Lecture 3 Divide and Conquer

Iterative vs Recursive Algorithms

An algorithm is a set of steps of operations that are performed by a computer to solve a problem. An algorithm received an input that satisfies a certain conditions, **pre-condition,** and transforms the input through the operations into an output, that satisfies desired conditions, **post-condition**. An algorithm needs to satisfy the following conditions:

a) The set of steps is finite: the algorithm stops after a finite number of steps are executed
b) Each step is computable (refer back to the definition of 'computable' in lecture 1.
c) Uniqueness: each step is uniquely defined and depends
d) Generality: the algorithm applies to all instances of the input that satisfy the pre-condition.

There are two main types of algorithms: iterative and recursive. Iterative algorithm consists of an iterative step which is then repeated a set number of times to solve the problem. Recursive algorithm use a 'recursive step' which is applied to smaller and smaller chunks of the input to produce the solution.

Recurrence Relations

Recurrent relations are mathematical equations that are defined in terms of themselves. These relations constitute the basis of mathematical induction.

Examples

Polynomial linear recurrence relation: $a_n = a_{n-1} + 1$ where $a_1 = 1$ what is the value of where a_n

Polynomial linear recurrence relation: $a_n = 1/(1 + a_{n-1})$

Exponential recurrence relation: $a_n = 2 * a_{n-1}$ where a_n

The recursion stops for a base case, e.g., $a_1 = 1$.

Recurrence relations are used to express the complexity of recursive functions.

Exercise 1. Solve the recurrence $na_n = (n-2)a_{n-1} + 2$ for n> 1 with $a_1 = 1$

Exercise 2. Solve the recurrence: $a_n = 2a_{n-1} + 1$ for n>1 with $a_1 = 1$

Exercise 3. Solve the recurrence: $a_n = \frac{n}{n+1}a_{n-1} + 1, n > 0$ with $a_0 = 1$

Recurrence relations are useful to help us understand recursive function and are also useful when calculating the complexity of recursive algorithms.

Recursive Algorithms

Recursive algorithms can use the following techniques:

1) **Simple recursion**

Example 1. Counting elements in a list.
If the list is empty, return zero.
Otherwise,
Step past the first element and count the remaining
Add one to the result.

Example 2. Factorial
The iterative version would be:

```
 8 def iterative_factorial(n):
 9     result = 1
10     for i in range(2,n+1):
11         result *= i
12     return result
13
14 def factorial(n):
15     if n == 1:
16         return 1
17     else:
18         return n * factorial(n-1)
19
```

```
 8
 9 def fib(n):
10     if n == 0:
11         return 0
12     elif n == 1:
13         return 1
14     else:
15         return fib(n-1) + fib(n-2)
16
17 def fibi(n):
18     a, b = 0, 1
19     for i in range(n):
20         a, b = b, a + b
21     return a
22
23
24 memo = {0:0, 1:1}
25 def fibm(n):
26     if not n in memo:
27         memo[n] = fibm(n-1) + fibm(n-2)
28     return memo[n]
29
```

Example 3: The Fibonacci Numbers

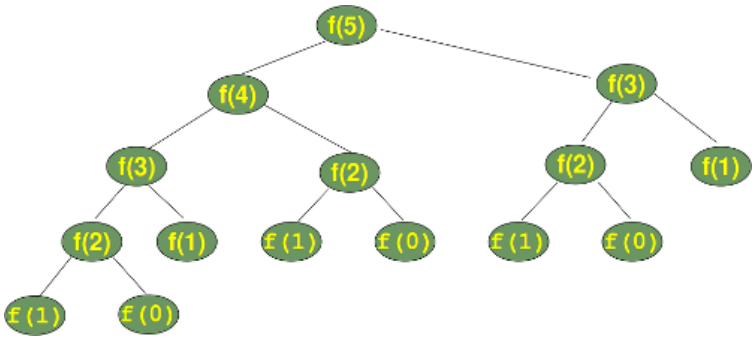
In the figure on the left we have three implementations.

Fib – is the recursive one; Fibi – the iterative one and Fibm is a recursive version that uses a memory to remember previous calculations. We will explain below.

The recursive definition of Fibonacci is much slower than the iterative one. For n= 15 the iterative solution is 240 times faster than the recursive solution**.**

The reason for such a bad performance of the recursive algorithm is that it repeats calculations several times. For example for fib(5), the calculations of fib(3) is done twice and the calculation of fib(2) is done 3 times. For n sufficiently large these repetition will slow down the performance dramatically. (See the graph below).

```
IPython console
Console 1/A
n=14, fib: 0.000703, fibi:  0.000006, percent:     126.54
n=15, fib: 0.001644, fibi:  0.000007, percent:     240.31
n=16, fib: 0.001809, fibi:  0.000006, percent:     302.21
n=17, fib: 0.002917, fibi:  0.000006, percent:     454.67
n=18, fib: 0.004812, fibi:  0.000007, percent:     703.25
n=19, fib: 0.008650, fibi:  0.000011, percent:     777.92
n=20, fib: 0.012942, fibi:  0.000008, percent:    1681.28
```



You can measure the time each algorithm takes to run for n up to 20.

```
 7
 8 from timeit import Timer
 9
10 t1 = Timer("fib(10)","from fibonacci import fib")
11
12 for i in range(1,20):
13     s = "fib(" + str(i) + ")"
14     t1 = Timer(s,"from fibonacci import fib")
15     time1 = t1.timeit(3)
16     s = "fibi(" + str(i) + ")"
17     t2 = Timer(s,"from fibonacci import fibi")
18     time2 = t2.timeit(3)
19     s = "fibm(" + str(i) + ")"
20     t3 = Timer(s,"from fibonacci import fibm")
21     time3 = t3.timeit(3)
22     print("n=%2d, fib: %8.6f, fibi:  %7.6f, fibm:  %7.6f, \
23         percent: %10.2f, percent: %10.2f" % \
24         (i, time1, time2, time3, time1/time2,time3/time2))
25
```

The recursive algorithm can be improved by remembering the calculations made for previous steps. This is done in the version of the suction **fibm** above.

Exercises. Implement the following in Python.

1. Think of a recursive version of the function f(n) = 3 * n, i.e. the multiples of 3
2. Write a recursive Python function that returns the sum of the first n integers. (Hint: The function will be similar to the factorial function!)
3. Write a function which implements the Pascal's triangle:

```
                1
             1     1
          1     2     1
       1     3     3     1
    1     4     6     4     1
1     5    10    10     5     1
```

There are other forms of recursion apart from the simple recursion. These are backtracking, dynamic programming and divide and conquer. We will spend a bit more time of divide and conquer.

2) **Backtracking**: **color a map with no more than 4 colours**.
If all countries have been coloured return true
Otherwise
For each colour c of four colours
If the adjiacent countries are not coloured with c color that country with c
And recursively color its neighbours
If successful return true

3) Dynamic programming: usually used for optimization problems. Usually solves the problems in O(n^2) or O(n^3) for problems for which an iterative algorithm might take exponential time. The problem is broken down in sub problems so that the optimal solution of subproblems will give is the optimal solution(s) for the initial problem.

Example (Longest Common Subsequence) Given two strings S of length n and T of length m, what is the longest sequence they have in common (not necessarily of consecutive letters).
For example S=ABAZDC and T=BACBAD have ABAD as the longest common sequence.
The idea is to look at all the prefixes of A and B. LCS[i,j] of S[1…i] and T[1…j]. Solve L[I,j] in terms of smaller problems.

	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	2	3	3	4

Another example: 1234 and 12224533324 have the longest sequence 1234.
Below is a recursive implementation of the LCS problem.

```
def lcs_rec(a, b):
    """
    >>> lcs('thisisatest', 'testing123testing')
    'tsitest'
    """
    if not a or not b:
        return ""
    x, xs, y, ys = a[0], a[1:], b[0], a[1:]
    if x == y:
        return x + lcs_rec(xs, ys)
    else:
        return max(lcs_rec(a, ys), lcs_rec(xs, b), key=len)
```

Check the lab files for a dynamic implementation of the lcs problem.

4) **Divide and Conquer:**

"Divide the problem into smaller (perhaps overlapping) sub problems, solve them recursively, then use the solutions to solve the original problem."

Mergesort

Sorting is a key operation that is performed by computers in solving problems. Sorting can speed up operations, e.g., finding an item in a large data set. Sometimes, semi-sorting is sufficient and full sorting may take a long time. Ranking and classification are a type of sorting. We will consider sorting lists/arrays of numbers for simplicity. Sorting of other more interesting types of data is done using keys, but the principles of sorting are the same.

Consider the following lists of numbers

a) [2,4,1,3,5]
b) [5,2,8,3,9,1]

You can notice that the degree of 'unsortness' of the two lists are different, the first list being 'almost' sorted. Suppose that you are task with an algorithm for counting the inversions in these arrays, i.e., the number of numbers that are not in the right order and would need to be swapped in order to sort the list.

In the list A) we have three inversions (2,1), (4,1), (4,3). In list B) we have 8 inversions (5,2), (5,3), (5,1), (2,1), (3,1), (8,3), (8,1), (9,1).

Task 1 Suppose you write an iterative algorithm. How would that work? What would be the complexity of that algorithm?

We will show how a recursive algorithm can perform this task in a shorter time using the merge-sort algorithm.

Example of iterative algorithm: Bubblesort

The idea of Bubblesort is a bit similar to the idea of counting inversions. The idea is to 'out right' all the inversions by going through the list several times and swapping any two adjacent elements that are not in the right position.

[2,4,1,3,5] after the first pass will become [2,1,3,4,5] in which we swap (4,1) and (4,3)

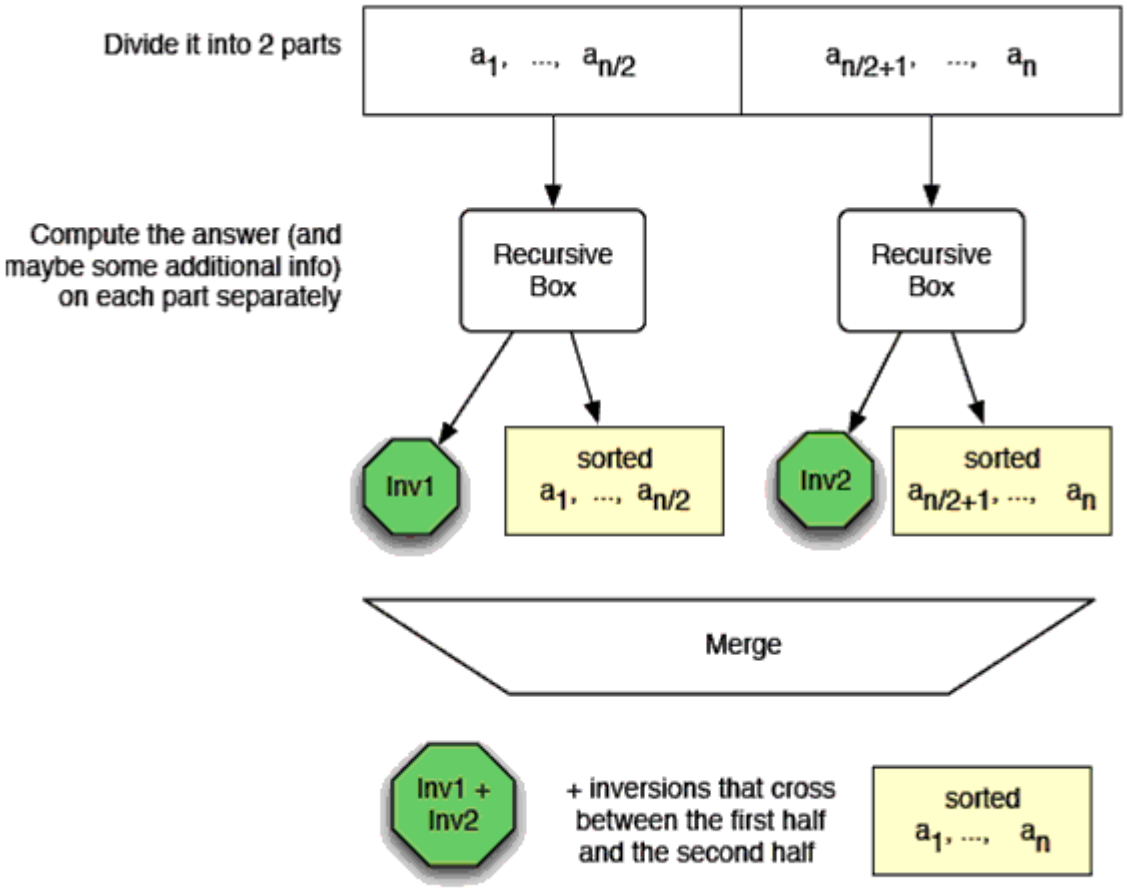
after the second pass will become [1,2,3,4,5] in which we swap (2,1).

Task 2. Write an algorithm for bubblesort in Python.

Example of recursive algorithm: Mergesort

The idea of Mergesort is to use divide and conquer to sort the array. Let us go back to the problem of counting inversions. Suppose you knew how many inversions are in the first half of the array and how many are in the right / second half. They you would simply need to sum these up to give you the overall number of inversions in the whole array. Even better still would be to sort the array while you also count the number of inversions.

The algorithm consists of the following steps:
divide: size of sequence n to two lists of size n/2
conquer: count recursively two lists
combine: the two parts into one that is sorted / is is a trick part (to do it in linear time)



The combine step: this needs to take place in Θ(n). The splitting in half $log_2 n$ times where *n* is the length of the list.So the merge operation for a total of $nlog_2 n$ operations.

Task 3 Write an algorithm that runs in Θ(n) to merge two sorted arrays. (We explain in class)

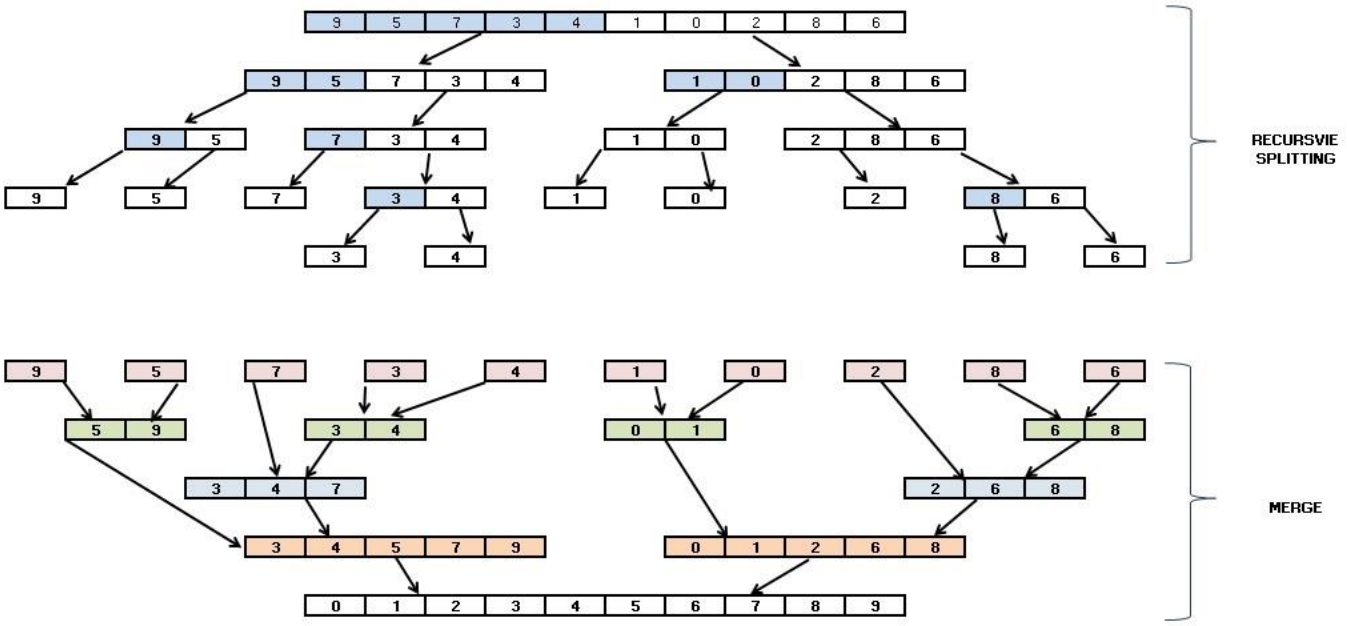


Figure 1 Mergesort Algorithm (https://pythonandr.files.wordpress.com/2015/07/mergesort1.jpg)

```
def merge(a,b):
    """ Function to merge two arrays """
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c += b
    else:
        c += a
    return c

# Code for merge sort

def mergesort(x):
    """ Function to sort an array using merge sort algorithm """
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = int(len(x)/2)
        a = mergesort(x[:middle])
        b = mergesort(x[middle:])
        return merge(a,b)
```

Exercises

Q1) For what kind of input bubblesort performs the best? The worst?

Q2) Explain the time complexity of bubblesort in terms of the three asymptotic notations.

Q3) Given the following list of numbers: [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40] which answer illustrates the list to be sorted after 3 recursive calls to mergesort?

(A) [16, 49, 39, 27, 43, 34, 46, 40]
(B) [21,1]
(C) [21, 1, 26, 45]
(D) [21]

Q4) Given the same list of numbers as in A1 which answer illustrates the first two lists to be merged?

(A) [21, 1] and [26, 45]
(B) [[1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]
(C) [21] and [1]
(D) [9] and [16]