**RECURSION**

Recursive Thinking

Consider searching for a target value in an array

* Assume the array elements are sorted in increasing order
* We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
* Instead of searching n elements, we search n/2 elements

Text, letter

Description automatically generated

Steps to Design a Recursive Algorithm

There must be at least one case (the base case), for a small value of n, that can be solved directly

A problem of a given size n can be reduced to one or more smaller versions of the same problem (recursive case(s))

Identify the base case(s) and solve it/them directly

Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case

Combine the solutions to the smaller problems to solve the larger problem

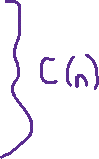
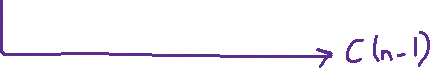
Recursive Algorithm for Finding the Length of a String

Graphical user interface, text, application, email

Description automatically generated

Text, letter

Description automatically generated



T(n) = θ(1) if n = 0

Recurrence relation for the running time of the algorithm

T(n) = T(n-1) + θ(n) if n > 0

Solve this recurrence relation:

* The operation that is done the most frequently is substring operation  
  There are n-1 copy operation in 1 recursive call so it is most frequent operation
* C(n) : number of copy operations
  + C(n) = 0 if n = 0

Recurrence relation for the number of copy operations

* + C(n) = C(n-1) + n-1 if n > 0
    - C(n) = n-1 + n-2 + n-3 + … + 2 + 1 = ( n(n+1) )/ 2 = θ()
* T(n) = θ() as well bc running time depends on these copy operations bc copy operations are the most frequent operations

T(n) is number of operations.

Diagram

Description automatically generated

T(n) is addition of all these terms.

T(n) = θ() = θ()

If substring method is constant time:

T(n) = = θ() = θ(n)

Another way to do it (substitution method):

* T(n) = T(n-1) + θ(n)
* T(n) = T(n-2) + θ(n-1) + θ(n)
* T(n) = T(n-3) + θ(n-2) + θ(n-1) + θ(n)
* …
* T(n) = T(0) + θ(1) + θ(2) + … + θ(n-1) + θ(n)
* T(n) = θ()

Recursive Algorithm for Printing String Characters

Graphical user interface, text

Description automatically generated

If we change order of last 2 lines, then string will be printed reverse.

Similar to previous example, substring method makes this method quadratic.

T(n) = θ(1) if n = 0

T(n) = T(n-1) + θ(n) if n > 0

Proving that a Recursive Method is Correct

Proof by induction

* Prove the theorem is true for the base case
* Show that if the theorem is assumed true for n, then it must be true for n+1

Recursive proof is similar to induction

* Verify the base case is recognized and solved correctly
* Verify that each recursive case makes progress towards the base case
* Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Diagram

Description automatically generated with medium confidenceTracing a Recursive Method

The process of returning from recursive calls and computing the partial results is called unwinding the recursion

Run-Time Stack and Activation Frames

Java maintains a run-time stack on which it saves new information in the form of an activation frame

The activation frame contains storage for:

* method arguments
* local variables (if any)
* the return address of the instruction that called the method

Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack

A picture containing graphical user interface

Description automatically generated

Text

Description automatically generatedText, letter

Description automatically generated

Analysis of Recursive Algorithms

General plan for analyzing efficiency of recursive algorithm :

* Decide on parameters indicating input’s size
  + decide what is “n”
* Calculate the number of operations executed at each recursive call
  + can vary on different inputs of the same size
    - worst-case, average-case, and best-case efficiencies must be investigated separately
* Set up a recurrence relation for the number of operations executed
  + with appropriate initial condition
* Solve the recurrence
  + at least ascertain the order of growth of its solution

Recursive Definitions of Mathematical Formulas

Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms

Examples:

* factorials
* powers
* greatest common divisors (gcd)

Factorial of n: n!

0! = 1 n = 0

n! = n x (n-1)! n > 0

Diagram, text

Description automatically generated with medium confidence

Multiplication is the basic (most frequent) operation in the algorithm. It determines the running time.

M(n) : number of multiplications for the factorial(n)

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

M(0) = 0

* M(n) = M(n-1) + 1
* M(n) = M(n-2) + 2
* M(n) = M(n-3) + 3
* …
* M(n) = M(n-k) + k
* k = n :
* M(n) = M(0) + n = n -----> number of multiplications in this algorithm

T(n) = θ(n)

Running time can be expressed as asymptotic notation while number of multiplications can be expressed exactly as a function.

Saying running time exactly is not easy bc recursive calls require forming an activation record, pushing it on stack, returning from factorial function require popping the stack and changing the problem counter, etc. We don’t know how many operations are there exactly. We don’t go into details and use asymptotic notation to consider algorithm only, not details.

Infinite Recursion and Stack Overflow

If you call method factorial with a negative argument, the recursion will not terminate bc n will never equal 0

If a program does not terminate, it will eventually throw the StackOverflowError exception

In the factorial method, you could throw an IllegalArgumentException if n is negative

Calculating x^n

Text

Description automatically generated

parameter : n

T(n) = T(n-1) + θ(1)

Calculating gcd

Text

Description automatically generated with low confidence

Text

Description automatically generated

Tb(n) = θ(1) 🡪 m % n = 0

Tw(n) = θ(logn)

T(n) = O(logn)

You cannot say exact running time in theta notation since running time depends on relation between m and n, not n only. Not only size of n but relation between m and n. n could be very large but still divides m and problem would be very simple.

After 2 recursive calls (neglect m < n case and base case), call would be 🡪 gcd(m%n, n%(m%n))

n%(m%n) is smaller than n/2. It reduces twice.

n ----> n/2 ----> n/4 ----> n/8 ----> … ----> 1

When n is 1, base case is always satisfied. It requires logn steps and each step 2 calls so 2logn recursive calls are necessary. We perform 2logn constant time operations (all operations in algorithm is constant).

Recursion vs. Iteration

You can always write an iterative solution to a problem that is solvable by recursion

A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Efficiency:

Recursive methods often have slower execution times relative to their iterative counterparts

The overhead for loop repetition is smaller than the overhead for a method call and return

If it’s easier to conceptualize an algorithm using recursion, then you should code it as recursive method

The reduction in efficience usually doesn’t outweigh advantage of readable code that is easy to debug

Fibonacci Numbers

Graphical user interface, text

Description automatically generated

T(n) = T(n-1) + T(n-2) + θ(1) if n = 2

T(n) = θ(1) if n <= 2

T(n) = O() (reason is next semester’s topic)

Diagram

Description automatically generated

Text

Description automatically generatedText

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

Method fibo is an example of tail recursion or last-line recursion

When recursive call is the last line of the method, arguments and local variable do not need to be saved in the activation frame

Recursive Array Search

Simplest 🡪 linear

On average, (n+1)/2 elements are examined to find the target in a linear search

T(n) = O(n)

Base cases for recursive search:

* empty array, target cannot be found; result is -1
* first element of the array being searched = target; result is the subscript of first element

The recursive step searches the rest of the array, excluding the first element

Text

Description automatically generated

Text

Description automatically generated

If we remove the first element that we look and send the remaining array, then running time will be a lot: T(n) = T(n-1) + n - 1 (n-1 element to copy) = θ()

Text

Description automatically generated

We can also do by changing the array but copying elements etc. takes a lot of time.

T(0) = θ(1)

Tb(n) = θ(1)

Tw(n) = Tw(n-1) + θ(1) = θ(n)

T(n) = O(n)

Diagram, text

Description automatically generated

Design of a Binary Search Algorithm

Only on sorted array

Base cases:

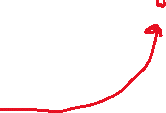
* The array is empty
* Element being examined matches the target

Text, letter

Description automatically generated



Tb(n) = θ(1) (element in the middle)



Tw(n) = θ(1) (if array is empty)

Tw(n) = Tw(n/2) + θ(1) = θ(logn) (element not found)  
 either one of these 2 will work so running time has 1 Tw(n/2)

* Tw(n) = T(n/2) + θ(1)
* Tw(n) = T(n/ + θ(1) + θ(1)
* Tw(n) = T(n/ + θ(1) + θ(1) + θ(1)
* …
* Tw(n) = T(n/ + kθ(1)
* :
* Tw(n) = T(1 + (logn)θ(1) = θ(logn)

At each recursive call, we eliminate half the array elements from consideration, making a binary search O(logn)

An array of 16 would search arrays of length 16, 8, 4, 2, and 1: 5 probes in the worst case

* 16 =
* 5 = + 1

A doubled array size would require only 6 probes in the worst case

* 32 =
* 6 = + 1

An array with 32768 elements require only 16 probes! 🡪 = 15

Comparable Interface

Classes that implement the Comparable interface must define a compareTo method

Method call obj1.compareTo(obj2) returns an integer with the following values:

* negative if obj1 < obj2
* zero if obj1 == obj2
* positive if obj1 > obj2

Implementing the Comparable interface is an efficient way to compare objects during search

Text

Description automatically generatedText, letter

Description automatically generated

You should test arrays with:

* an even number of elements
* an odd number of elements
* duplicate elements

Test each array for the following cases:

* the target is the element at each position of the array, starting with the first position and ending with the last position
* the target is less than the smallest array element
* the target is greater than the largest array element
* the target is a value between each pair of items in the array

Method Arrays.binarySearch

Java API class Arrays contains a binarySearch method:

* Called with sorted arrays of primitive types or with sorted arrays of objects
* If the objects in the array are not mutually comparable or if the array is not sorted, the results are undefined
* If there are multiple copies of the target value in the array, there is no guarantee which one will be found
* Throws ClassCastException if the target is not comparable to the array elements

**Recursive Data Structures**

The first language developed fo AI research was a recursive language called LISP (next year course)

Recursive Definition of a Linked List

We define a class LinkedListRec<E> that implements several list operations using recursive methods

Text

Description automatically generated

SAME

Graphical user interface, text, application

Description automatically generated

**If a method doesn’t have any parameter, you cannot talk about smaller problems. So it cannot be performed recursively. So we use helper method.**

n : size of the linked list

T(0) = θ(1)

T(n) = T(n-1) + θ(1) = θ(n)

At each recursive call, we keep a head value. Each call, there is a θ(1) space used in run-time stack.  
S(0) = θ(1)   
S(n) = S(n-1) + θ(1) = θ(n) {You can write this method iteratively by constant space.}

Graphical user interface, text, application

Description automatically generated



String concatenation is performed by copying since Strings are inmutable in Java.

So :

* T(0) = θ(1)
* T(n) = T(n-1) + θ(n)
* T(n) = θ()

You can use StringBuilder to get rid of θ(n) and make T(n) linear time.

Graphical user interface, text, application

Description automatically generated



T(n) = T(n-1) + θ(1) = θ(n)

Graphical user interface, text

Description automatically generated

T(n) = T(n-1) + θ(1) = θ(n)

Text

Description automatically generatedText

Description automatically generated

Sb(n) = θ(1) Sw(n) = Sw(n) + θ(1) = θ(n)

S(n) = O(n)

T(n) = T(n-1) + θ(1) 🡪 T(n) = O(n)

not θ(n) bc in the worst case we have to go until the end. Tb(n) = θ(1) , Tw(n) = θ(n)

Recursive Algorithm for Towers of Hanoi

Table

Description automatically generated with medium confidence

Diagram

Description automatically generated

n diski L’den R’ye taşıyacağız.

Önce n-1 diski L’den M’ye taşıyacağız.

En alttaki diski R’ye taşıyınca tekrardan n-1 diski M’den R’ye taşıyacağız.

In this recursive definition, how disk problem is gonna be solved is not necessary to worry about. We don’t need to worry about recursive calls. If recursive calls end with the base case, and if you can prove by induction that the algorithm solves the problem if 1. and 3. steps solve the smaller problem, you should be done. You don’t need to worry about how 1. and 3. steps (smaller problems) are solved. Induction handles that for you.

Diagram

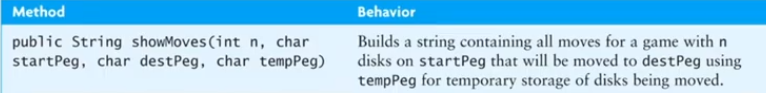
Description automatically generated

Diagram

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence



Text

Description automatically generated



You can’t just say “T(n) = 2T(n-1) + θ(1)” because there are string concatenations. You cant say θ(n) instead of θ(1) too because it is more than linear time. For n, number of moves is exponential. So it is like θ(), 🡪 T(n) = 2T(n-1) + θ() for concatenation.

M(n) : number of moves

* M(n) = 1 if n = 1
* M(n) = 2M(n-1) + 1 if n > 1
  + M(n) = 2(2M(n-2) + 1) + 1 = M(n-2) + 2 + 1
  + M(n) = (2M(n-3) + 1) + 2 + 1 = M(n-3) + + +
  + …
  + M(n) = M(n-k) +

k = n-1 :

* + M(n) = (M(1)) +
  + M(n) =
  + M(n) = -1

Exponential sayıda move yapmamız lazım. Bu yüzden:

T(n) = θ()

Counting Cells in a Blob

Consider how we might process an image that is presented as 2D array of color values

Every cell in 2D array is pixel, and each pixel has a color

Information in the image may come from

* an X-ray
* an MRI
* satellite imagery
* etc.

The goal is to determine size of an area in image that is considered abnormal because of its color values

**PROBLEM**

Given a 2D grid of cells, each cell contains either a normal background color or a 2nd color, which indicates the presence of an abnormality

A blob is a collection of contiguous abnormal cells

A user will enter the x, y coordinates of a cell in the blob, and the program will determine the count of all cells in that blob

Anormal renklere sahip birbirine bağlantılı blob bulmaya çalışıyoruz.

**ANALYSIS**

Problem inputs:

* the 2D grid of cells
* the coordinates of a cell in a blob

Problem outputs:

* the count of cells in the blob

**DESIGN**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

else if kısmının sebebi 2 şey olabilir:

* normal bir noktaya gelmişizdir (blobun dışında)
* daha önce gördüğümüz bir noktaya gelmişizdir

else kısmında temporary color’a çevirmemizin sebebi cell’in tekrar sayılmaması

Text

Description automatically generated Text

Description automatically generated

8 kere çağırıyoruz çünkü griddeki 1 cellin 8 komşusu var.

mxm gridimiz olsun, n tane cell içeren blobumuz olsun. Çevresindekilerin hepsinin abnormal olması halinde 1 cell için 8 tane recursive call yapma ihtimalimiz var. Yani n tane cell için 8n tane recursive call yapma ihtimalimiz var. Ayrıca bir de 8n tane de (blobtaki her cell için max 8 tane) blobun çevresindeki abnormal olmayan celler için recursive call yapma ihtimalimiz var.

En fazla 16n tane recursive call yapıyoruz. Her recursive callda da constant time operasyonlar var. Çalışma süresi gridin değil, blobun büyüklüğüyle alakalı.

T(n) = θ(n) 🡪 n: input değil output (return value) m: input (mxm matrix)

Blobun içindeki her cell access edilmeli, o yüzden θ kullandık.

Recursive call sayısı n ile 16n arası bir yerde.

Direkt grid boyuyla hesaplarsan: mxm’den tane cell var. Her cell için 8 komşudan 8 recursive call var. Yani 8’den 🡪 T(m) = θ()

Toplama işlemleri vs. constant time ama büyük bir constant time. Ama sonuçta constant.

Recurrence relation yazmak bu tarz durumlarda (sonraki grid dışarıda mı olacak vs.) zor olduğundan direkt recursive metodun kaç kere çağrılacağını sayıyoruz.

Table

Description automatically generated

**TESTING**

Verify that the code works for the following cases:

* A starting cell that is on the edge of the grid
* A starting cell that has no neighboring abnormal cells
* A starting cell whose only abnormal neighbour cells are diagonally connected to it
* A “bull’s-eye”: a starting cell whose neighbours are all normal bu their neighbours are abnormal
* A starting cell that is normal
* A grid that contains all abnormal cells
* A grid that contains all normal cells

**Backtracking**

Backtracking is an approach to implementing a systematic trial and error search for a solution

An example is finding a path through a maze

If you are attempting to walk through a maze, you will probably walk down a path as far as you can go

* Eventually, you will reach your destination or you won’t be able to go any farther
* If you can’t go any farther, you will need to consider alternative paths

Backtracking is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don’t work

If you never try the same path more than once, you will eventually find a solution path if one exists

Problems that are solved by backtracking can be described as a set of choices made by some method

Recursion allows you to implement backtracking in a relatively straightforward manner

* Each activation frame is used to remember the choice that was made at that particular decision point

A program that plays chess may involve some kind of backtracking algorithm

Finding a Path through a Maze

Problem:

* Use backtracking to find a display the path through a maze
* From each point in a maze you can move to the next cell in a horizontal or vertical direction if the cell is not blocked

Analysis:

* The maze will consist of a grid of colored cells
* The starting point is at the top left corner (0, 0)
* The exit point is at the bottom right corner
  + (getNCols() - 1 , getNRow - 1)
* All cells on the path will be BACKGROUND color
* All cells that represent barriers will be ABNORMAL color
* Cells that we have visited will be TEMPORARY color
* If we find a path, all cells on the path will be set to PATH color

Text

Description automatically generated

Text, letter

Description automatically generated

If maze is nxn, minimum path will require 2n-1 moves.

Tb(n) = θ(n)

Other things other than recursive calls is constant time: θ(1). Running time depends on θ(1)\*(number of calls). Number of calls could be as large as . I don’t have any case where I traverse cells. All the cells should be BACKGROUND color for that case. If so I can go in 2n-1 moves which is best case. But we assume it is near to but smaller. So worst case is not θ() but O().

Tw(n) = O()

T(n) = O()

Testing:

* Mazes that can be solved
* Mazes that can’t be solved
* A maze with no barrier cells
* A maze with a single barrier cell at the exit point

Base cases should be solved directly without any complexity.