, plus some additions and digit shifts. The pseudocode is below.



# Divide and Conquer

The divide-and-conquer design strategy involves the following steps:

1. Divide an instance of a problem into one or more smaller instances.
2. Conquer (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, *use recursion to do this*.
3. If necessary, combine the solutions to the smaller instances to obtain the solution to the original instance.

some algorithms like Binary Search do not required a combination step

# Divide and Conquer

The real work is done piecemeal, in three different places:

- in the partitioning of problems into subproblems
- at the very tail end of the recursion, when the subproblems are so small that they are solved outright
- in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.