MERRIMACK COLLEGE

CSC6023 - Advanced Algorithms

More Recursion & Dynamic Programming algorithms

Patrick Neff

Making our way through the course

This is a very fast paced course, After this third week topic we are just one week to half the course. So, let’s see the dynamic programming algorithms today.

MERRIMACK COLLEGE

Agenda Presentation

MERRIMACK COLLEGE

Review - Types of Algorithms

Dynamic Programming

- The Basics
    a. Mathematical Optimization Problems
    b. Why it is not in the ...-and conquer family?
- Computer Algorithms:
    a. Factorial
    b. Fibonacci

Examples

- First Example - Balanced 0-1 Matrices
    a. Balanced row permutations
    b. Recursive row additions
- Second Example - Towers of Hanoi
    a. Solving the puzzle
    b. Counting the number of movements

Brute Force Algorithms

MERRIMACK COLLEGE

Straightforward approach

- To find a 4-pin code with digits from 0-9 you can try all combinations: 0000, 0001, ... until the good one is found. In the worst case it will take 10,000 attempts (10^4) to find the right solution
- Cursed by the combinatorics explosion
  - If you have n choices with m possible options you will need to try m^n cases
    - Often leading to an exponential complexity
- Finding the largest element in an array just going through all elements is also a brute force algorithm, that delivers the best possible complexity - linear - $\mathbf{O}(n)$

Recursive Algorithms

A solution by breaking a problem into subproblems of the same kind

- To find a number in a sorted array using Binary Search can be implemented recursively:
  - Check the middle element of the array
    - If it is the searched element, done
    - If it is greater than the searched element, search the right subvector
    - If it is smaller than the searched element, search the left subvector
      - If the array is empty the searched element is absent
- It delivers good results under specific conditions
  - Often leading to a logarithmic complexity ***O****(log n)* - ***O****(n log n)*

Kinds of Algorithms

The … and conquer family

MERRIMACK COLLEGE

Breaking the problem nature to better solve it

- It is not necessarily recursive, but it often is
- Usual kinds:
    - Decrease-and-Conquer
        - Breaking the problem into a smaller one until the solution is trivial
    - Divide-and-Conquer
        - Dividing the problem into subproblems that can be dealt more easily
    - Transform-and-Conquer
        - Modifying the problem to a more amenable form to a particular solution

The … and conquer family

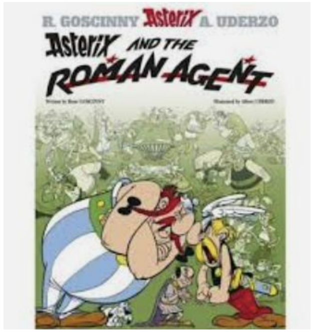
MERRIMACK COLLEGE

Divide-and-Conquer

- To find a fake coin (lighter than legitimate ones) in a pile of *n* coins using a simple scale that compares to piles of coins and say either if both piles weigh the same or if a pile weighs less than the other
  - Split the coins in two piles (if it is even, include all coins; if it is odd leave one out), this allows you to figure out in which split is your fake (lighter) coin, so you can repeat the process with the half pile
    - At each step your problem decreases until you measure just two (maybe out of three) coins a find the fake one
- This one is recursive, and it splits in halves, thus logarithmic - ***O****(log n)*

Kinds of Algorithms

The … and conquer family

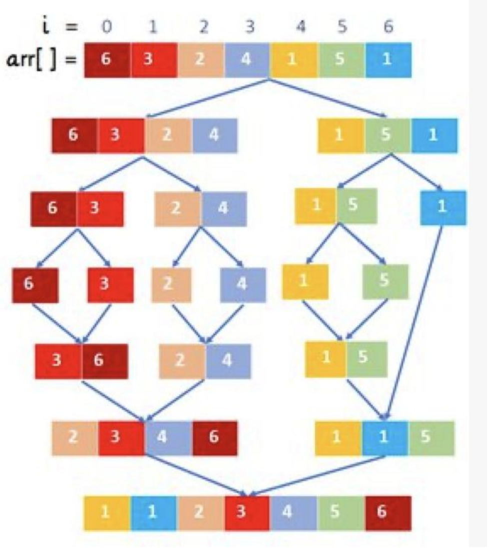


MERRIMACK COLLEGE

Divide-and-Conquer

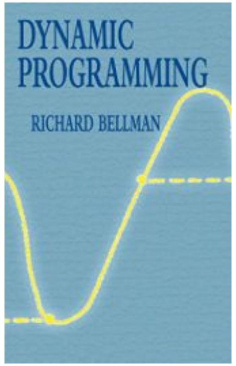
- To sort an array you can split it in half, then in half again, until you have subarrays with a single element, then you merge subarray pairs (sorting) until having the fully sorted array

- The split part happens *log n* times
- The merge part pass by all *n* elements



- This algorithm is recursive, and it splits in halves, thus logarithmic
  O(*n* log *n*)

Basics of Dynamic Programming



MERRIMACK COLLEGE

A Blast from the Past

- In the 50's Richard Bellman, an applied mathematician, developed the concept of dynamic programming
  - From start it was intended as mathematical optimization technique
    - Breaking down a complicated problem into simpler subproblems recursively
      - It was published in the 1950s when computers were about to be important
    - From a computer algorithms point of view this is quite similar to the idea of recursive divide-and-conquer algorithms

Dynamic Programming versus Divide-and-Conquer

"Dynamic programming typically applies to optimization problems in which you make a set of choices in order to arrive at an optimal solution, each choice generates subproblems of the same form as the original problem, and the same subproblems arise repeatedly. The key strategy is to store the solution to each such subproblem rather than recompute it."
-CLRS, 361

The difference between ...

- From definition:
  - Divide-and-conquer combines the solution of subproblems to solve the main problem
  - Dynamic programming uses the solution of subproblems to find the optimum solution to the main problem
- From a practical perspective:
  - Dynamic programming handles overlapping, Divide-and-conquer does not
  - The solution of the subproblems is processed by Dynamic programming, and just absorbed by Divide-and-conquer
  - In Dynamic programming the subproblems are interdependent, and in Divide-and-conquer they are not

MERRIMACK COLLEGE

How does DP work?

- Identify subproblems and storing the solution so as to avoid redundancy
    - merge_sort([4,2,1,3,4,2,1,3])
- Build up to the main solution: use stored solution to build up to the main solution
- Avoid redundancy: by storing solutions to subproblems, DP avoids solving the same problem multiple times

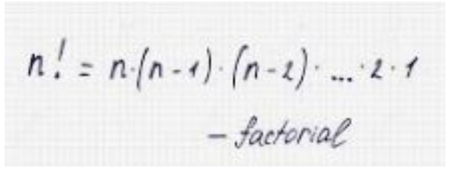
When to use DP?

- Problems with optimal substructure:
  - Problems where the optimal solution to the main problem is found by combining the optimal results of subproblems
- Problems with overlapping subproblems (see Fibonacci sequence example)
  - Problems where non-DP solutions would compute solutions to the same problems more than once or solutions to problems known to be unnecessary

Two Main Types of DP

1. Bottom Up Approach:
    a. Start with the smallest subproblem and build up to the solution to the main problem (as in factorial problem)
2. Top Down Approach:
    a. Start with the main problem and breaks it down to subproblems recursively, somehow checking to ensure that the program does not calculate already known solutions (as we will see in the Fibonacci problem)

Factorial



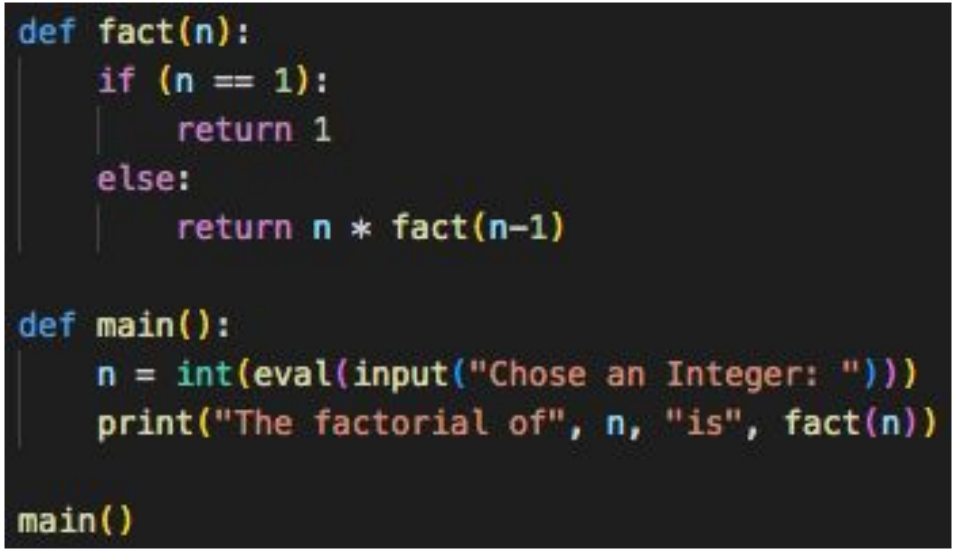
$$n! = n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1$$

– factorial

At every call, but the last, you have to solve an exactly smaller subproblem (the factorial of *n*-1)

Not a Dynamic Programming Implementation

- Computing the factorial of *n* recursively



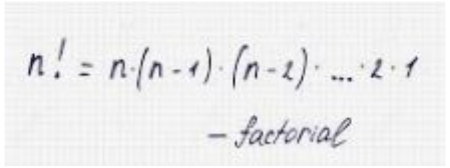
```
def fact(n):
    if (n == 1):
        return 1
    else:
        return n * fact(n-1)

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The factorial of", n, "is", fact(n))

main()
```

MERRIMACK COLLEGE

Factorial

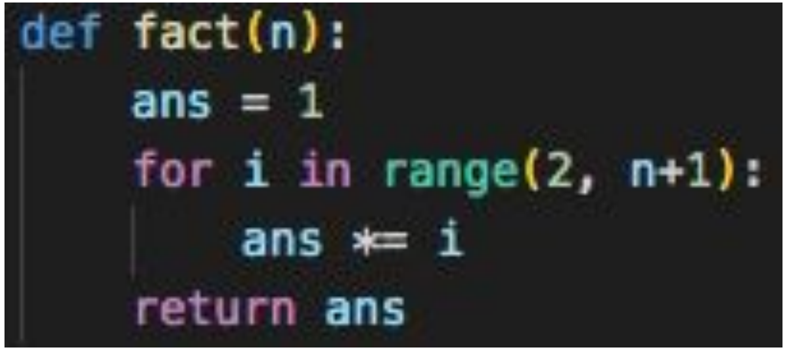
A Dynamic Programming Implementation

- Computing the factorial of *n* iteratively



$$n! = n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot2\cdot1$$

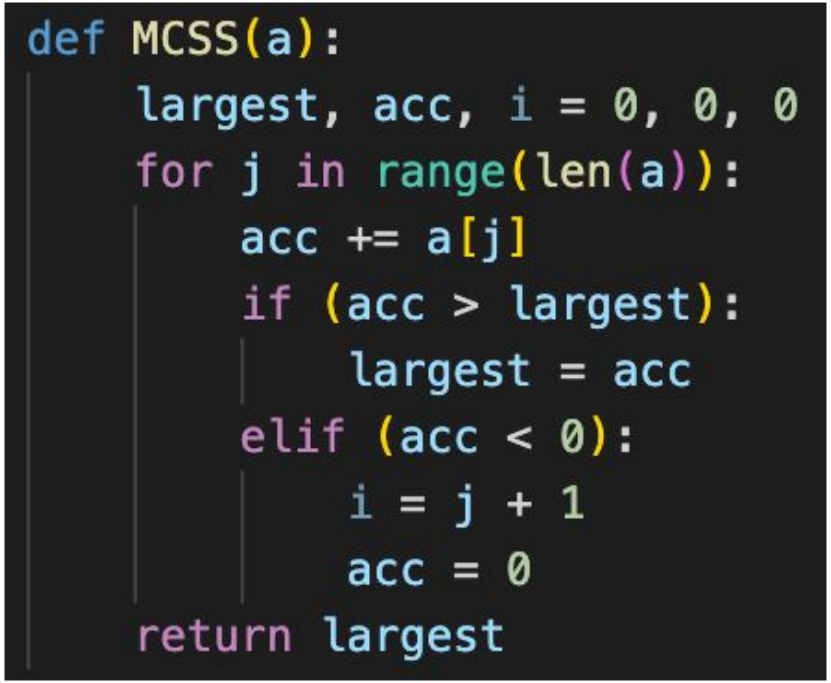
– factorial



```python
def fact(n):
    ans = 1
    for i in range(2, n+1):
        ans *= i
    return ans
```

It builds up the factorial of *n* based on the factorial of all factorials of smaller numbers *n-1, n-2, ..., 1*

Do you Remember MCSS?

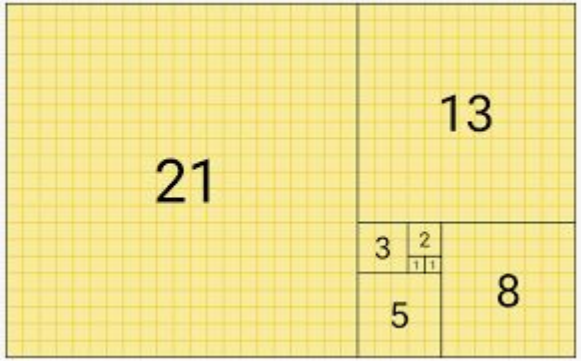


```python
def MCSS(a):
    largest, acc, i = 0, 0, 0
    for j in range(len(a)):
        acc += a[j]
        if (acc > largest):
            largest = acc
        elif (acc < 0):
            i = j + 1
            acc = 0
    return largest
```

This is an example of DP because it is using the acc variable to build up the current contiguous subsequence sum.

Doing so prevents many unnecessary calculations for this problem which is how we got form O(n^3) to O(n).

Fibonacci



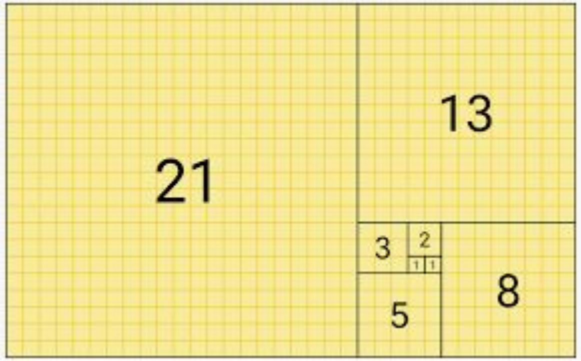
This ratio is frequently found in nature and arts

The problem

- Computing the *n*-th element of a Fibonacci sequence
- A Fibonacci sequence, named after the Italian Mathematician that in the 1202 century introduced both the arabic numerals as well as the sequence to Europe
  - Starting from two numbers 1, the sequence has each number equal to the sum of the two previous one, thus:
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
- To compute the elements is trivial if the previous ones are known, but extremely hard otherwise
  - The definition of the sequence is naturally recursive, but can we do better using dynamic programming?

Computer Algorithms

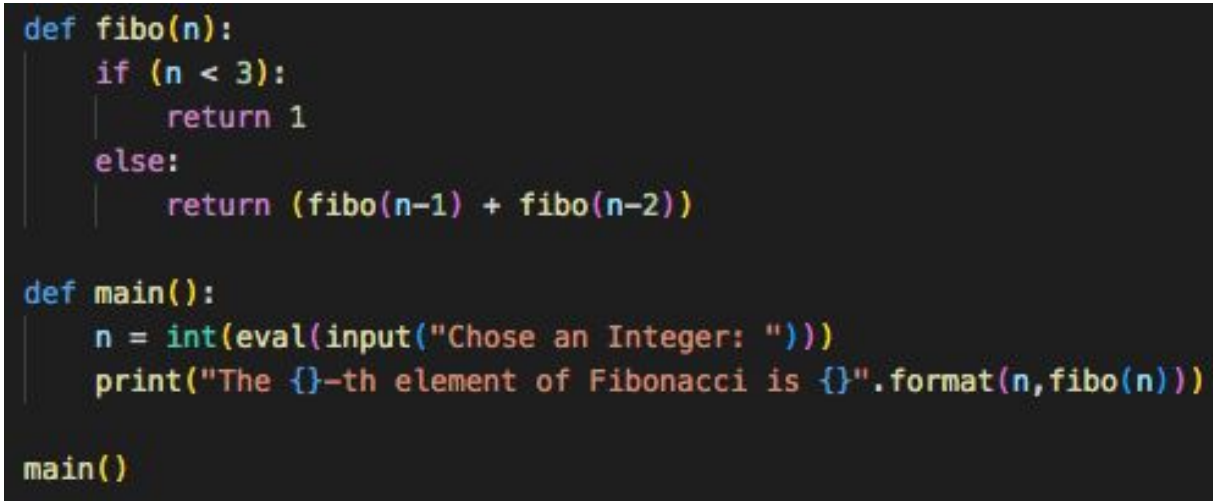
Fibonacci



A pure recursive solution

Not a Divide-and-Conquer algorithm

- Computing the *n*-th element of a Fibonacci sequence



```python
def fibo(n):
    if (n < 3):
        return 1
    else:
        return (fibo(n-1) + fibo(n-2))

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The {}-th element of Fibonacci is {}".format(n,fibo(n)))

main()
```

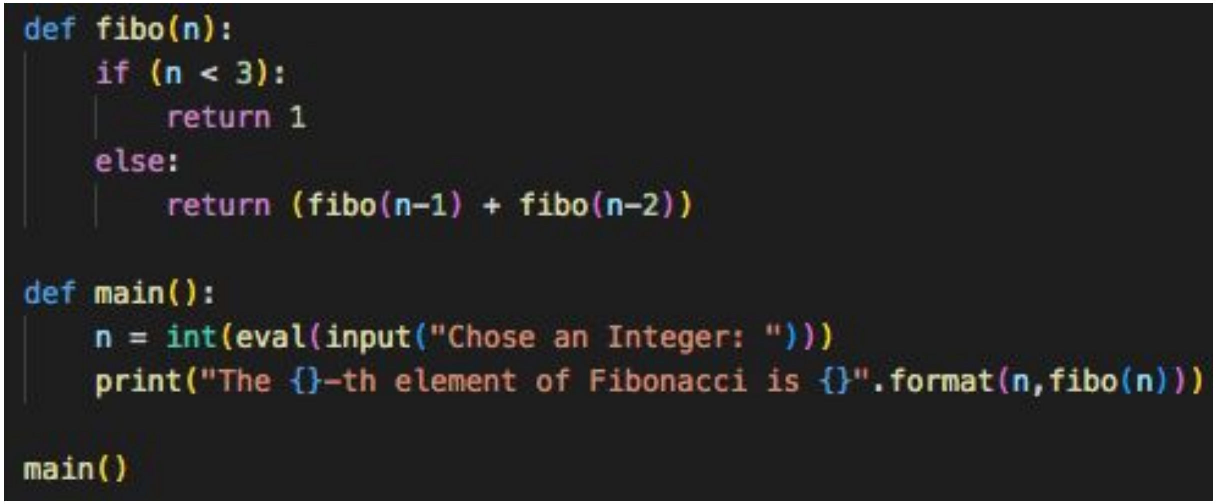
At every call, but the last two, two other calls are made

MERRIMACK COLLEGE

Fibonacci

- Computing the *n*-th element of a Fibonacci sequence

```
fibo(5)
    fibo(4)
        fibo(3)
            fibo(2)
            fibo(1)
        fibo(2)
    fibo(3)
        fibo(2)
        fibo(1)
```

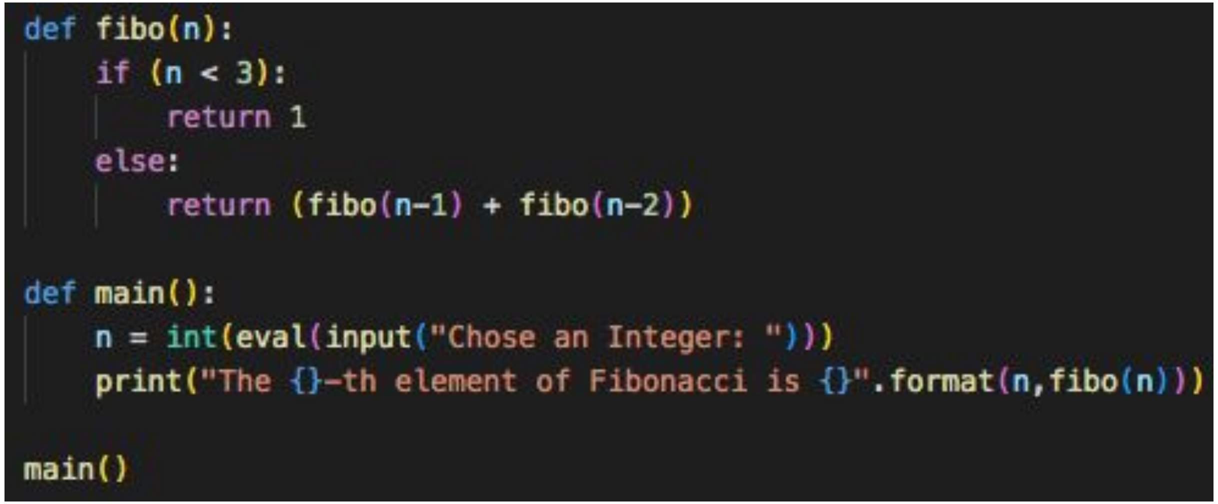


```python
def fibo(n):
    if (n < 3):
        return 1
    else:
        return (fibo(n-1) + fibo(n-2))

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The {}-th element of Fibonacci is {}".format(n,fibo(n)))

main()
```

At every call, but the lasts ones, two other calls are made, creating a lot of repeated calls - exponential complexity

Central Problem of Dynamic Programming

```
fibo(5)
    fibo(4)
        fibo(3)
            fibo(2)
            fibo(1)
        fibo(2)
    fibo(3)
        fibo(2)
        fibo(1)
```



```python
def fibo(n):
    if (n < 3):
        return 1
    else:
        return (fibo(n-1) + fibo(n-2))

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The {}-th element of Fibonacci is {}".format(n,fibo(n)))

main()
```

Dynamic programming asks: how can we prevent all of these repeated calls?

Memoization can be the answer!

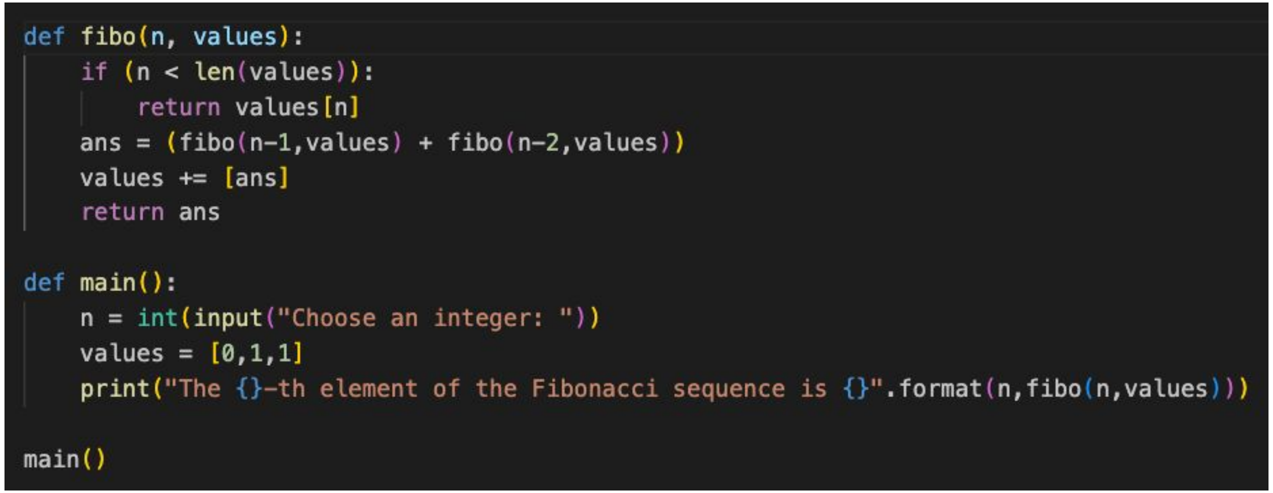
What is memoization?

- “Memoization” is the process by which a computer stores the results of function called on a sub-problem; dynamic programming algorithms are designed to check to see if a sub-problem has already been solved in order to prevent it being solved again.
- The term “memoization” was coined by British AI researcher Donald Michie in the 1960s from the word “memo”, which is in turn short for the Latin “memorandum” (= *that which must be remembered*)

A Dynamic Programming algorithm

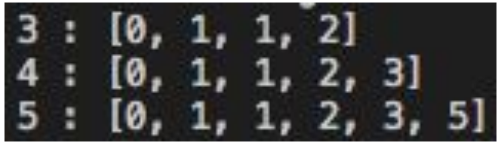
Fibonacci

- Computing the *n*-th element of a Fibonacci sequence



```
def fibo(n, values):
    if (n < len(values)):
        return values[n]
    ans = (fibo(n-1,values) + fibo(n-2,values))
    values += [ans]
    return ans

def main():
    n = int(input("Choose an integer: "))
    values = [0,1,1]
    print("The {}-th element of the Fibonacci sequence is {}".format(n,fibo(n,values)))

main()
```

fibo(5) *update*
    fibo(4) *update*
        fibo(3) *update*
            fibo(2) *V*
            fibo(1) *V*
        fibo(2) *V*
    fibo(3) *V*



```
3 : [0, 1, 1, 2]
4 : [0, 1, 1, 2, 3]
5 : [0, 1, 1, 2, 3, 5]
```

MERRIMACK COLLEGE

Keep in the array *values* the previously computed numbers of Fibonacci sequence elements - ??? complexity

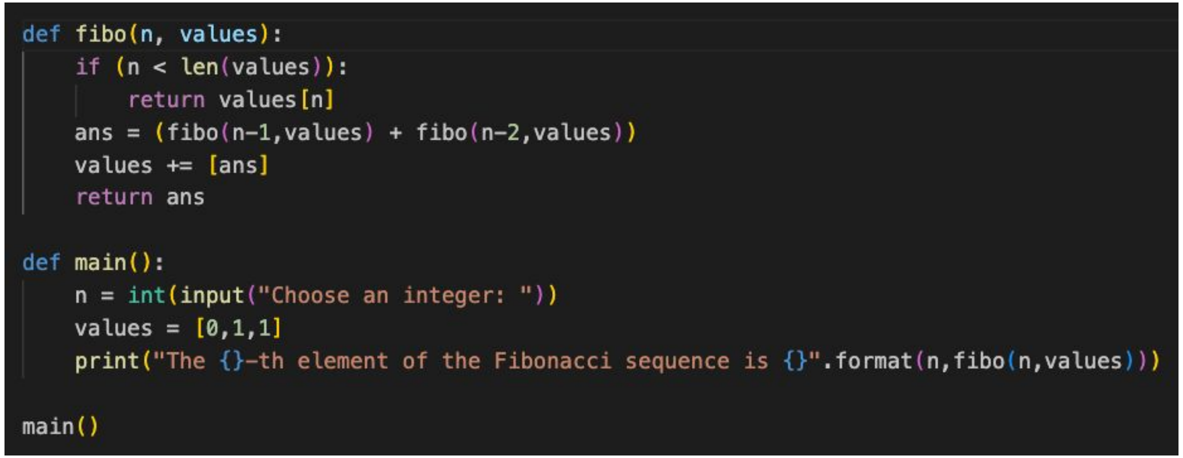
Computer Algorithms

A Dynamic Programming algorithm

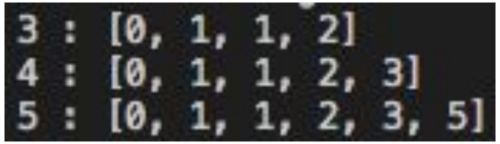
Fibonacci

- Computing the *n*-th element of a Fibonacci sequence

- fibo(5) *update*
  - fibo(4) *update*
    - fibo(3) *update*
      - fibo(2) V
      - fibo(1) V
    - fibo(2) V
  - fibo(3) V



```python
def fibo(n, values):
    if (n < len(values)):
        return values[n]
    ans = (fibo(n-1,values) + fibo(n-2,values))
    values += [ans]
    return ans

def main():
    n = int(input("Choose an integer: "))
    values = [0,1,1]
    print("The {}-th element of the Fibonacci sequence is {}".format(n,fibo(n,values)))

main()
```



```
3 : [0, 1, 1, 2]
4 : [0, 1, 1, 2, 3]
5 : [0, 1, 1, 2, 3, 5]
```

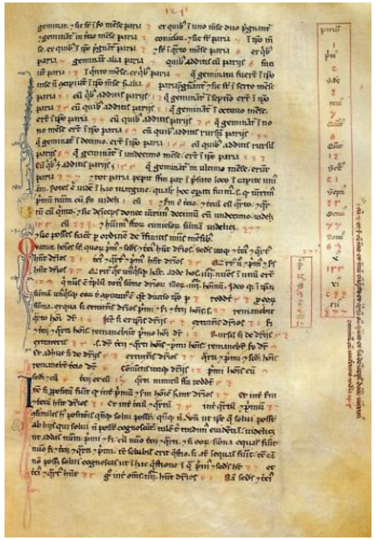
MERRIMACK COLLEGE

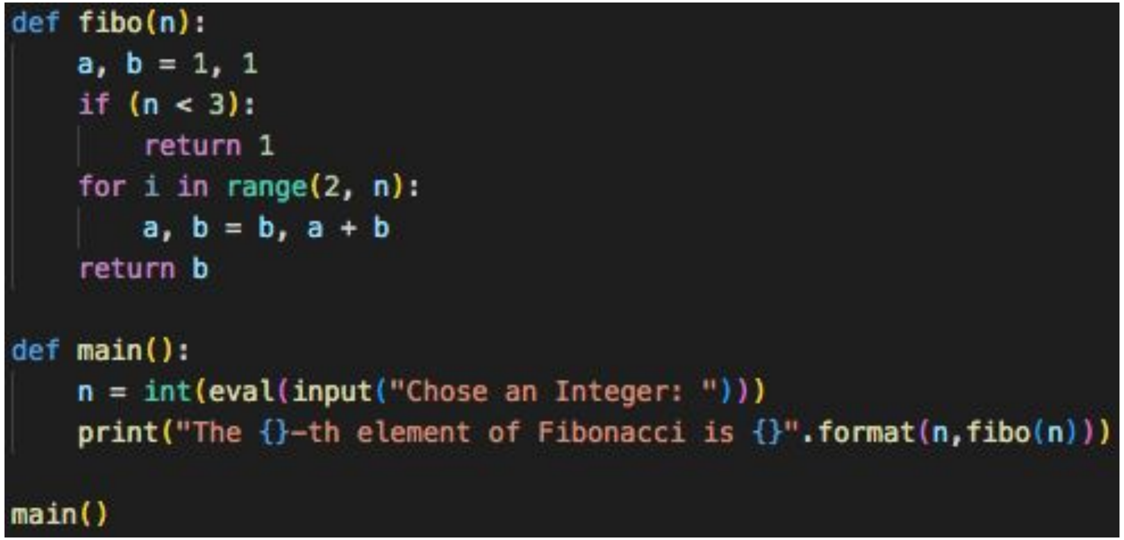
Keep in the array *values* the previously computed numbers of Fibonacci sequence elements - linear complexity

Fibonacci

A Slightly Better Dynamic Programming algorithm

- Computing the *n*-th element of a Fibonacci sequence





```
def fibo(n):
    a, b = 1, 1
    if (n < 3):
        return 1
    for i in range(2, n):
        a, b = b, a + b
    return b

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The {}-th element of Fibonacci is {}".format(n,fibo(n)))

main()
```

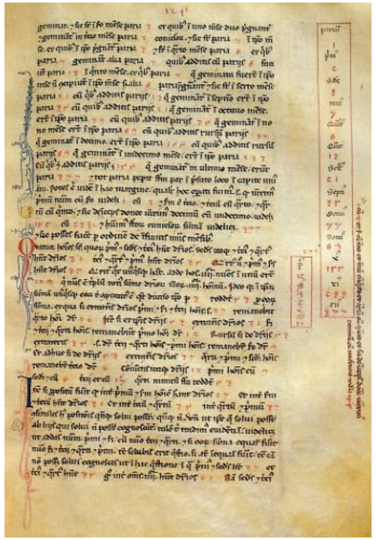
MERRIMACK COLLEGE

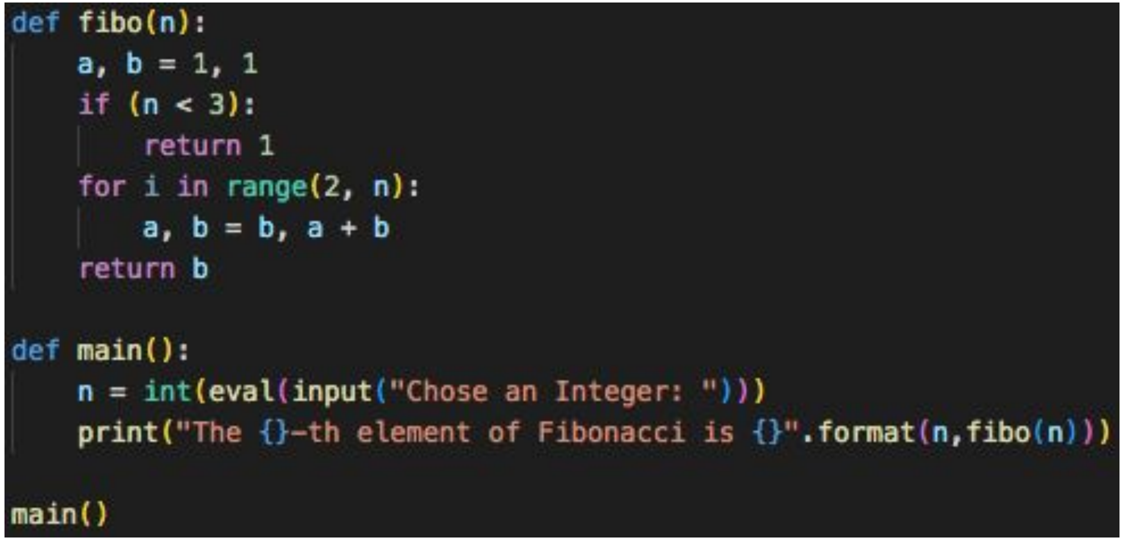
Instead of going backwards, it builds up the sequence by its definition - ??? complexity

Fibonacci

A Slightly Better Dynamic Programming algorithm

- Computing the *n*-th element of a Fibonacci sequence





```
def fibo(n):
    a, b = 1, 1
    if (n < 3):
        return 1
    for i in range(2, n):
        a, b = b, a + b
    return b

def main():
    n = int(eval(input("Chose an Integer: ")))
    print("The {}-th element of Fibonacci is {}".format(n,fibo(n)))

main()
```

MERRIMACK COLLEGE

Instead of going backwards, it builds up the sequence by its definition - linear complexity, same as the recursive dynamic algorithm

Let’s break to run some code.

Examples

Balanced 0-1 matrix

The problem

- Assign 0 or 1 to the elements of a square matrix of size n even, so every row and column have the same number of 0s and 1s. How many combinations have this property?
    - For example for *n* = 4 some of the 90 balanced combinations are these 5 combinations:

$$\begin{bmatrix}0&1&0&1\\1&0&1&0\\0&1&0&1\\1&0&1&0\end{bmatrix} \text{ and } \begin{bmatrix}0&0&1&1\\0&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix} \text{ and } \begin{bmatrix}1&1&0&0\\0&0&1&1\\1&1&0&0\\0&0&1&1\end{bmatrix} \text{ and } \begin{bmatrix}1&0&0&1\\0&1&1&0\\0&1&1&0\\1&0&0&1\end{bmatrix} \text{ and } \begin{bmatrix}1&1&0&0\\1&1&0&0\\0&0&1&1\\0&0&1&1\end{bmatrix}$$

MERRIMACK COLLEGE

To solve it we can do brute force… or dynamic programming

Examples

Balanced 0-1 matrix

MERRIMACK COLLEGE

A Brute Force algorithm

- Assign 0 or 1 to the elements of a square matrix of size n even, so every row and column have the same number of 0s and 1s. How many combinations have this property?
  - If you generate all possible combinations you can check it
    - n^2 elements being either 0 or 1:
      - 2 to the power of n^2
    - For n = 4 there are 2^{16} = 65,536 combinations to test
      - The test itself would require up to n^2 sums to be done for each combination

The brute force solution is simply not feasible for large problems as the complexity is $O(2^{n^2})$

Balanced 0-1 matrix

A Dynamic Programming algorithm

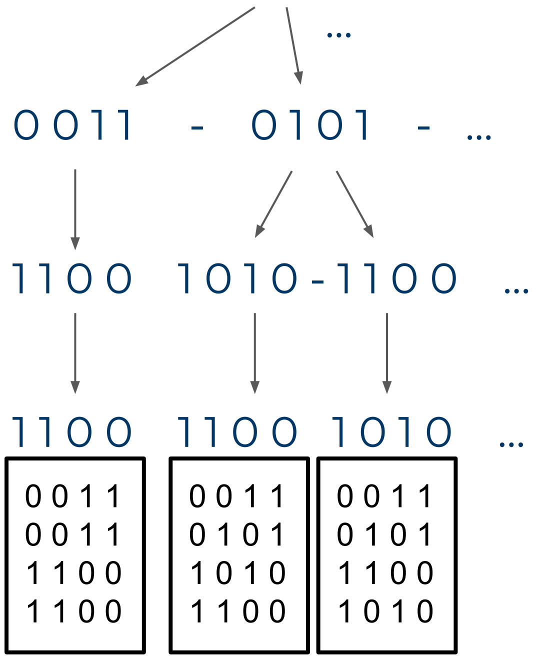
- Try to assign to a row at a time a permutation of 0s and 1s that satisfy the distribution of balanced 0s and 1s
    - For example for *n* = 4 the possible rows are:
        - 0 0 1 1 - 0 1 0 1 - 0 1 1 0 - 1 0 0 1 - 1 0 1 0 - 1 1 0 0
- For the first row all six possibilities are valid
    - For the second row all six possibilities are valid
        - For the third row some combinations are not valid anymore
            - For the fourth row some other combinations are not valid
- This approach requires:
    - A generation of permutations of elements in a row, plus a recursive call of possible rows in a tree structure

Balanced 0-1 matrix

A Dynamic Programming algorithm

- For example for *n* = 4 the possible rows are:

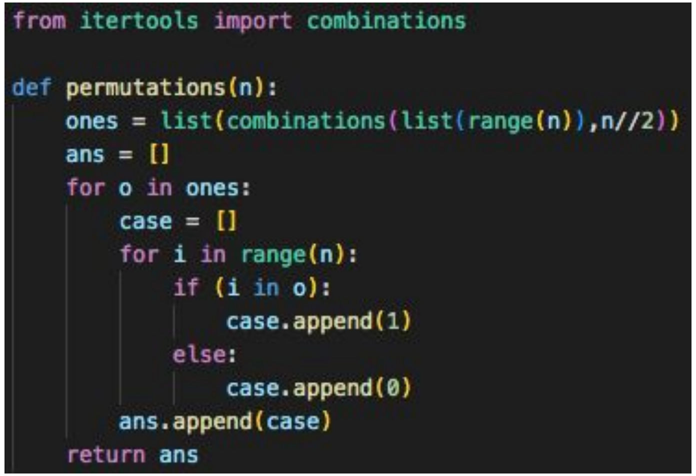
0 0 1 1 - 0 1 0 1 - 0 1 1 0 - 1 0 0 1 - 1 0 1 0 - 1 1 0 0



- For the first row all six possibilities are valid
- For the second row all six possibilities are valid
- For the third row some combinations are not valid anymore
- For the fourth row some other combinations are not valid
- Each leaf of such tree is a possible combination of a balanced matrix

Examples

Balanced 0-1 matrix



```python
from itertools import combinations

def permutations(n):
    ones = list(combinations(list(range(n)),n//2))
    ans = []
    for o in ones:
        case = []
        for i in range(n):
            if (i in o):
                case.append(1)
            else:
                case.append(0)
        ans.append(case)
    return ans
```

MERRIMACK COLLEGE

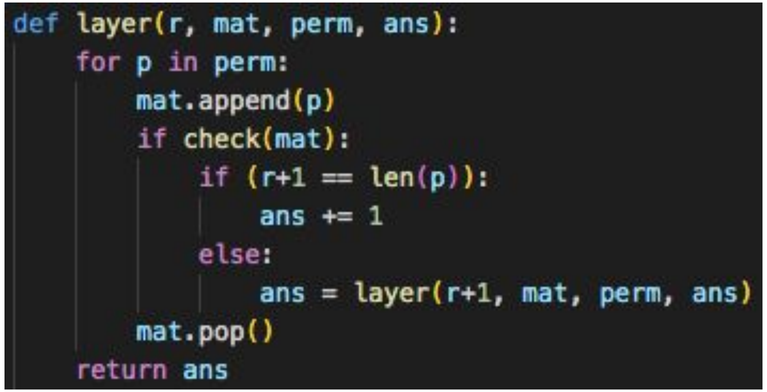
A Dynamic Programming algorithm

- To compute the permutations for each row
  - Given the size *n* it is necessary to have *n/2* numbers 0 and *n/2* numbers 1
    - Therefore we can choose n/2 indices to be marked as 1, and the rest as 0
      - From *n* choose *n/2*
- Package ***itertools***, function ***combinations***
  - https://docs.python.org/3/library/itertools.html
  - Example:
    - https://datagy.io/python-combinations-of-a-list/

  - list(range(4)) -> [0, 1, 2, 3]
  - combinations([0, 1, 2, 3], 4//2) -> [0,1], [0,2], ...
  - return [[1,1,0,0], [1,0,1,0], ...]

Examples

Balanced 0-1 matrix



```
def layer(r, mat, perm, ans):
    for p in perm:
        mat.append(p)
        if check(mat):
            if (r+1 == len(p)):
                ans += 1
            else:
                ans = layer(r+1, mat, perm, ans)
        mat.pop()
    return ans
```

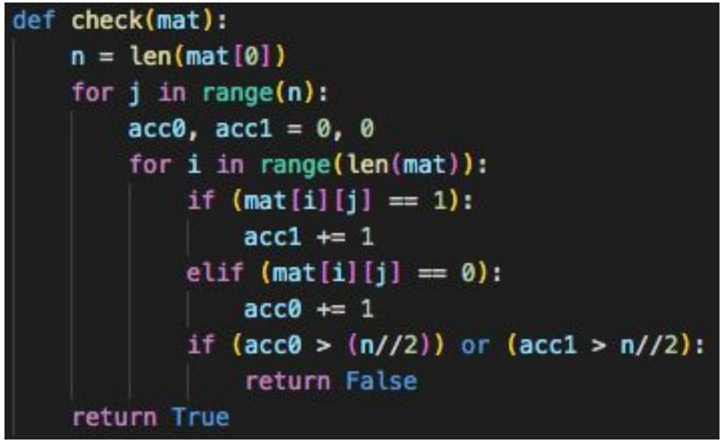
MERRIMACK COLLEGE

A Dynamic Programming algorithm

- To create the matrices we use a recursive function that adds the possible rows (permutations) and check if the new row does not violate the balance
  - The recursive function ***layer***
  - Input parameters:
    - ***r*** - the current row, 0 is the first row, *n-1* is the last row
    - ***mat*** - the current array of rows
    - ***perm*** - the balanced permutation rows
    - ***ans*** - the current number of balanced matrices (also the returned output parameter)
  - Add a new row with all possible permutations
    - check it, if last row count it, if not, call recursively for the next row, then remove it

Examples

Balanced 0-1 matrix



```
def check(mat):
    n = len(mat[0])
    for j in range(n):
        acc0, acc1 = 0, 0
        for i in range(len(mat)):
            if (mat[i][j] == 1):
                acc1 += 1
            elif (mat[i][j] == 0):
                acc0 += 1
            if (acc0 > (n//2)) or (acc1 > n//2):
                return False
    return True
```

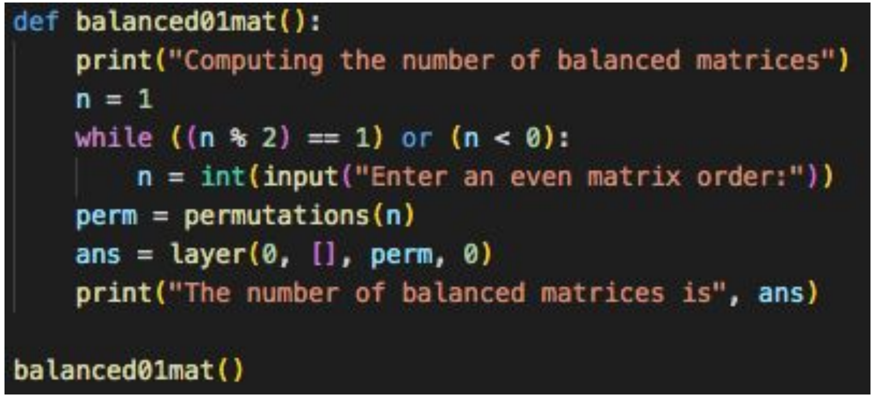
MERRIMACK COLLEGE

A Dynamic Programming algorithm

- Check if the current partial matrix does not violate the column balance (row balance is assured)
    - For all columns
        - Counts the number of 0s and 1s
            - Return *False* if 0s or 1s are more than half the order (*n//2*)
            - If it does not violate the column balance returns *True*
    - Checking prevents the recursive calls of ***layer*** function to all possible permutation combinations
        - For example for *n=4*, there are 6 permutations, therefore 6^4 = 1,296 combinations, but due to ***check*** function only 133 recursive calls are made

Examples

Balanced 0-1 matrix



```
def balanced01mat():
    print("Computing the number of balanced matrices")
    n = 1
    while ((n % 2) == 1) or (n < 0):
        n = int(input("Enter an even matrix order:"))
    perm = permutations(n)
    ans = layer(0, [], perm, 0)
    print("The number of balanced matrices is", ans)

balanced01mat()
```

MERRIMACK COLLEGE

A Dynamic Programming algorithm

- Putting all together
  - Asks the user the matrix order (an even number)
    - A while loop guarantees a positive even number
      - It computes the permutations
      - It calls ***layer*** function for the first row, with an empty matrix (no rows), and initially no balanced matrix found
      - Full code available here: balanced01mat.py
      - What is the complexity of the solution?
  - Better than brute force? 2^{n^2}

Examples

Balanced 0-1 matrix

You can find more info on combinatorics like “n choose k” at:
[Binomial Coefficient (wikipedia)]

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

A Dynamic Programming algorithm

- The complexity of the Dynamic Programming algorithm will be
  - *O(2^n)* for the find permutations part
    - Note the number of permutations is

$$\frac{n!}{\frac{n}{2}!\frac{n}{2}!}$$

For example for n=4:

$$\frac{4x3x2x1}{2x1x2x1}$$

  - *O(2^n)* for the add possible rows recursive part
    - As the of number of possible valid combinations is bounded by *2^n*
- It is big, but much better than brute force still

MERRIMACK COLLEGE

Dynamic Programming Algorithms

Balanced 0-1 matrix

Task #1 for this week's In-class exercises

- Run the balanced01mat.py program and compute the number of balanced matrices for matrices of order 2, 4 and 6 (run the program as it is for discovering the numbers)
- Change the program to instead of asking the user, it calls the function with 2, 4 and 6 sequentially
- Include remarks in the code with the expected values for 2, 4 and 6

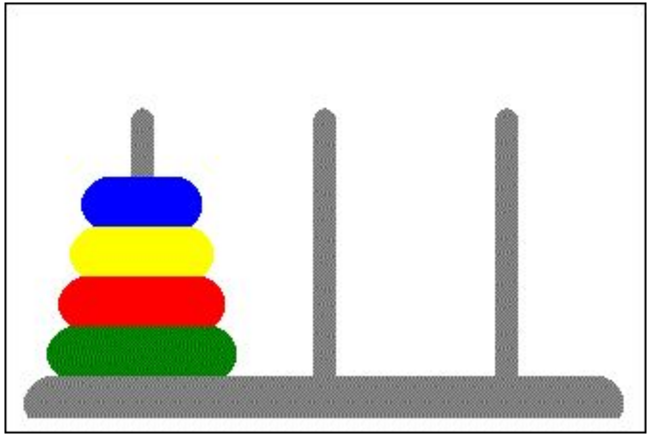
Go to IDLE and try to program it
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: Friday 11:59 PM EST

MERRIMACK COLLEGE

Examples

Second Example



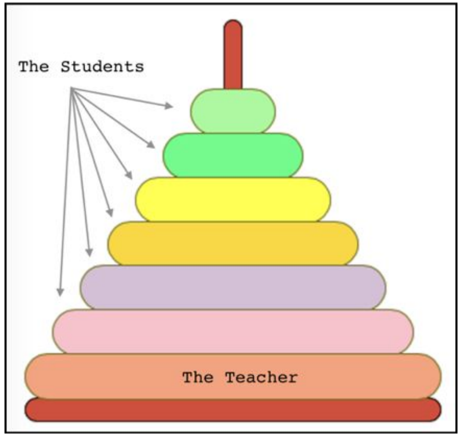
MERRIMACK COLLEGE

Towers of Hanoi Puzzle

- All *n* disks start in the first pole
  - Each movement takes one single disk (the one on top) from a pole to another
  - A larger disk cannot be placed on top of a smaller one
- Two kind of problems
  - How to move efficiently (minimal moves) all disks to the last pole?
  - How many are the minimal moves for *n* disks
- Try it here with 3, 4, 5, and 6 disks
  - https://www.mathsisfun.com/games/towerofhanoi.html

Examples

Towers of Hanoi

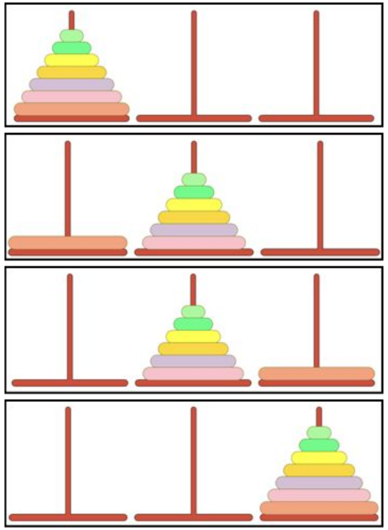


MERRIMACK COLLEGE

How to Solve it Efficiently

- The largest disk is the biggest problem
  - it can only be moved when it is the only disk in a pole
  - it can only go to an empty pole
  - it should move only once from the first to the last pole
- Basic Procedure
  - Release “Teacher”
    - move all “Students” to the intermediate pole (a *n-1* problem)
  - Move “Teacher” to the destination pole
  - Put back all “Students” on top of “Teacher” (another *n-1* problem)

Towers of Hanoi



How to Solve it Efficiently

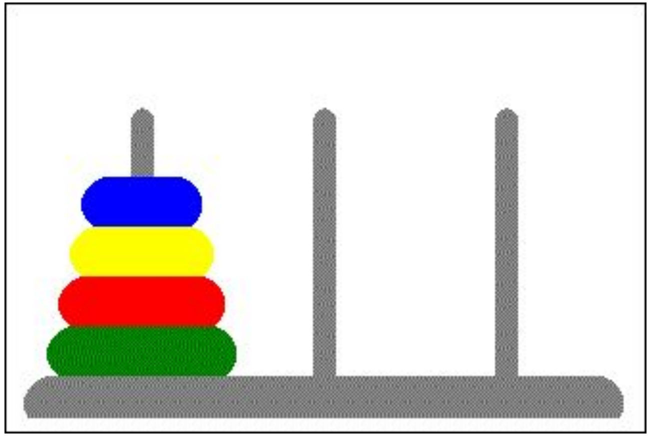
- The solution may be summarized as
  - ***S(n, from, to)***
    - Moving ***n*** disks from pole ***from*** towards pole ***to***
  - ***S(1, from, to)***
    - Simply move the one disk from pole ***from*** towards pole ***to***
  - ***S(n, from, to)*** implies performing
    - ***S(n-1, from,*** *not* ***(from,to))***
    - ***S(1, from, to)***
    - ***S(n-1,*** *not* ***(from,to), to)***

MERRIMACK COLLEGE

To move an odd number of disks you go towards the final pole
To move an even number, you go towards the intermediate one

Examples

Second Example



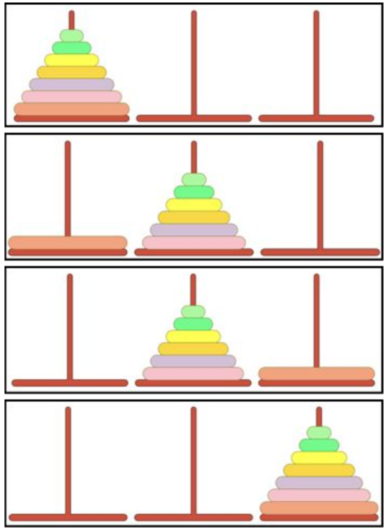
MERRIMACK COLLEGE

Towers of Hanoi Puzzle

- Try it again here with 3, 4, 5, and 6 disks
  - https://www.mathsisfun.com/games/towerofhanoi.html
- Make sure you decide each time:
  - Which disk to move
    - Try to "release" the disks in the appropriate order
  - Towards each pole you should move it
    - Keeping in mind how many recursions until moving a single disk
      - Odd number of disks (1, 3, 5, etc.) go straight to your final pole

How Many Are The Minimal Moves

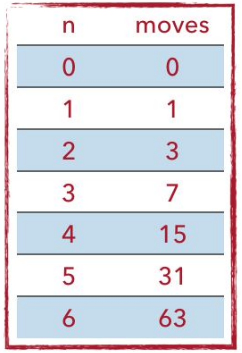
Towers of Hanoi



- To move all "Students" to the middle pole
  - $T(n-1)$
- Move "Teacher"
  - 1
- To move all "Students" to the destination pole
  - $T(n-1)$

A recursive formula:

- $T(n) = T(n-1) + 1 + T(n-1)$
- $T(n) = 2\,T(n-1) + 1$
- $T(1) = 1$



n	moves
0	0
1	1
2	3
3	7
4	15
5	31
6	63

Dynamic Programming Algorithms

Towers of Hanoi

Task #2 for this week's In-class exercises

- Implement a recursive program that asks the number of disks and delivers the minimal number of moves to solve the Towers of Hanoi efficiently
- Your program must have a recursive function that delivers the number of movements for a given number of disks

Go to IDLE and try to program it
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: Friday 11:59 PM EST

MERRIMACK COLLEGE

Dynamic Programming Algorithms

Third Assignment

Project #3 - this week's Assignment

- Create a program that computes the "Tribonacci" sequence numbers
  - Unlike the traditional Fibonacci sequence (a number is the sum of the two previous ones), here a number is the sum of the three previous ones (the initial numbers are 1,1,1)
- The first 9 elements of the sequence are:
  - 1, 1, 1, 3, 5, 9, 17, 31, 57, ...
    - Your program should asks the user a positive Integer *n* and the deliver the *n*-th element of the Tribonacci sequence
      - For example, for ***n = 6***, it delivers ***9***
- Make sure your program uses Dynamic Programming in an efficient way (for example, keeping in memory previously computed elements)

MERRIMACK COLLEGE

Dynamic Programming Algorithms

Third Assignment

Project #3 - this week's Assignment

- This program must be your own, do not use someone else's code
- Any specific questions about it, please bring to the Office hours meeting this Friday or contact me by email
- This is a challenging program to make sure you are mastering your Python programming skills, as well as your asymptotic analysis understanding
- Don't be shy with your questions

Go to IDLE and try to program it
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: next Monday 11:59 PM EST

MERRIMACK COLLEGE

That's all for today folks!

This week's tasks

- Discussion:
  - Deadline: Friday for comments
- Tasks #1 and #2 for the worksheet
  - Deadline: Friday 11:59 PM EST
- Quiz #3 to be available this Friday
  - Deadline: Next Monday 11:59 PM EST
- Project #3 assignment
  - Deadline: Next Monday 11:59 PM EST

- Try all exercises seen in class and consult the reference sources, as the more you practice, the easier it gets

Next week

- Greedy algorithms

- Don't let work pile up!
- Don't be shy about your questions

MERRIMACK COLLEGE

Have a Great Week!