Data Structures and Algorithms Lecture slides: Analyzing algorithms: recurrences, Brassard section 4.7

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

Current semester map

Introduction : basic operations, worst case analysis

Asymptotic notations

Analyzing iterative algorithms

Analyzing recursive algorithms

Basic data structures : arrays, linked lists, etc.

Topics cover

Recursive algorithms

- We occasionally see recursions in the "real" world :
  - Two almost parallel mirrors http://www.flickriver.com/photos/doc-click/150637729
  - A video camera pointed at the monitor (feedback) https://softology.com.au/videofeedback/videofeedback.htm
- In computer science, a recursion is a code in which the name of the function appears in the function itself.
- Algorithms that make use of recursions are call recursive algorithms.
- There is a particular type of recursive algorithms we will often meet :divide-and-conquer algorithms (D&C). Examples of D&C algorithms are :
  - Binary Search
  - Mergesort

Iterations vs recursions

Iterative algorithms are driven by loops which repeat the same operations for different values of the loop index

Iterative Algo(A[1..n])
$total = 0$
for $i = 1$ **to** n **do**
    $total = total + i$

Recursive algorithms (kind of) call the same function on each value of the loop index, i.e. repeatedly calling the same algo for executing a single loop iteration

Recursive Algo(n)
if $n == 1$ **then return** 1 ;
return n + **Recursive Algo**(n-1)

A simple recursive algorithm

Find factorial of n (i.e. $n \times n-1 \times n-2 \times \ldots \times 2 \times 1$)

```
factorial (n)
        if (n ≤ 1)
           return 1 ;
        else
           return n × factorial(n − 1) ;
```

As we can see, this algorithm calls a function "factorial($n-1$)" which has the same name as the algorithm, this is a recursive algorithm because it calls itself to compute the factorial of an integer n

Running time analysis of recursive algorithms

The running time of recursive algorithms cannot be analyzed in the same way as iterative algorithms

There are no loops, so summations described in the previous lecture to count the number of basic operations in loops do not apply for recursive algorithms

Rather recurrence relations will be used to express the number of times basic operations are executed in recursive algorithms

Recurrence relations are equations where at least one term appears on both sides of the equation :

$$T(n) = 2T(n-1) + n^2 + 3$$

Another Example of recursive algorithm

Suppose we want to write an algorithm that outputs the sequence of natural numbers in decreasing order from n to 1, we will call this the CountDown(n) problem.

For example, an algo that computes CountDown(5) will output the sequence '5 4 3 2 1'.

An iterative algorithm for CountDown(n) is the following loop :

function *CountDown*(n)
    for $(i = n; i > 0; i--)$
        write i ;

Now, let's consider how to design a recursive algorithm for the CountDown(n) problem.

Recursive CountDown(n)

CountDown(n) outputs n followed by the sequence from $n-1$ down to 1.

The sequence $n-1$ down to 1 is actually the output of CountDown($n-1$).

The output from CountDown(n) is n followed by the output from CountDown($n-1$).

Thus, we can obtain a recursive version of CountDown(n) as follows :

function *CountDown*(n)
    write n ;
    CountDown($n-1$) ;

Nice, but something is wrong here. What is it ?

Recursive CountDown(n) : Error

The problem is, CountDown never stops, it will output as well, 0, -1, -2, ...

Execute	Output	then call
CountDown(3)	3	**CountDown(2)**
CountDown(2)	2	**CountDown(1)**
CountDown(1)	1	**CountDown(0)**
CountDown(0)	0	**CountDown(-1)**
CountDown(-1)	-1	**CountDown(-2)**
⋮	⋮	⋮

To fix this problem, we modify the code such that a call to CountDown(0) produces no output and exit the algorithm rather than making a recursive call.

This new instruction is called the *stopping condition* which ends the recursion

Recursive CountDown(n) : Fixed

Calls to CountDown(n) when $n < 1$ should produce no output.

The following version is correct :

```
function CountDown(n)
        if n > 0 then
            write n;
            CountDown(n − 1);
```

For any recursive algorithm, one must ensure that at some point the function stop calling itself.

Note that each recursive call is made on a different "problem instance", a problem instance that is smaller, otherwise the recursion will also not stop !

Back to factorial

- Recursive definition of factorial :

$$n! = \begin{cases} 1 & \text{when } n = 1 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

- Example :

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 \times (1)! = 2 \times 1 = 2 \\ 3! &= 3 \times (2)! = 3 \times 2 = 6 \\ 4! &= 4 \times (3)! = 4 \times 6 = 24 \end{aligned}$$

- Recursive algorithm :

```
factorial(n)
        if (n ≤ 1)
          return 1;
        else
          return n× factorial(n − 1);
```

Recurrence relations

A recurrence relation is an equation in which at least one term appears on both sides of the equation such as

$$R(n) = 2R(n-1) + n^2 + 3$$

in which the term $R()$ appears on both side of the equation.

For recursive algorithms, running time is expressed using recurrence relations

The recurrence relation must be solved, i.e. find a closed form, to obtain a finite mathematical expression that will be used to classify the algorithm in an appropriate complexity class.

Recurrence for Factorial

```
factorial(n)
        if (n ≤ 1)
          return 1;
        else
          return n× factorial(n − 1);
```

Running-time of factorial, $T(n)$, is given by the following recurrence relation :

$$T(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ T(n-1)+1 & \text{otherwise} \end{cases}$$

Types of terms in recurrences

$$T(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

- This recurrence relation has 2 types of terms : The *recursive* term $T(n-1)$ and the *non-recursive* term 1.
- The *recursive* term displays the size of the next smaller instance (here it is $n-1$) and stands for the running time $T(n-1)$ of this smaller instance.
- The *non-recursive* term expresses the number of basic operations executed each time the computer enters in the recursive function.

The solution to this recurrence is $T(n) = n \Rightarrow T(n) \in O(n)$

Binary Search (a recursive version)

BinarySearch$(L[i..j], x)$
    if $i = j$ **then return** i
    $k = \lfloor \frac{(i+j)}{2} \rfloor$
    if $x \leq L[k]$ **then**
        return $BinarySearch(L[i..k], x)$
    else
        return $BinarySearch(L[k+1..j], x)$

It is assumed x is in the array L and L is sorted.

The search for value x starts with the entry $k = \lfloor \frac{(i+j)}{2} \rfloor$ of L, if $x > L[k]$ then search proceed on right side of the array, otherwise on the left side.

Recurrence for Binary Search

BinarySearch$(L[i..j], x)$
    if $i = j$ **then return** i
    $k = \lfloor \frac{(i+j)}{2} \rfloor$
    if $x \leq L[k]$ **then**
        return $BinarySearch(L[i..k], x)$
    else
        return $BinarySearch(L[k+1..j], x)$

- To solve an instance of binary search, we need to do a compare operation, followed by an instance of binary search of half the size.
- Thus, the recurrence for the run-time of this algorithm is

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

The time $T(n)$ needed to solve an instance of side n is equal to the time $T(\lfloor \frac{n}{2} \rfloor)$ needed to solve an instance of size $\lfloor \frac{n}{2} \rfloor + 1$

Recurrence relations with their closed form solution

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

Solution : $T(n) = \log n \Rightarrow T(n) \in O(\log n)$

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

Solution : $T(n) = n \log n + n \Rightarrow T(n) \in O(n \log n)$

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

Solution : $T(n) = \log \log n \Rightarrow T(n) \in O(\log \log n)$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases}$$

Solution : $T(n) = n \log n \Rightarrow T(n) \in O(n \log n)$

Analyzing the running time of recursive algorithms

To analyze the running time of a recursive algorithm, we first identify the number of elementary operations execute in each recursive call.

Write the recurrence relation (rather than the summation) expressing the number of basic operations executed based on n, the input.

Find the closed form of the recurrence relation.

This lecture focuses on how to find closed forms to recurrence relations. Three methods are described :

- The substitution method
- The recursion tree method
- The Master Theorem

Solving recurrence relations : Substitution method

The substitution method consists of two steps :

1. Guess the form of the closed form.
2. Use mathematical induction to show the guess is correct.

It can be used to obtain either upper ($O()$) or lower bounds ($\Omega()$) on a recurrence.

A good guess is vital when applying this method. If the initial guess is wrong, the guess needs to be adjusted later.

Solve the recurrence $T(n) = T(n-1) + 2n - 1,\ T(0) = 0$

Form a guess using "forward substitution" :

n	0	1	2	3	4	5
$T(n)$	0	1	4	9	16	25

Guess is $T(n) = n^2$. Proof by induction :

Base case $n = 0$: $T(0) = 0 = 0^2$
Induction step : Assume the inductive hypothesis is true for $n = n - 1$.
We have

$$\begin{aligned} T(n) &= T(n-1) + 2n - 1 \\ &= (n-1)^2 + 2n - 1 \text{ (Induction hypothesis)} \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2 \end{aligned}$$

Solve the recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n$, $T(0) = 0$
Guess using forward substitution :

n	0	1	2	3	4	5	8	16	32	64
$T(n)$	0	1	3	4	7	8	15	31	63	127

Guess : $T(n) \leq 2n$. Proof by induction :

Base case $n = 0$: $T(n) = 0 \leq 2 \times 0$
Induction step : Assume the inductive hypothesis is true for some $n = \lfloor \frac{n}{2} \rfloor$. We have

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + n \\ &\leq 2\lfloor \frac{n}{2} \rfloor + n \text{ (Induction hypothesis)} \\ &\leq 2(n/2) + n \\ &= 2n \\ &\in O(n) \end{aligned}$$

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

Since the input size is divided by 2 at each recursive call, we can guess that $T(n) \leq cn \log n$ for some constant c (that is, $T(n) = O(n \log n)$)

Base case : $T(1) = 1 = n \log n + n$ (note : we need to show that our guess holds for some base case (not necessarily $n = 1$, some small n is ok).
Induction step : Assume the inductive hypothesis holds for $n/2$ (n is a power of 2) : $T(n/2) \leq c\frac{n}{2} \log \frac{n}{2}$.

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

Which is true if $c \geq 1$

Exercise 1

Assume that the running time $T(n)$ satisfies the recurrence relation

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n.$$

Show that $T(n) \in O(n \log n)$ if $T(0) = 0$ and $T(1) = 1$.

Since the guess is already given here, you only need to prove by induction this guess is correct.

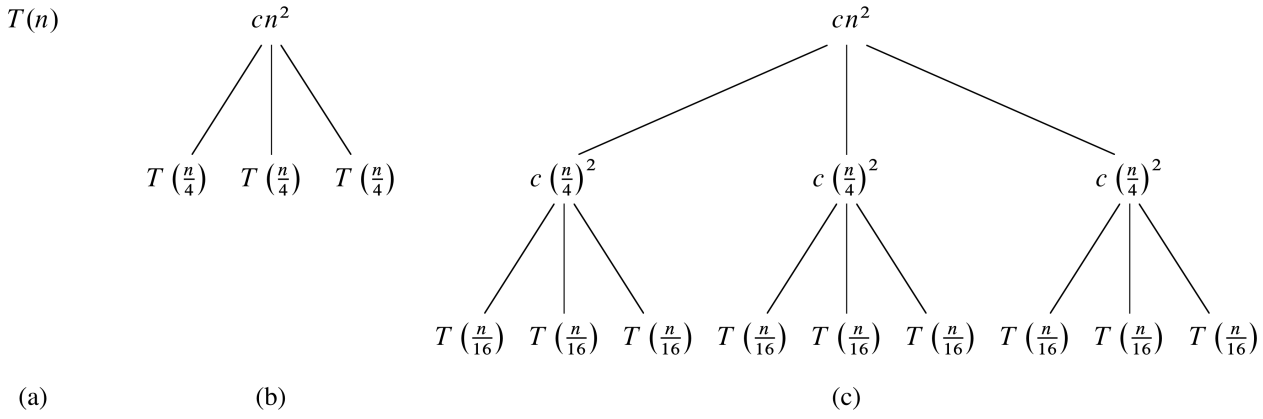
Solving recurrence relations : Recursion tree method

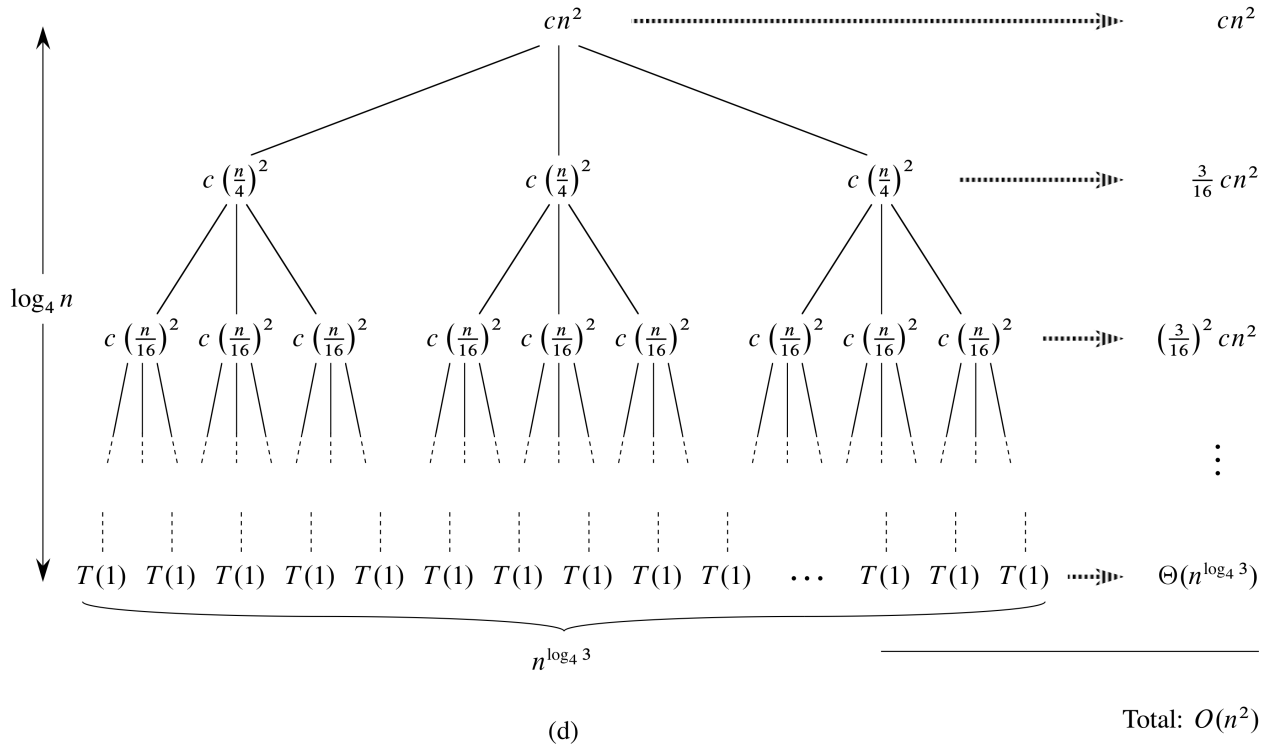
To analyze the recurrence by using the recursion tree, we will

- draw a recursion tree with cost of single call at each node
- find the running time at each level of the tree by summing the running time of each call at that level
- find the number of levels
- Last, the running time is sum of costs in all nodes

Example of a recurrence relation :

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
(a)
(b)
(c)


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$ $c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $\cdots$ $T(1)$ $T(1)$ $T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
(d)
Total: $O(n^2)$

Analysis

Number of levels in the tree :

The number of levels in the tree depends by how much subproblem sizes decrease

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$

In this example, the subproblem sizes decrease by a factor of 4 at each new level

The last level is where the size of each subproblem $= 1$, i.e. when $n/4^i = 1$

Therefore the number of levels is $i = \log_4 n$.

Analysis

Number of nodes at each level of the tree :

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$

From the recurrence we can see the number of nodes at level i is 3 times the number of nodes at level $i - 1$

In terms of level 0, the number of nodes at level $i \geq 0$ is 3^i

- level $0 = 3^0 = 1$
- level $1 = 3^1 = 3$
- $\vdots$
- level $\log_4 n = 3^{\log_4 n} = n^{\log_4 3}$ (using the rule $a^{\log b} = b^{\log a}$)

Running time

According to the recurrence relation the number of basic operations executed at level 0 is cn^2 (given the input size $= n$)

At level 1, the input size of each subproblem is $c\lfloor \frac{n}{4} \rfloor$, therefore the number of basic operations executed by each subproblem is $c(\lfloor \frac{n}{4} \rfloor)^2$

Assuming n is a power of 2, the number of basic operations is $c(\frac{n}{4})^2$. Since there are 3 subproblems, the running time is the sum of the running time of each node, i.e. $c(\frac{3n}{4})^2$ or $\frac{3}{16}cn^2$

At level i, the running time is also the sum of the running time of each node at level i, i.e. $(\frac{3}{16})^i cn^2$

Running time

As the number of levels is $\log_4 n$, the summation of the running time of each level is

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{n}{16})^{\log_4(n-1)} cn^2 + cn^{\log_4 3} \\ &= cn^2 \left[1 + \frac{3}{16} + (\frac{3}{16})^2 + \cdots + (\frac{3}{16})^{log_4(n-1)}\right] + cn^{\log_4 3} \\ &= O(n^2) \end{aligned}$$

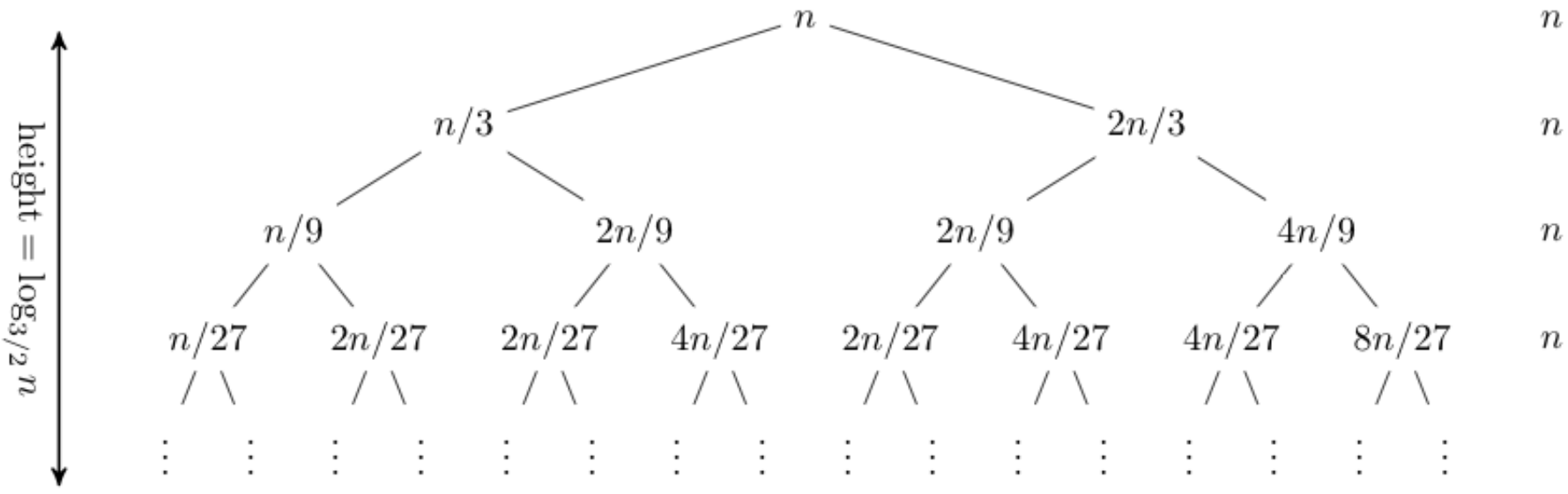
The last step is a decreasing geometric series (each terms multiplied by $\frac{1}{3}$), which converges to $\frac{16}{13}$

Exercise 2

Use the recursion tree method to find a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$

Exercise 3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/3) + T(2n/3) + n$.

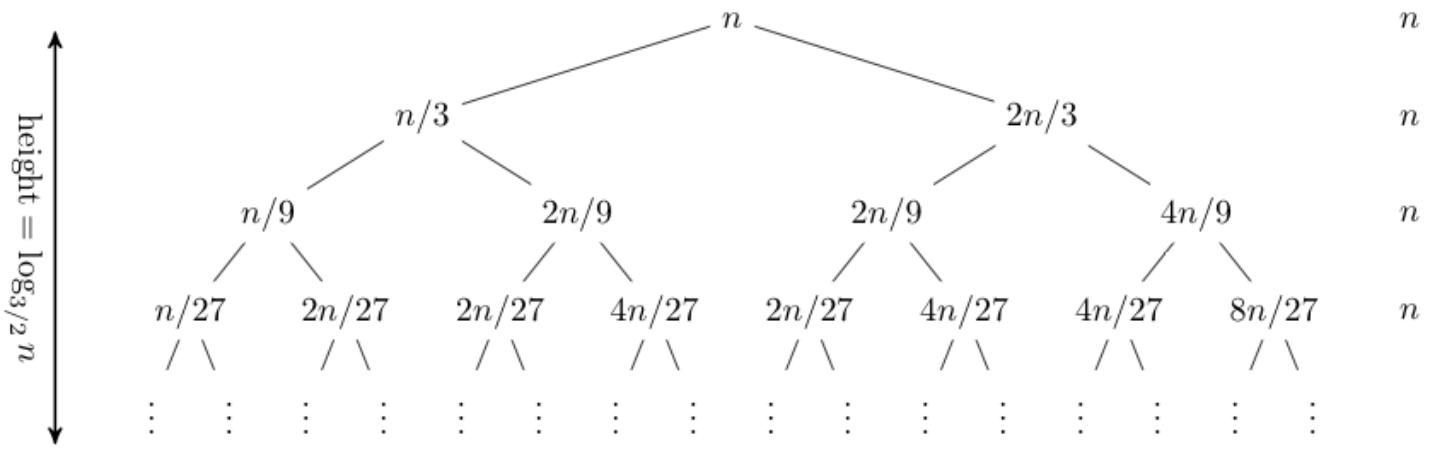


Each node at level i expands into 2 nodes at level $i + 1$

Exercise 3 : number of levels

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

.



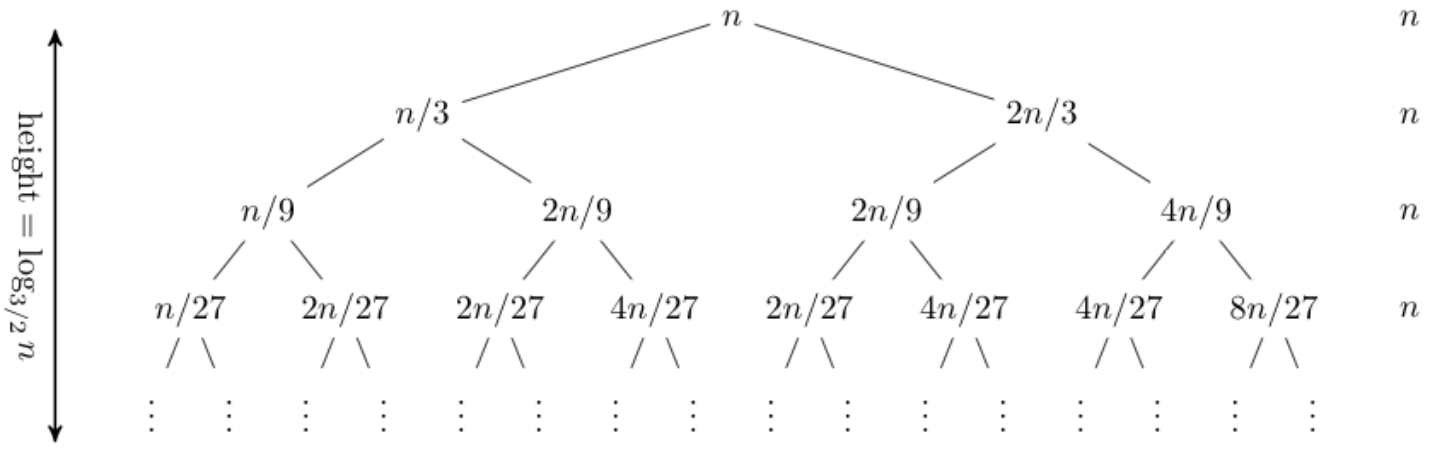
The input size of the node on the left is $\frac{n}{3}$, while the size of the other node is $\frac{3n}{2}$

The input size decreases much faster on the left than on the right side

Exercise 3 : number of levels

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

.



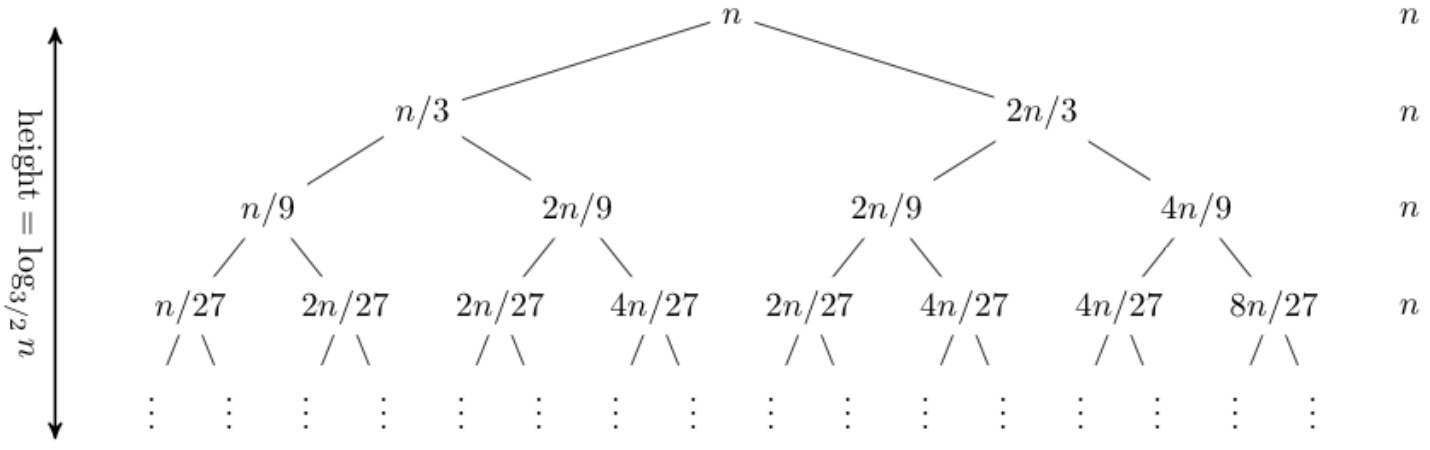
The number of levels of the leftmost path (the shortest path) is $\log_3 n$

The number of levels on the rightmost path (the longest one) is $\log_{3/2} n$ (as $\frac{2n}{3} = \frac{n}{3/2}$)

The number of levels in the other branches are a mix of these two extremes

Exercise 3 : running time

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



The longest branch of the tree has $\log_{3/2} n$ levels

At each level n basic operations are executed

$n \times \log_{3/2} n$ is in $O(n \log n)$ as logs from different bases differ only by a constant multiplicative factor

The Master Theorem

The Master Theorem provides closed forms to recurrences of the form :

$$T(n) = aT(\frac{n}{b}) + cn^m$$

where constants $a \geq 1, b > 1, m \geq 0$.

Examples of recurrence relations

$$T(n) = aT(\frac{n}{b}) + cn^m$$

$a \geq 1, b > 1, m \geq 0$

1. $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$. Here $a = 1, b = 2, m = 0$.
2. $T(n) = 2T(n/2) + n$. Here $a = 2, b = 2, m = 1$.
3. $T(n) = T(\sqrt{n}) + n$. Recurrence does not satisfies de conditions for master theorem, b is undefined
4. $T(n) = T(n/3) + T(2n/3) + n$. Recurrence does not satisfies conditions for master theorem. Two recursive terms, each with a different b : 1- $b = 3$ and 2- $b = 3/2$

The Master Theorem

The Master Theorem provides closed forms to recurrences of the form :

$$T(n) = aT(\frac{n}{b}) + cn^m$$

A solution to a recurrence of the above form (its closed form) is determined by the values of the constants a, b and m

$$T(n) \in \begin{cases} \Theta(n^m) & \texttt{if } a < b^m; \\ \Theta(n^m \log n) & \texttt{if } a = b^m; \\ \Theta(n^{\log_b a}) & \texttt{if } a > b^m. \end{cases}$$

This formulation is sometime refereed as the "restricted" form of the Master Theorem

Example 1

Give the exact order of the recurrence

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

Here $a = 2$, $b = 2$ and $m = 1$.

We have $a = 2 = b^m = 2^1 = 2$, therefore the case that applies is $a = b^m$.

Consequently, $T(n) \in \Theta(n^m \log n) = \Theta(n \log n)$.

Example 2

Give the exact order of the recurrence

$$T(n) = 6\,T(\frac{n}{4}) + n^2$$

Here $a = 6$, $b = 4$ and $m = 2$.

We have $a = 6 < b^m = 4^2 = 16$, therefore the case that applies is $a < b^m$.

Consequently, $T(n) \in \Theta(n^m) = \Theta(n^2)$.

Example 3

Give the exact order of the recurrence

$$T(n) = 7T(\frac{n}{2}) + n^2$$

Here $a = 7$, $b = 2, c = 1$ and $m = 2$.

We have $a = 7 > b^m = 2^2 = 4$, therefore the case that applies is $a > b^m$.

Consequently, $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$.

General form of the Master Theorem

General Master Theorem : Let constants $a \geq 1, b > 1$, and ϵ be a strictly positive real number. Assume that $T(n)$ satisfies the recurrence relation

$$T(n) = aT(\frac{n}{b}) + f(n),$$

Then $T(n)$ has the following asymptotic bounds :

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) \in O(n^{\log_b a - \epsilon}); \\ \Theta(f(n) \log n) & \text{if } f(n) \in \Theta(n^{\log_b a}); \\ \Theta(f(n)) & \text{if } f(n) \in \Omega(n^{\log_b a + \epsilon}) \text{ and } af(n/b) \leq cf(n) \\ & \text{for some } c < 1 \text{ and } n \text{ large enough.} \end{cases}$$

General vs restricted Master Theorem

This form of the Master Theorem is called "general" because $f(n)$ can be any function, it is not restricted to n^m.

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) \in O(n^{\log_b a - \epsilon}); \\ \Theta(f(n) \log n) & \text{if } f(n) \in \Theta(n^{\log_b a}); \\ \Theta(f(n)) & \text{if } f(n) \in \Omega(n^{\log_b a + \epsilon}) \text{ and } af(n/b) \leq cf(n) \\ & \text{for some } c < 1 \text{ and } n \text{ large enough.} \end{cases}$$

The general form can be of course applied to recurrences in the form $T(n) = aT(\frac{n}{b}) + cn^m$ but with the restricted form it easier to identify the exact order of a recurrence

$$T(n) \in \begin{cases} \Theta(n^m) & \text{if } a < b^m; \\ \Theta(n^m \log n) & \text{if } a = b^m; \\ \Theta(n^{\log_b a}) & \text{if } a > b^m. \end{cases}$$

Example 4

Use the general Master Theorem to give the exact order of the recurrence

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

Here $a = 2$, $b = 2$ and $m = 1$, and $f(n) = n$.

Is $n \in O(n^{\log_b a - \epsilon}) = O(n^{\log_2 2 - \epsilon}) = O(n^{1-\epsilon})$. No

Is $n \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$. Yes.

Therefore the case $T(n) \in \Theta(f(n) \log n)$ applies, $T(n) \in \Theta(n \log n)$

Example 5

Use the general Master Theorem for the recurrence

$$T(n) = 6T(\frac{n}{4}) + n^2$$

Here $a = 6$, $b = 4$ and $m = 2$, and $f(n) = n^2$.

Is $n^2 \in O(n^{\log_b a - \epsilon}) = O(n^{\log_4 6 - \epsilon}) = O(n^{\approx 1.29 - \epsilon})$. No

Is $n^2 \in \Theta(n^{\log_b a}) = \Theta(n^{\log_4 6}) = \Theta(n^{\approx 1.29})$. No

Is $n^2 \in \Omega(n^{\log_b a}) = \Omega(n^{\log_4 6 + \epsilon})$. Yes. Does $af(n/b) < cf(n) = 6(\frac{n}{4})^2 < .9n^2$, it is true for any value of $n \geq 4$ and $c = .9$.

Therefore case $\Theta(f(n))$ applies, $T(n) \in \Theta(n^2)$

Example 6

Use the general Master Theorem to give the exact order of the following recurrence

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

Here $f(n) = \log\log n$. $f(n) \in O(n^{\log_2 2-\epsilon}) = O(n^{1-\epsilon})$, because $\log\log n < \log n$, also $\log n$ grow slower than n^c for any value of $c > 0$, therefore $\log\log n \in O(n^{1-\epsilon})$

$f(n) \notin \Omega(n^{\log_b a+\epsilon})$ as we cannot bound $\log\log n$ below with $n^{1.\epsilon}$, therefore $f(n) \notin \Theta(n^{\log_2 2})$.

Therefore, as the case $f(n) \in O(n^{\log_2 2-\epsilon})$ applies, $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

Example 7

Use the general Master Theorem to give the exact order of the following recurrence

$$T(n) = 2T(n/4) + \sqrt{n}$$

Here $f(n) = \sqrt{n}$. Note that $\sqrt{n} = \sqrt[2]{n^1} = n^{\frac{1}{2}}$

Here $a = 2$ and $b = 4$. We want to know in which order $f(n) = \sqrt{n}$ is with respect to $n^{\log_b a} = n^{\log_4 2}$

It does happen that $n^{\log_4 2} = n^{\frac{1}{2}}$ since $\log_4 2 = \frac{\log_2 2}{\log_2 4} = \frac{1}{2}$

Therefore, $\sqrt{n} \in n^{\log_4 2}$ and $n^{\log_4 2} \in \sqrt{n}$, thus $\sqrt{n} \in \Theta(n^{\log_4 2})$

We conclude that $T(n) = \Theta(\sqrt{n} \log n)$

Exercises

1. Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers

a) $T(n) = 2T(n/2) + n^4$
b) $T(n) = T(7n/10) + n$
c) $T(n) = 16T(n/4) + n^2$
d) $T(n) = 7T(n/3) + n^2$
e) $T(n) = 7T(n/2) + n^2$
f) $T(n) = 3T(n/4) + \sqrt{n}$
g) $T(n) = T(n-2) + n^2$
h) $T(n) = 4T(n/3) + n \log n$
i) $T(n) = 3T(n/3) + \frac{n}{\log n}$
k) $T(n) = 4T(n/2) + n^2\sqrt{n}$
l) $T(n) = 3T(n/3 - 2) + \frac{n}{2}$
m) $T(n) = 2T(n/2) + \frac{n}{\log n}$
n) $T(n) = \sqrt{n}T(\sqrt{n}) + n$
o) $T(n) = T(n-2) + \log n$

Exercises : solutions

a) $T(n) = 2T(n/2) + n^4$. Solution $\Theta(n^4)$

b) $T(n) = T(7n/10) + n$. Solution, here $a = 1, b = \frac{10}{7}, m = 1$. Case $a < b^m$ applies, $T(n) \in \Theta(n)$

c) $T(n) = 16T(n/4) + n^2$. Solution, $a = b^m$ $T(n) \in \Theta(n^2 \log n)$

d) $T(n) = 7T(n/3) + n^2$. Solution, here $a = 7, b = 3, m = 2$. Case $a < b^m$ applies, $T(n) \in \Theta(n^2)$

e) $T(n) = 7T(n/2) + n^2$. Solution, here $a > b^m$, case $T(n) \in \Theta(n^{\log_n a})$ applies $T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.803})$

f) $T(n) = 3T(n/4) + \sqrt{n}$. Here $\sqrt{n} = n^{.5}$, $n^{\log_b a} = n^{\log_4 3} = n^{0.79}$. Therefore case $f(n) \in O(n^{\log_b a - \epsilon}) = n^{0.5} \in O(n^{0.79-\epsilon})$ applies, $T(n) \in \Theta(n^{0.79})$

g) $T(n) = T(n-2) + n^2$. Master Theorem does not apply, no b

Exercises : solutions

h) $T(n) = 4T(n/3) + n \log n$. Here $n^{\log_b a} = n^{\log_3 4} = n^{1.27}$. $n^{1.27} = n \times n^{.27}$. We already know that $\log n < n^{\epsilon > 0}$ therefore $f(n) = n \log n \in O(n^{1.27-\epsilon})$. Case $f(n) \in O(n^{\log_b a - \epsilon}) = n \log n \in O(n^{1.27-\epsilon})$ applies, $T(n) \in \Theta(n^{1.27})$

i) $T(n) = 3T(n/3) + \frac{n}{\log n}$. Master Theorem does not apply as $\frac{n}{\log n}$ does not grow polynomially

k) $T(n) = 4T(n/2) + n^2\sqrt{n}$. Solution, $n^2\sqrt{n} = n^2 \times n^{0.5} = n^{2.5}$. $a = 4, b = 2, m = 2.5$, $a < b^m$, case $T(n) \in \Theta(n^m)$ applies, therefore $T(n) \in \Theta(n^{2.5})$.

l) $T(n) = 3T(n/3 - 2) + \frac{n}{2}$. Master Theorem does not apply because $b \not> 1$

m) $T(n) = 2T(n/2) + \frac{n}{\log n}$. Same as i)

n) $T(n) = \sqrt{n}T(\sqrt{n}) + n$. Master Theorem does not apply because no b

o) $T(n) = T(n-2) + \log n$. Same as n)

Execution sequence of recursive factorial

First we describe the execution sequence of the simple recursive algorithm factorial

Then describe the less basic execution sequence of merge sort

```
factorial (n)
→if (n ≤ 1)
      return 1;
   else
      return n × factorial(n − 1);
```

Execution sequence of recursive factorial

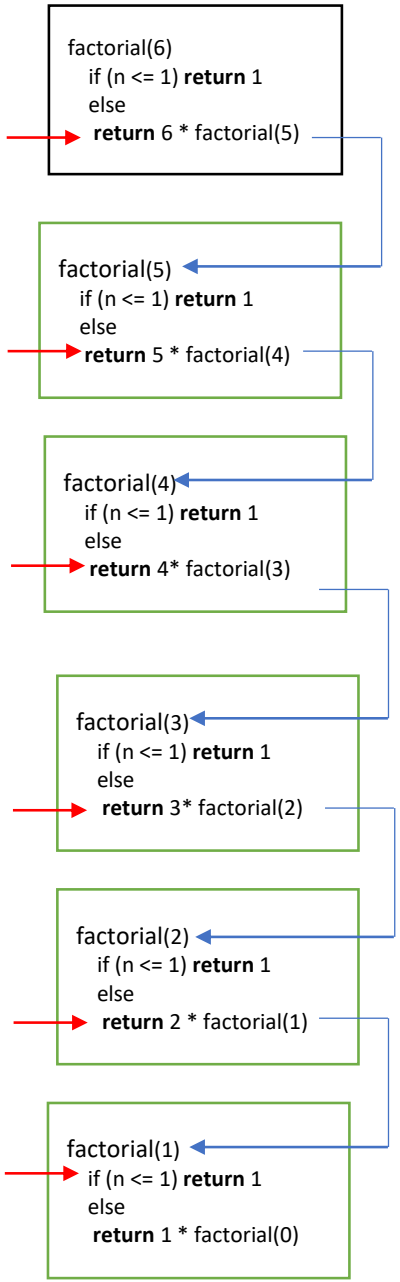
```
factorial (n)
   if (n ≤ 1)
      return 1 ;
   else
→     return n × factorial(n − 1) ;
```

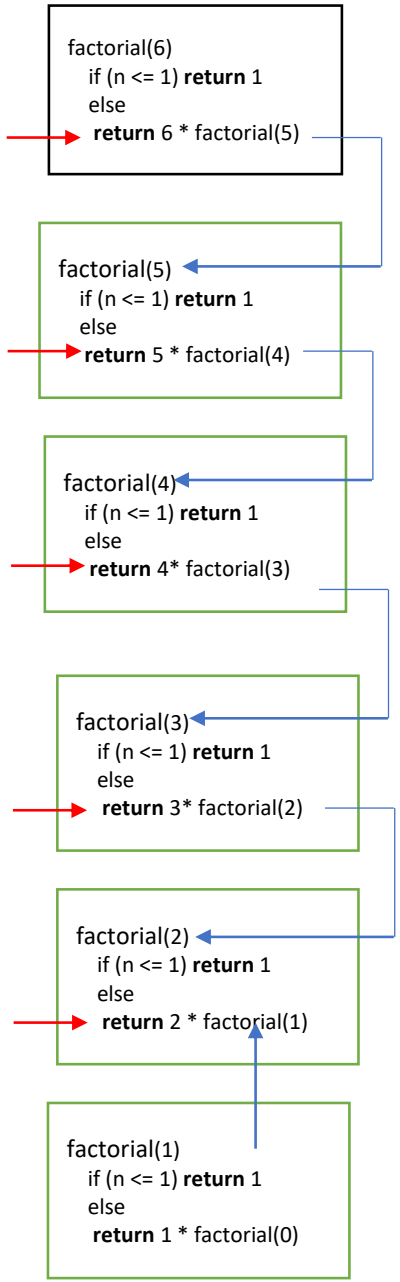
Execution sequence of recursive factorial

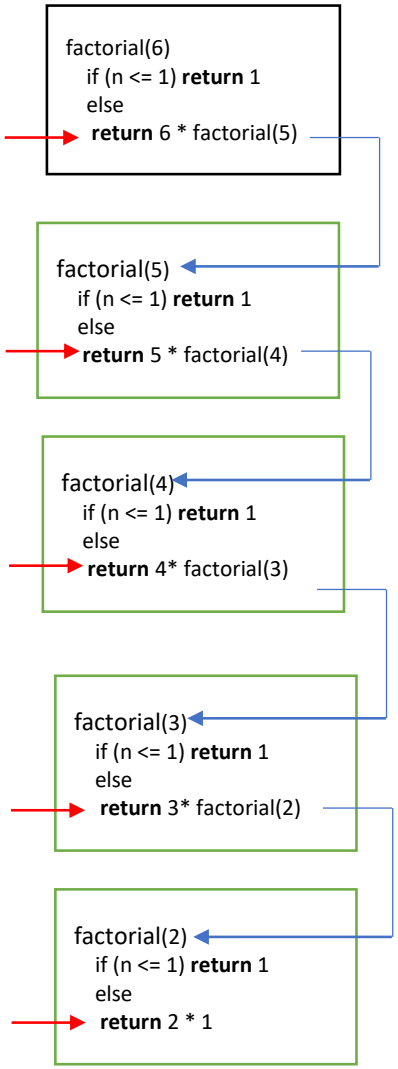
factorial $(n-1)$
→**if** $(n \leq 1)$
    return 1 ;
  else
    return $n \times$ factorial$(n-1)$;

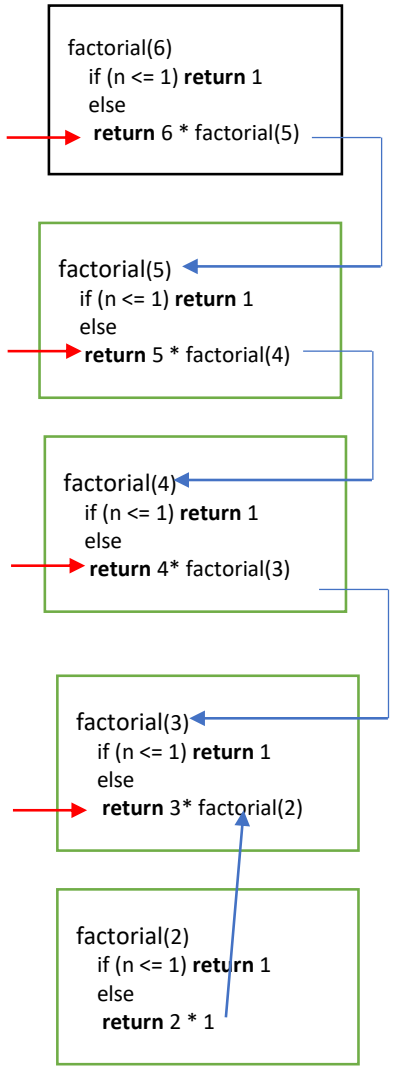
Execution sequence of recursive factorial

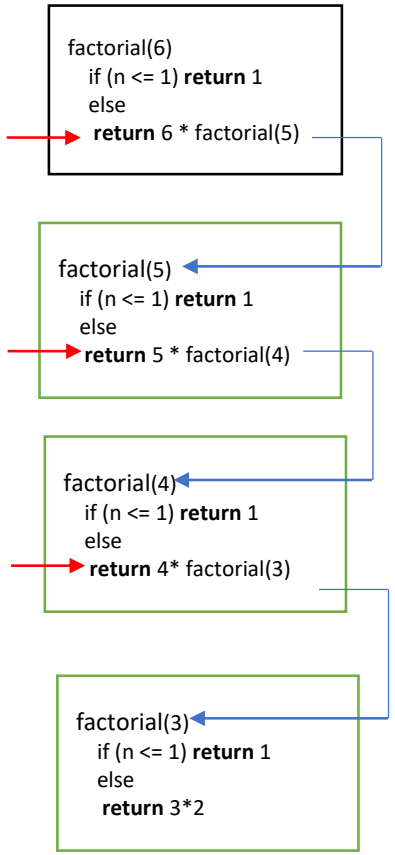
```
factorial (n − 1)
   if (n ≤ 1)
      return 1 ;
   else
→     return n × factorial(n − 1) ;
```

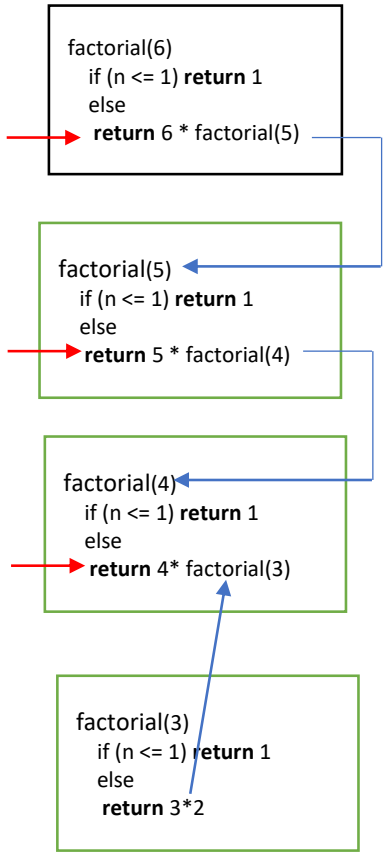

factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3* factorial(2)
factorial(2)
if (n <= 1) return 1
else
return 2 * factorial(1)
factorial(1)
if (n <= 1) return 1
else
return 1 * factorial(0)

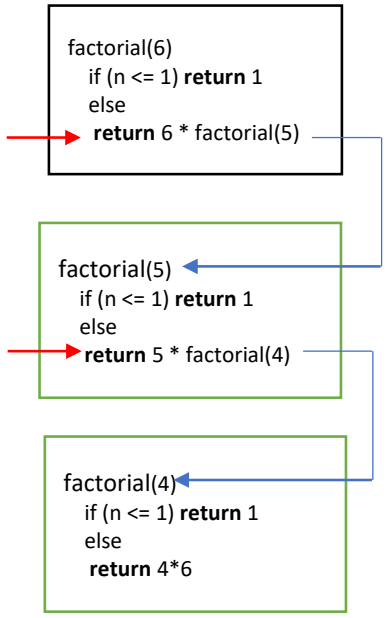

factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3* factorial(2)
factorial(2)
if (n <= 1) return 1
else
return 2 * factorial(1)
factorial(1)
if (n <= 1) return 1
else
return 1 * factorial(0)


factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3* factorial(2)
factorial(2)
if (n <= 1) return 1
else
return 2 * 1


factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3* factorial(2)
factorial(2)
if (n <= 1) return 1
else
return 2 * 1


factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3*2


factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4* factorial(3)
factorial(3)
if (n <= 1) return 1
else
return 3*2


factorial(6)
if (n <= 1) return 1
else
return 6 * factorial(5)
factorial(5)
if (n <= 1) return 1
else
return 5 * factorial(4)
factorial(4)
if (n <= 1) return 1
else
return 4*6

Merge Sort

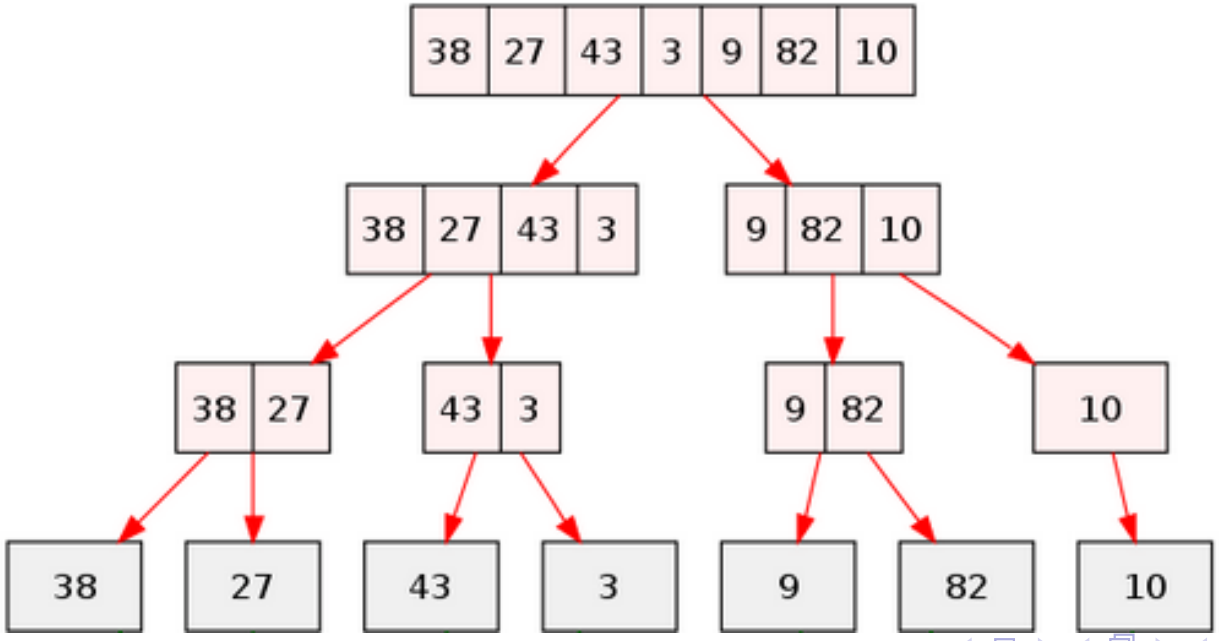
Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Mergesort does the following :

- **Divide** the array in half.
- **Recursively** sort each half.
- **Merge** the two sorted halves.

Merge Sort : the intuition

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



Merge Sort : execution sequence

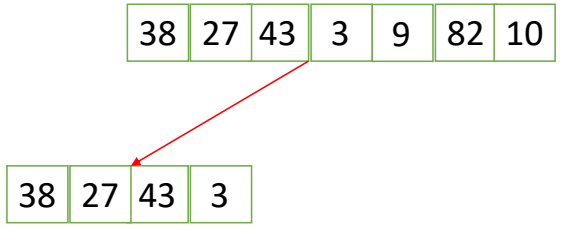
Mergesort($A[p..r]$)
→**if** $p < r$
$q = \lfloor \frac{p+r}{2} \rfloor$
Mergesort($A[p..q]$)
Mergesort($A[q+1..r]$)
Merge(A, p, q, r)

Merge Sort : execution sequence

Mergesort($A[p..r]$)
  if $p < r$
→ $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

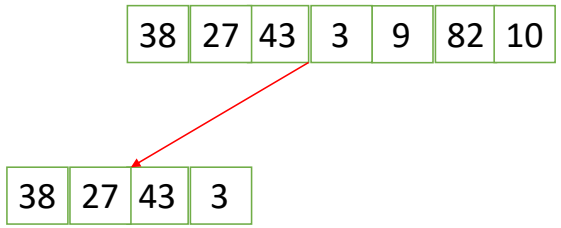
Merge Sort : execution sequence

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
→ Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

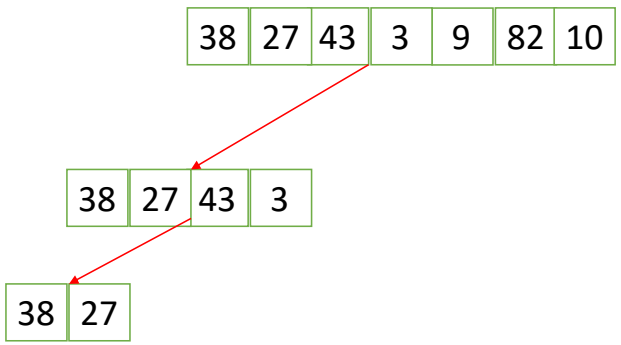


Merge Sort : execution sequence

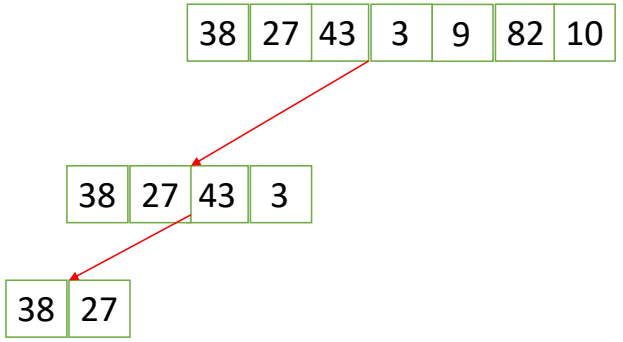
Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
→ Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



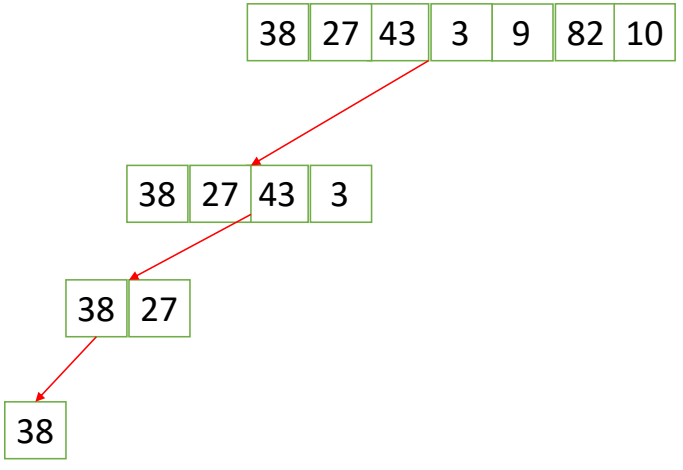
Mergesort($A[1..4]$)
  if $1 < 4$
    $q = \lfloor \frac{1+4}{2} \rfloor$
→ Mergesort($A[1..2]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



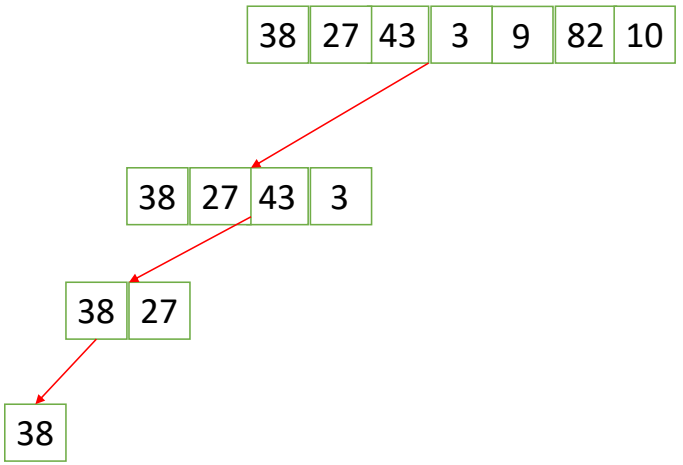
Mergesort($A[1..4]$)
  if $1 < 4$
    $q = \lfloor \frac{1+4}{2} \rfloor$
→ Mergesort($A[1..2]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



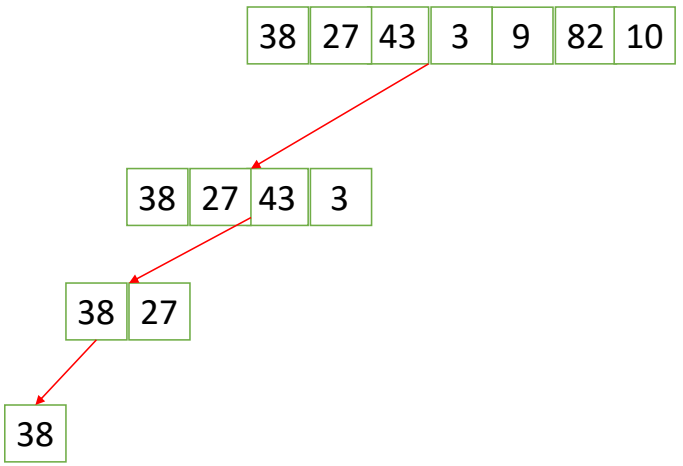
Mergesort($A[1..2]$)
  if $1 < 2$
    $q = \lfloor \frac{1+2}{2} \rfloor$
→ Mergesort($A[1..1]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



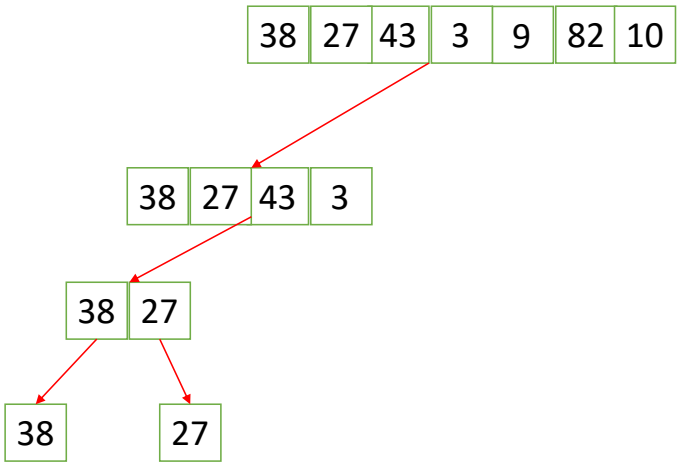
Mergesort($A[1..1]$)
→**if** $1 < 1$
$q = \lfloor \frac{1+4}{2} \rfloor$
Mergesort($A[1..2]$)
Mergesort($A[q+1..r]$)
Merge(A, p, q, r)



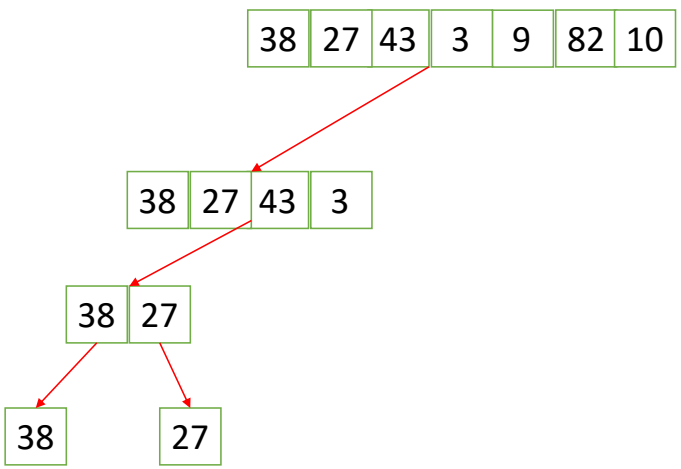
Mergesort($A[1..2]$)
if $1 < 2$
$q = \lfloor \frac{1+2}{2} \rfloor$
→ Mergesort($A[1..1]$)
Mergesort($A[q+1..r]$)
Merge(A, p, q, r)



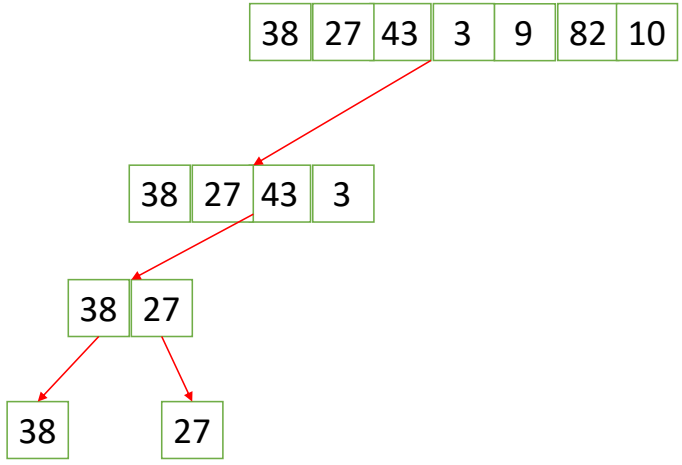
Mergesort($A[1..2]$)
  if $1 < 2$
    $q = \lfloor \frac{1+2}{2} \rfloor$
    Mergesort($A[1..1]$)
→ Mergesort($A[2..2]$)
    Merge(A, p, q, r)



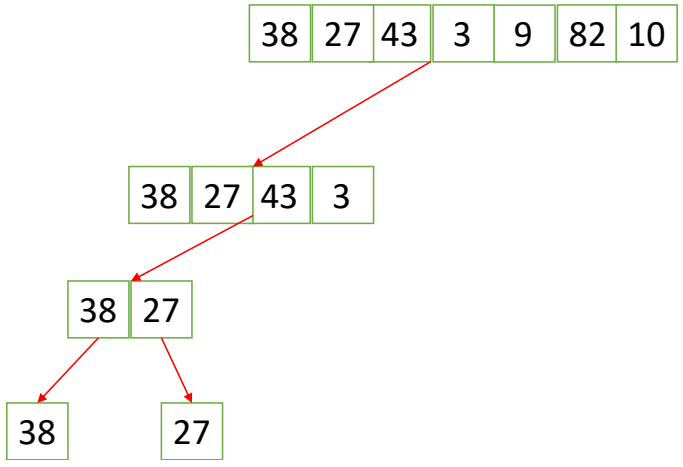
Mergesort($A[2..2]$)
→**if** $2 < 2$
    $q = \lfloor \frac{1+4}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[2..2]$)
    Merge(A, p, q, r)



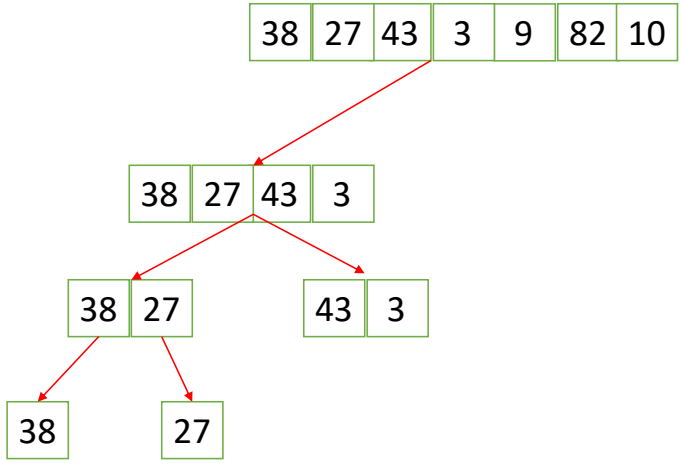
Mergesort($A[1..2]$)
  if $1 < 2$
    $q = \lfloor \frac{1+2}{2} \rfloor$
    Mergesort($A[1..1]$)
    Mergesort($A[2..2]$)
→ Merge(A, p, q, r)



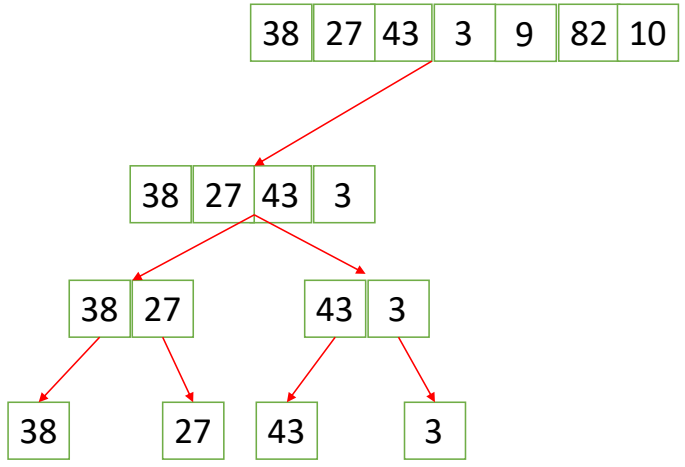
Mergesort($A[1..4]$)
  if $1 < 4$
    $q = \lfloor \frac{1+4}{2} \rfloor$
→ Mergesort($A[1..2]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)



Mergesort($A[1..4]$)
  if $1 < 4$
    $q = \lfloor \frac{1+4}{2} \rfloor$
    Mergesort($A[1..2]$)
→ Mergesort($A[3..4]$)
    Merge(A, p, q, r)



Mergesort(A[3..4]) executes the same sequence of recursive calls as for Mergesort(A[1..2])
Then it returns to Mergesort(A[1..4]) to execute the merge function, and then returns to Mergesort(A[1..7]) which is the initial call



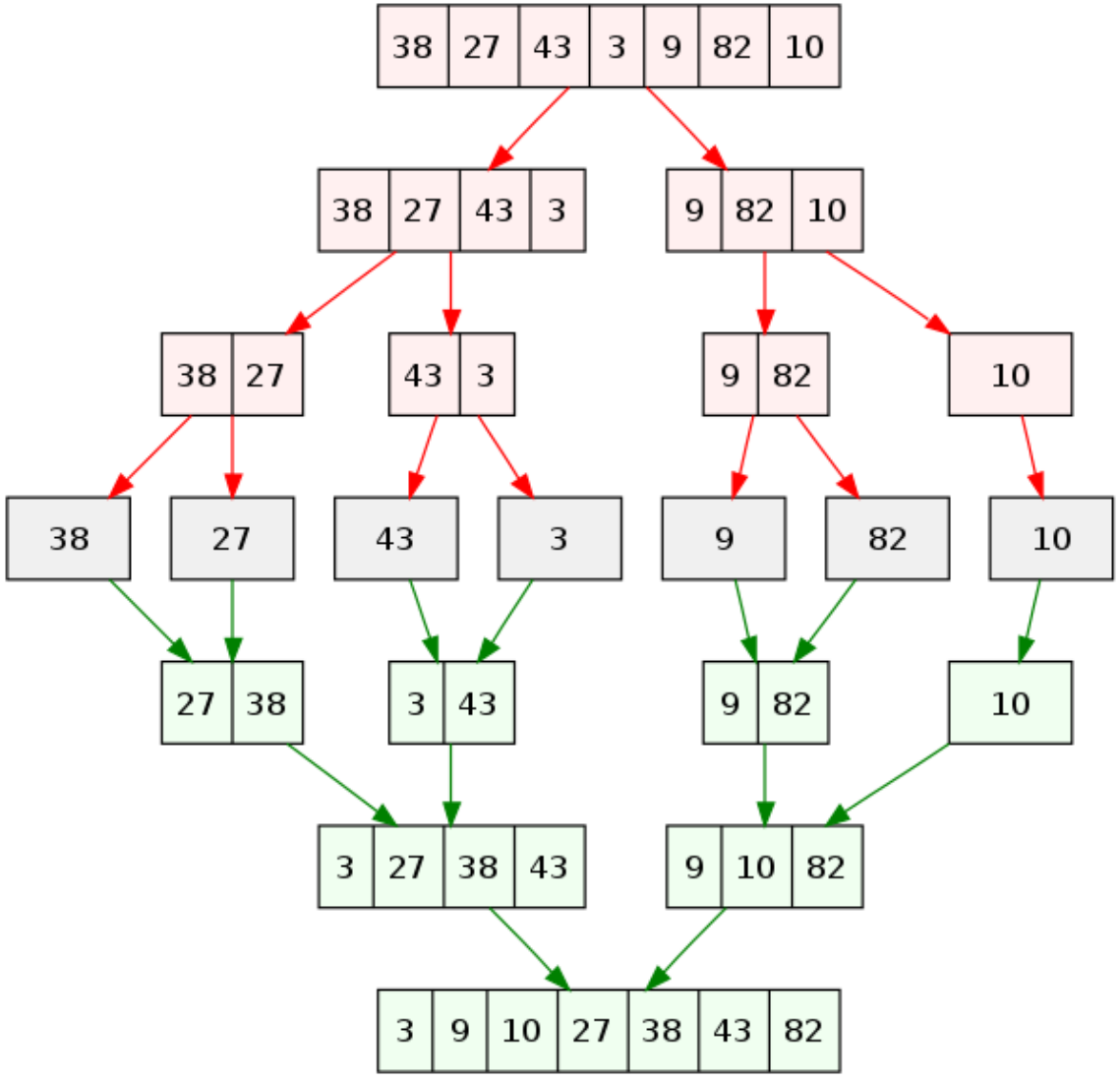
Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
→ Mergesort($A[1..4]$)
    Mergesort($A[5..7]$)
    Merge(A, p, q, r)

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[1..4]$)
→ Mergesort($A[5..7]$)
    Merge(A, p, q, r)

The divide and merge parts !

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
→   Merge(A, p, q, r)

The merge part takes 2 sorted arrays and merges them into a single array sorted in increasing order



Merge algorithm

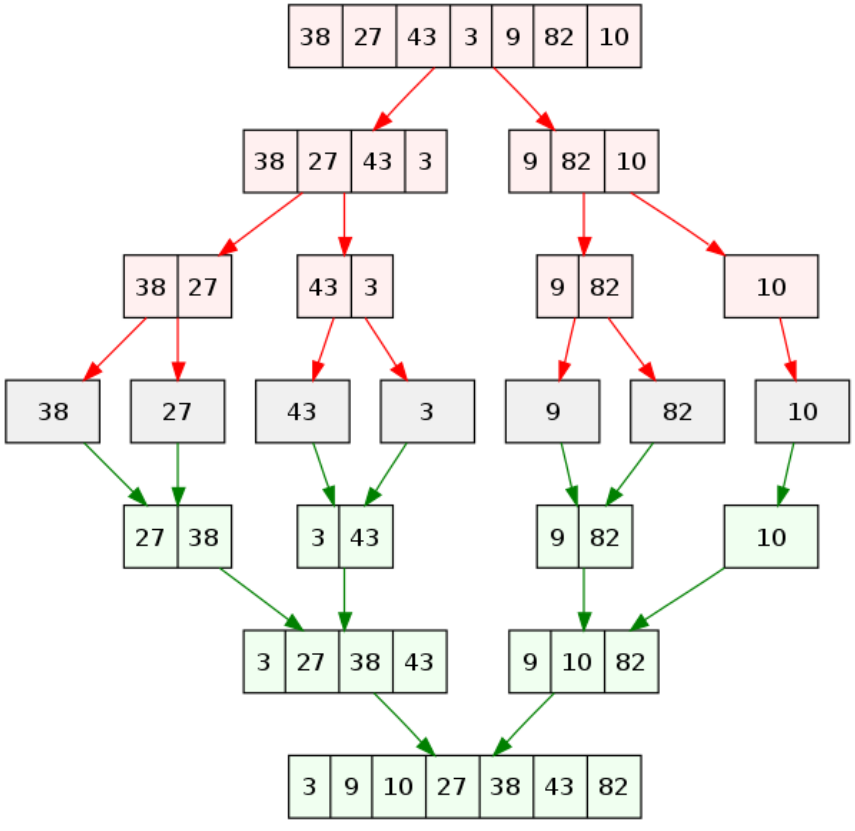
$Merge(A, p, q, r)$
$n1 = q - p + 1$
$n2 = r - q$
$L[1..n1 + 1]$, $R[1..n2 + 1]$
for $i = 1$ **to** $n1$
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** $n2$
  $R[j] = A[q + j]$
$L[n1 + 1] = -\infty$
$R[n2 + 1] = -\infty$
$i = 1, j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[j]$
    $j = j + 1$

Works as follows :

- Has a pointer to the beginning of each subarray.
- Put the smaller of the elements pointed to in the new array.
- Move the appropriate pointer.
- Repeat until new array is full.

Merge algorithm

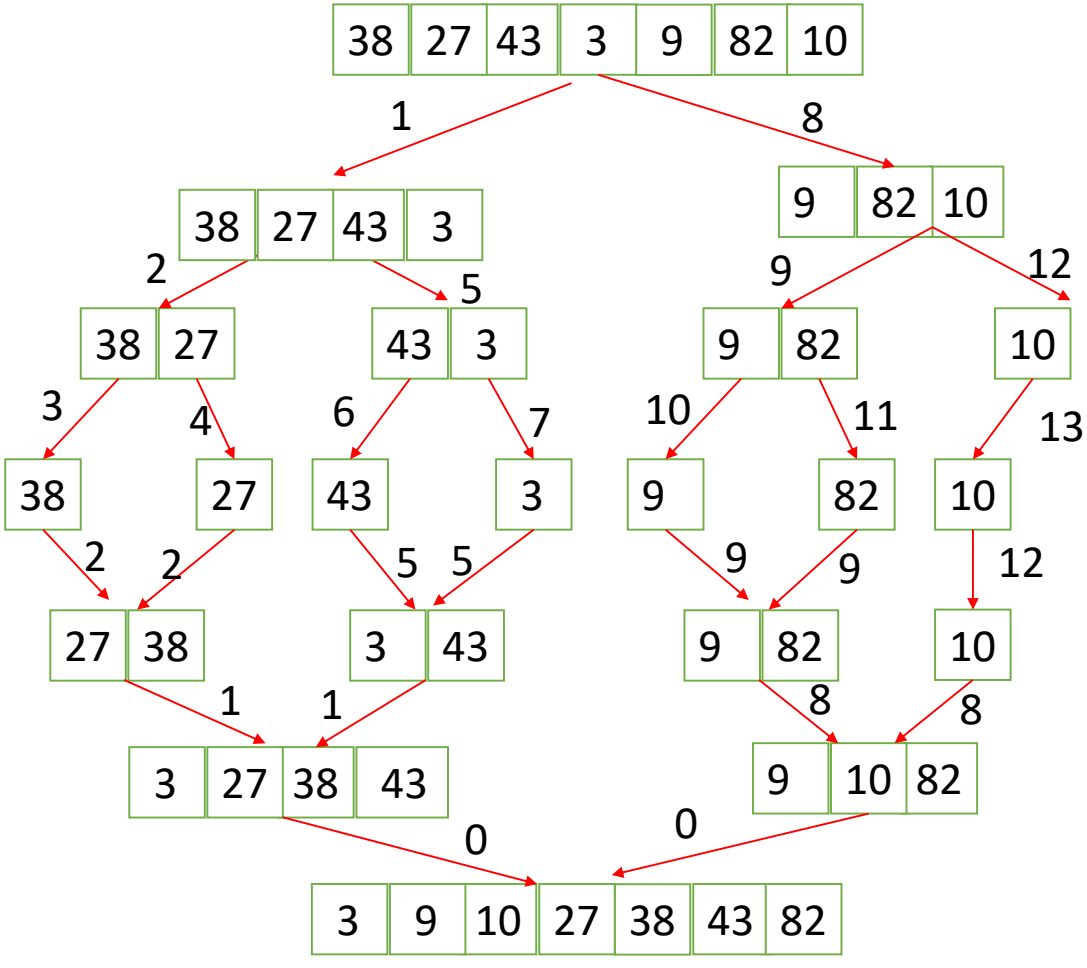
$Merge(A, p, q, r)$
$n1 = q - p + 1$
$n2 = r - q$
$L[1..n1 + 1]$, $R[1..n2 + 1]$
for $i = 1$ **to** $n1$
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** $n2$
  $R[j] = A[q + j]$
$L[n1 + 1] = -\infty$
$R[n2 + 1] = -\infty$
$i = 1, j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[j]$
    $j = j + 1$



Sequence of recursive calls

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
L- Mergesort($A[p..q]$)
R- Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

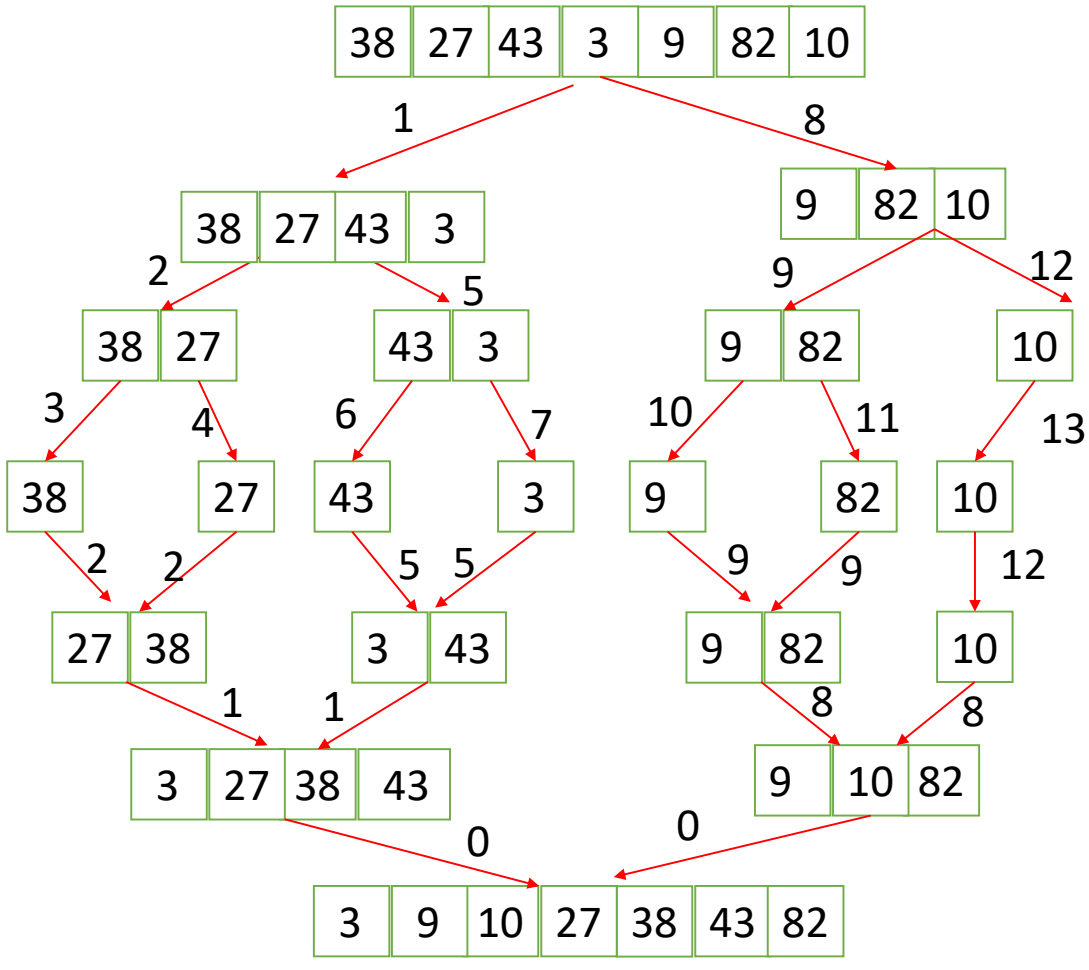
1 L-Mergesort(A[1..4])
2 L-Mergesort(A[1..2])
3 L-Mergesort(A[1..1])
4 R-Mergesort(A[2..2])
2 Merge(A,1,1,2)
5 R-Mergesort(A[3..4])
6 L-Mergesort(A[3..3])
7 R-Mergesort(A[4..4])
5 Merge(A,3,3,4)
1 Merge(A,1,2,4)



Sequence of recursive calls

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
L- Mergesort($A[p..q]$)
R- Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

8 R-Mergesort(A[5..7])
9 L-Mergesort(A[5..6])
10 L-Mergesort(A[5..5])
11 R-Mergesort(A[6..6])
9 Merge(A,5,5,6)
12 R-Mergesort(A[7..7])
13 L-Mergesort(A[7..7])
12 Merge(A,7,7,7)
8 Merge(A,5,6,7)
0 Merge(A,1,4,7)



Running time of merge sort

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

The cost of each "Merge" is $O(n)$, as the procedure has to scan two subarrays.

The size of the problem instance is reduced by half each time the recursive function is called. So the recurrence relation is as followed :

$$T(n) = 2T(\frac{n}{2}) + cn.$$

This recurrence can be solved using the Master Theorem : $a = 2$, $b = 2$ and $m = 1$, $a = b^m$, $T(n) \in \Theta(n^m \log n) = \Theta(n \log n)$.

A few relations to remember

$\log_a(xy) = \log_a x + \log_a y$
$\log_a \frac{x}{y} = \log_a x - \log_a y$
$\log_a(x^r) = r \log_a x$
$\log_a a = 1; \log_a 1 = 0$
$\log_a \frac{1}{x} = -\log_a x$
$\log_a x = \frac{\log x}{\log a}$
$\log_b x = \frac{log_2 x}{\log_2 b}$ changing to log base b
$x^{\frac{a}{b}} = \sqrt[b]{x^a}$
$x^{1/b} = z$ means $z^b = x$; $x^{a/b} = z$ means $z^b = x^a$
$c^{\log(a)} = a^{\log(c)}$: take log of both sides.
$(b^a)^c = b^{ac}$
$b^a b^c = b^{a+c}$
$b^a / b^c = b^{a-c}$