



MERRIMACK COLLEGE

CSC 6013

Week 5

Complexity of Recursive Algorithms

Algorithms and Discrete Structures - Dr. Paulo Fernandes

Presentation Agenda

Week 5

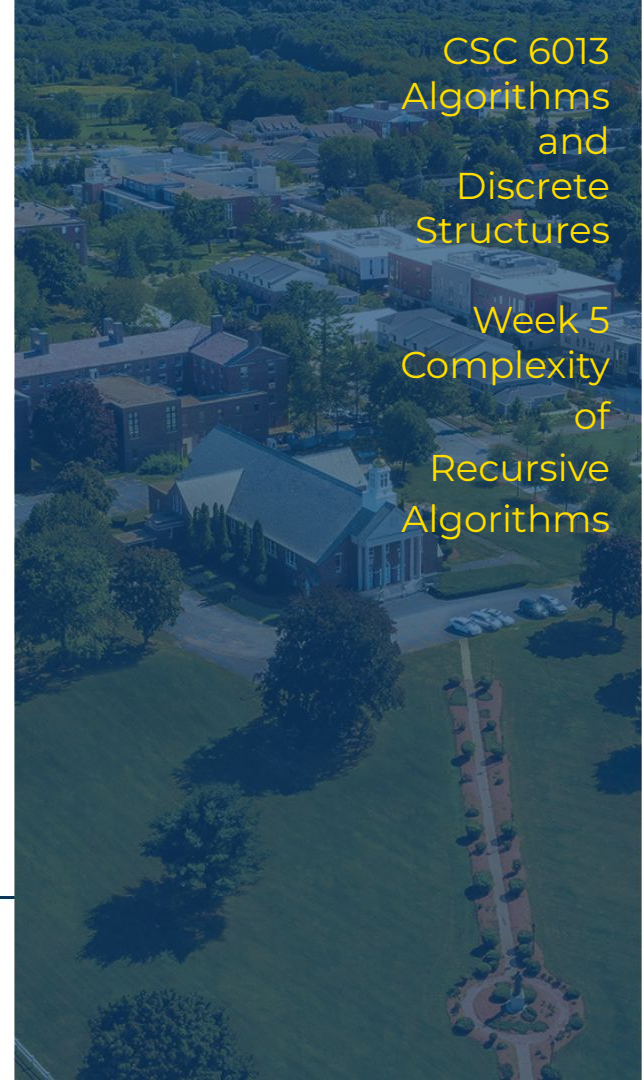
- Recursive Examples
 - a. factorial
 - b. Fibonacci sequence
 - c. depth first search in graphs (DFS)
 - d. towers of hanoi
 - e. binary search
 - f. maximum element in an array
- Complexity of Recursive Algorithms
 - a. Back-substitution method
 - b. Master method
- This Week's tasks



MERRIMACK COLLEGE

CSC 6013
Algorithms
and
Discrete
Structures

Week 5
Complexity
of
Recursive
Algorithms



Recursive Algorithms Examples

Algorithms Examples

- factorial
- Fibonacci sequence
- depth first search in graphs (DFS)
- towers of hanoi
- binary search
- maximum element in an array



Example A - Factorial

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



```
1 def fact(n):
2     if (n == 1):
3         return 1
4     else:
5         return n * fact(n-1)
```

- Computing the factorial of n iteratively is a simple for loop that performs $n-1$ iterations:
 - from **2** to n
 - for example for $n = 5$

$1*2$

$2*3$

$6*4$

$24*5$

- Computing the factorial of n recursively performs $n-1$ recursive calls:
 - for $n-1$ to **1**
 - for example for $n = 5$

$5*fact(4)$ $4*fact(3)$ $3*fact(2)$ $2*fact(1)$



Example B - Fibonacci Sequence

```
1 def fibo(n):
2     if (n in [1, 2]):
3         return 1
4     else:
5         nMinus2 = 1
6         nMinus1 = 1
7         for i in range(3, n+1):
8             nMinus2, nMinus1 = nMinus1, nMinus2+nMinus1
9     return nMinus1
```



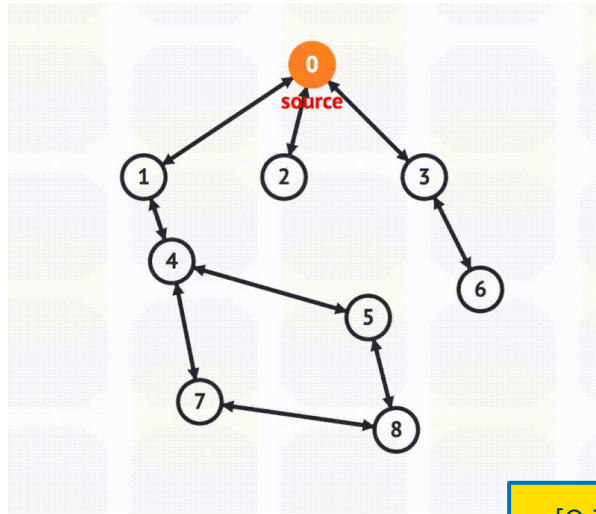
```
1 def fibo(n):
2     if (n in [1, 2]):
3         return 1
4     else:
5         return fibo(n-1) + fibo(n-2)
```

Easy to
read, but
much
slower!

- Computing the ***n-th*** term of Fibonacci iteratively is a for loop that performs ***n-2*** iterations:
 - from **3** to ***n***, for example ***n = 6***
1+1 1+2 2+3 3+5
- Computing the ***n-th*** term of Fibonacci iteratively performs a little less than ***2n*** recursive calls:
 - fibo(1) and fibo(2) require 0 calls;
 - fibo(3) requires 2 calls;
 - fibo(4) requires 4 calls;
 - fibo(5) requires 8 calls;
 - fibo(6) requires 14 calls.



Example 1 - Depth First Search in Graphs



0
1
4
5
8
7
2
3
6

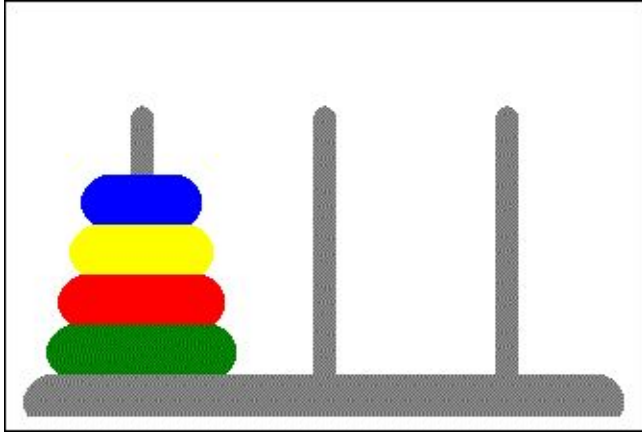
[0 1 6 7 2 3 8 5 4]
0 1 2 3 4 5 6 7 8
i = 1...8 count = 9

```

1  def DFS(V, E):
2      def __visit(i, count):
3          V[i], count = count, count+1
4          for e in E:
5              if (e[0] == i) and (V[e[1]] == -1):
6                  count = __visit(e[1], count)
7              elif (e[1] == i) and (V[e[0]] == -1):
8                  count = __visit(e[0], count)
9          return count
10
11     for i in range(len(V)):
12         V[i] = -1
13     count = 0
14     for i in range (len(V)):
15         if (V[i] == -1):
16             count = __visit(i, count)
17
18     V = [0]*9
19     E = [[0,1,1], [0,2,1], [0,3,1], [1, 4, 1], [3,6,1],
20         [4,5,1], [4,7,1], [5,8,1], [7,8,1]]
21     DFS(V,E)
22     print(V)
    
```



Example 2 - Towers of Hanoi



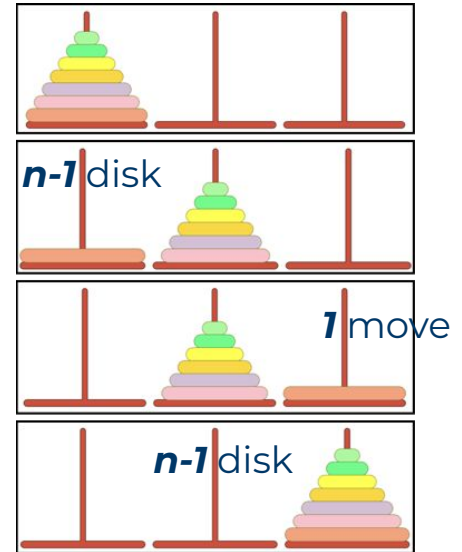
To solve a n disk problem needs to solve two $n-1$ disk problems plus 1 movement.

The number of required movements, let's call it $T(n)$ for n disks, can be expressed as:

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

The stop condition (trivial case) can be:

- $T(0) = 0$




Example 3 - Binary Search

- Giving the slice **start** to **end**, check **mid**:
 - If **mid** is equal to **k**, returns the **mid**;
 - If **mid** > **k**, search from **start** to **mid**;
 - If **mid** < **k**, search from **mid** to **end**;

Target = 5

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

```
1 def binSearch(A, start, end, k):
2     mid = (end+start)//2
3     if (start > end):
4         return None
5     elif (A[mid] == k):
6         return mid
7     elif (A[mid] > k):
8         return binSearch(A, start, mid-1, k)
9     else:
10        return binSearch(A, mid+1, end, k)
11
12 A = list(range(10))
13 for i in [5, 2, 9, 4, 10]:
14     print("{} is at index {}".format(i, binSearch(A, 0, len(A)-1, i)))
```



5 is at index 5
2 is at index 2
9 is at index 9
4 is at index 4
10 is at index None

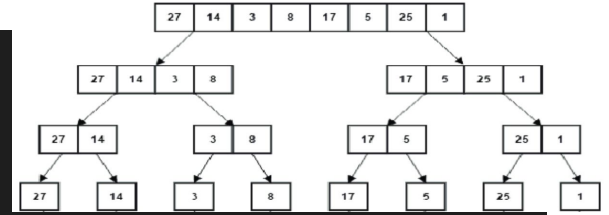
- At the start check the whole array.



Example 4 - Maximum Element in an Array

- Giving a slice of the array (from **start** to **end**), the largest element is either the the largest among the first half or the larger of the second half of it;
- At the start check the whole array.

```
1 def Max(A, start, end):
2     if (start == end):
3         return end
4     else:
5         mid = (end+start)//2
6         fst = Max(A, start, mid)
7         lst = Max(A, mid+1, end)
8         return fst if A[fst] > A[lst] else lst
9
10 from random import randint
11 A = [randint(0,100000) for _ in range(1000)]
12 i = Max(A, 0, len(A)-1)
13 print("The maximum number is", A[i], "at index", i)
```



Complexity of Recursive Algorithms

Counting the number of recursive calls

To compute the complexity of recursive algorithms it is necessary to count the number of recursive calls. There is basically two ways to do that. One is tricky, but can be applied to any recursive algorithm. The other is a straightforward method, but it cannot be applied to all recursive algorithms.

- **Back-substitution method**
 - Factorial
 - Towers of Hanoi
- Master method
 - Binary Search
 - Maximum Element in a array



Back-Substitution Method

- To find out the complexity of a recursive algorithm may require a different technique than the one we saw for brute force algorithms, not because they were brute force, but because they were non recursive.
- One of the techniques to compute the complexity of recursive algorithms is called back-substitution. It basically consists to express the amount of work of each call as a function of the calls it may require.
- Let's see how it work in a practical example, the complexity of the factorial example.




Back-Substitution Method

- The **factorial algorithm** basically performs 1 multiplication per call of the recursive function, except the last call (**$n = 1$**) that just delivers the number 1 with 0 multiplications.
- Therefore, the number of multiplications can be expressed by the recursive formula **$T(n)$** as the number of multiplications for a number **n** :

$$T(1) = 0 \qquad T(n) = 1 + T(n - 1)$$


```
1 def fact(n):  
2     if (n == 1):  
3         return 1  
4     else:  
5         return n * fact(n-1)
```



Back-Substitution Method

$$T(n) = 1 + T(n - 1) \quad T(1) = 0$$

```
1 def fact(n):  
2     if (n == 1):  
3         return 1  
4     else:  
5         return n * fact(n-1)
```




- We can rewrite the recursive relation replacing n :
 - $T(x) = 1 + T(x-1)$
- We can replace x by $n-1$:
 - $T(n-1) = 1 + T((n-1)-1)$
 - $T(n-1) = 1 + T(n-2)$
- Replacing $T(n-1)$ into the original recurrence relation:
 - $T(n) = 1 + T(n-1)$
 - $T(n) = 1 + 1 + T(n-2)$
- We can replace x by $n-2$:
 - $T(n-2) = 1 + T((n-2)-1)$
 - $T(n-2) = 1 + T(n-3)$
- Replacing $T(n-2)$ into the current recurrence relation:
 - $T(n) = 1 + 1 + T(n-2)$
 - $T(n) = 1 + 1 + 1 + T(n-3)$
- And we can keep performing substitution backwards...
 - $T(n) = 1 + 1 + 1 + 1 + T(n-4)$
 - $T(n) = 1 + 1 + 1 + 1 + 1 + T(n-5)$



Back-Substitution Method

$$T(1) = 0 \quad T(n) = 1 + T(n - 1)$$

```
1 def fact(n):  
2     if (n == 1):  
3         return 1  
4     else:  
5         return n * fact(n-1)
```



Substituting from **$n-k$** backwards
we have:

- **$T(n) = 1 + T(n-1)$**
- **$T(n) = 1 + 1 + T(n-2)$**
- **$T(n) = 1 + 1 + 1 + T(n-3)$**
- **$T(n) = 1 + 1 + 1 + 1 + T(n-4)$**
- **$T(n) = 1 + 1 + 1 + 1 + 1 + T(n-5)$**
- ...

The pattern is **$T(n) = k + T(n-k)$**

Since we know **$T(1) = 0$** ,
which value of **k** gives
 $n-k$ equal to 1?

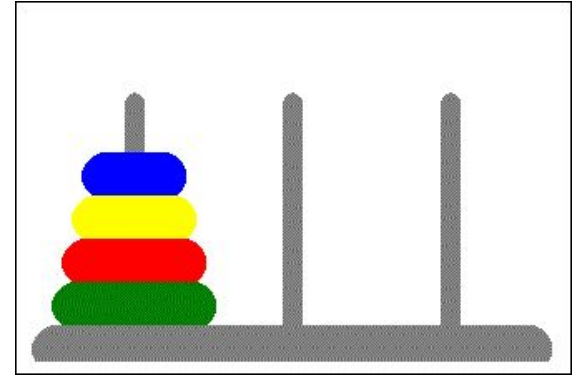
- **$k = n-1$**
 - **$T(n) = n-1 + T(n-(n-1))$**
 - **$T(n) = n-1 + T(1)$**
 - **$T(n) = n-1$**



Back-Substitution Method

- The **Towers of Hanoi example** basically performs one multiplication and one sum, except the last call (**$n = 0$**) that just delivers the value 0 with 0 operations.
- Therefore, the number of multiplications and sums can be expressed by the recursive formula **$T(n)$** as the number of operations for a number of disks **n** :

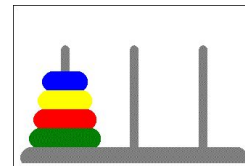
$$T(0) = 0 \quad T(n) = 1 + 2T(n - 1)$$



| | |
|--------------|------------------------|
| with 0 disks | you need 0 movements |
| with 1 disks | you need 1 movements |
| with 2 disks | you need 3 movements |
| with 3 disks | you need 7 movements |
| with 4 disks | you need 15 movements |
| with 5 disks | you need 31 movements |
| with 6 disks | you need 63 movements |
| with 7 disks | you need 127 movements |
| with 8 disks | you need 255 movements |
| with 9 disks | you need 511 movements |



Back-Substitution Method

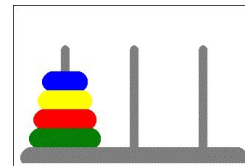


$$T(n) = 1 + 2T(n-1) \quad T(0) = 0$$

- We can rewrite the recursive relation replacing n :
 - $T(x) = 1 + 2T(x-1)$
- We can replace x by $n-1$:
 - $T(n-1) = 1 + 2T((n-1)-1)$
 - $T(n-1) = 1 + 2T(n-2)$
- Replacing $T(n-1)$ into the original recurrence relation:
 - $T(n) = 1 + 2T(n-1)$
 - $T(n) = 1 + 2(1 + 2T(n-2))$
 - $T(n) = 1 + 2 + 4T(n-2)$
- We can replace x by $n-2$:
 - $T(n-2) = 1 + 2T((n-2)-1)$
 - $T(n-2) = 1 + 2T(n-3)$
- Replacing $T(n-2)$ into the current recurrence relation:
 - $T(n) = 1 + 2 + 4T(n-2)$
 - $T(n) = 1 + 2 + 4(1 + 2T(n-3))$
 - $T(n) = 1 + 2 + 4 + 8T(n-3)$
- And we can keep performing substitution backwards...
 - $T(n) = 1 + 2 + 4 + 8 + 16T(n-4)$
 - $T(n) = 1 + 2 + 4 + 8 + 16 + 32T(n-5)$



Back-Substitution Method



$$T(0) = 0 \quad T(n) = 1 + 2T(n-1)$$

Substituting from **$n-k$** backwards we have:

- $T(n) = 1 + 2T(n-1)$
- $T(n) = 1 + 2 + 4T(n-2)$
- $T(n) = 1 + 2 + 4 + 8T(n-3)$
- $T(n) = 1 + 2 + 4 + 8 + 16T(n-4)$
- $T(n) = 1 + 2 + 4 + 8 + 16 + 32T(n-5)$

...

The pattern is

$$T(n) = 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} + 2^k T(n-k)$$

Since we know **$T(0) = 0$** , which value of **k** gives **$n-k$** equal to 0?

- **$k = n$**
 - $T(n) = 2^0 + 2^1 + \dots + 2^{n-1} + 2^n T(n-n)$
 - $T(n) = 2^0 + 2^1 + \dots + 2^{n-1} + 2^n T(0)$
 - $T(n) = 2^0 + 2^1 + \dots + 2^{n-1}$
 - $T(n) = 2^0 + 2^1 + \dots + 2^{n-1} = 2^n - 1$



MERRIMACK COLLEGE

Video: [Solution for the recurrence relation of Tower of Hanoi.](#)

$$O(2^n)$$

Complexity of Recursive Algorithms

To compute the complexity of recursive algorithms it is necessary to count the number of recursive calls.

There is basically two ways to do that.

One is tricky, but can be applied to any recursive algorithm.

The other is a straightforward method, but it cannot be applied to all recursive algorithms.

- Back-substitution method
 - Factorial
 - Towers of Hanoi
- **Master method**
 - Binary Search
 - Maximum Element in a array



Master Method - a.k.a. Master Theorem

- Another way to compute the complexity of recursive algorithms is the Master method, a deep mathematical theory behind the scenes provides us with a three-part rule for determining the asymptotic class of an algorithm whose workload is given by a recurrence relation of the form:



$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

The Master Method is not applicable to all recursions.



MERRIMACK COLLEGE

Wikipedia: [Master theorem \(analysis of algorithms\)](#).

Master Method - a.k.a. Master Theorem

- Encoding the recurrence relation in the form:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$



- The complexity is given according to three different situations:

$$\begin{array}{ll} \text{if } f(n) < n^{\log_b a} & \text{then } T(n) = O(n^{\log_b a}) \\ \text{if } f(n) = n^{\log_b a} & \text{then } T(n) = O(n^{\log_b a} \log n) \\ \text{if } f(n) > n^{\log_b a} & \text{then } T(n) = O(f(n)) \end{array}$$

Comparing the
complexity of
 $f(n)$ with **$\log_b a$**



Master Method - a.k.a. Master Theorem

- The **Binary Search example** basically performs two comparisons in each call, except for the last call that just delivers the single element index.
- Therefore, the number of comparisons can be expressed by the recursive formula **$T(n)$** for an array of size **n** that becomes half of it for each call:

$$T(n) = 2 + T\left(\frac{n}{2}\right) \quad T(1) = 1$$



```
1 def binSearch(A, start, end, k):
2     mid = (end+start)//2
3     if (start > end):
4         return None
5     elif (A[mid] == k):
6         return mid
7     elif (A[mid] > k):
8         return binSearch(A, start, mid-1, k)
9     else:
10        return binSearch(A, mid+1, end, k)
```



Master Method - a.k.a. Master Theorem

- We need to transform the recurrence relation in the proper format:

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

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

$$n^{\log_b a} = n^{\log_2 1} = n^0$$

- Becomes:

$$a = 1$$

$$b = 2$$

if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$

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

$$f(n) = 2n^0$$

$$T(n) = O(n^{\log_b a} \log n)$$

```
1 def binSearch(A, start, end, k):
2     mid = (end+start)//2
3     if (start > end):
4         return None
5     elif (A[mid] == k):
6         return mid
7     elif (A[mid] > k):
8         return binSearch(A, start, mid-1, k)
9     else:
10        return binSearch(A, mid+1, end, k)
```



Master Method - a.k.a. Master Theorem

- The **Maximum Element in an Array example** basically performs one comparison in each call, except for the last call that just delivers the single element index.
- Therefore, the number of comparisons can be expressed by the recursive formula **$T(n)$** for an array of size **n** that becomes half of it each call:

$$T(n) = 1 + 2T\left(\frac{n}{2}\right) \quad T(1) = 0$$

The stop condition is ignored by the Master Method.



```
1 def Max(A, start, end):
2     if (start == end):
3         return end
4     else:
5         mid = (end+start)//2
6         fst = Max(A, start, mid)
7         lst = Max(A, mid+1, end)
8         return fst if A[fst] > A[lst] else lst
9
10 from random import randint
11 A = [randint(0,100000) for _ in range(1000)]
12 i = Max(A, 0, len(A)-1)
13 print("The maximum number is", A[i], "at index", i)
```



Master Method - a.k.a. Master Theorem

- We need to transform the recurrence relation in the proper format:

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

- Becomes:

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

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

$$a = 2$$

$$b = 2$$

$$f(n) = 1n^0$$

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

→ if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$

$$T(n) = O(n^{\log_b a})$$

```
1 def Max(A, start, end):
2     if (start == end):
3         return end
4     else:
5         mid = (end+start)//2
6         fst = Max(A, start, mid)
7         lst = Max(A, mid+1, end)
8         return fst if A[fst] > A[lst] else lst
9
10 from random import randint
11 A = [randint(0,100000) for _ in range(1000)]
12 i = Max(A, 0, len(A)-1)
13 print("The maximum number is", A[i], "at index", i)
```



This Week's tasks

- In-class Exercise E#5
- Coding Project P#5
- Quiz Q#5

Tasks

- Fill the worksheet as required.
- Develop 2 Python programs:
 - number of digits in a binary expansion;
 - sum of squares of positive Integers.
- Quiz #5 about this week topics.



In-class Exercise - E#5

Download the pdf ([link here also](#)) and perform the following tasks:

- (1) apply back substitution to three examples;
- (2) apply the master method to five examples.

| CSC6013 - Worksheet for Week 5 | CSC6013 - Worksheet for Week 5 | CSC6013 - Worksheet for Week 5 |
|---|---|--|
| Back Substitution Compute the complexity of the recursive algorithms based on the recursive equation and stop condition. Show your work, not just your final answer. 1. $T(n) = 2T(n-1) + 1$ and $T(0) = 1$ a. You can compute this complexity as a tight upper bound. 2. $T(n) = T(n-2) + n^2$ and $T(0) = 1$ a. Hint: Assume n is even; that is, $n = 2k$ for some integer k . 3. $T(n) = T(n-1) + 1/n$ and $T(1) = 1$ a. Hint: Go online and find a formula for the sum of the first n terms of the "harmonic series". | Master Method Compute the complexity of the recursive algorithms based on the recursive equation and stop condition. Show your work, not just your final answer. 4. $T(n) = 2T(n/4) + 1$ and $T(0) = 1$ a. Be sure to rewrite 1 as n^0 . 5. $T(n) = 2T(n/4) + n^{1/2}$ and $T(0) = 1$ a. Note that $n^{1/2}$ is the square root of n . 6. $T(n) = 2T(n/4) + n^2$ and $T(0) = 1$ a. This is similar to the previous one. | Master Method Compute the complexity of the recursive algorithms based on the recursive equation and stop condition. Show your work, not just your final answer. 7. $T(n) = 10T(n/3) + n^2$ and $T(0) = 1$ a. In your answer, round the value of the logarithm to 2 decimal places. b. Remember that the $\log_3(a)$ is equal to $\log_2(a) / \log_2(3)$. 8. $T(n) = 2T(2n/3) + 1$ and $T(0) = 1$ a. In your answer, round the value of the logarithm to 2 decimal places. b. Be sure to rewrite 1 as n^0 . c. Remember that the $\log_3(a)$ is equal to $\log_2(a) / \log_2(3)$. d. Hint: rewrite $2n/3$ as $n/(3/2)$. |

You have to submit a **.pdf** file with your answers.

Feel free to type it or hand write it, but you have to submit a single **.pdf** with your answers.



MERRIMACK COLLEGE

This task counts towards the In-class Exercises grade and the deadline is This Friday.

Fifth Coding Project - P#5

1. **Develop a Python program** with a recursive algorithm to calculate the number of digits in the binary expansion/representation of a positive Integer **n** .
 - i. if **$n == 1$** , the answer is 1;
 - ii. if **$n > 1$** , the answer is 1 plus the number of digits in the binary representation of **$n//2$** .
- b. Run your code for the instances: **$n = 256$** and **$n = 750$** .
- c. Create the recurrence relation and stopping condition for your algorithm, and compute the Big Oh complexity using either back substitution or the master method.

To both programs you have to submit the code (**.py** file) of your algorithm and a **.pdf** with the two suggested executions, the recurrence relation/stop condition, and the Big Oh development and result (show your work, not only the result).



Fifth Coding Project - P#5

2. **Develop a Python program** with a recursive algorithm to calculate the sum of the squares of the positive Integers $1^2 + 2^2 + 3^2 + \dots + n^2$, given the value of n .
- i. if $n == 1$, the answer is 1;
 - ii. if $n > 1$, the answer is n^2 plus the sum of the squares up to $(n-1)$.
- b. Run your code for the instances: $n = 12$ and $n = 20$.
- c. Create the recurrence relation and stopping condition for your algorithm, and compute the Big Oh complexity using either back substitution or the master method.

To this program you have to submit the code (**.py** file) and a **.pdf** as for the first program (show your work, not only the result).

Your task:

Go to Canvas, and submit your **.py** files and **.pdf** files within the deadline.



MERRIMACK COLLEGE

This assignment counts towards the Projects grade and the deadline is Next Monday.

Fifth Quiz - Q#5

- The fifth quiz in this course covers the topics of Week 5;
- The quiz will be available this Friday, and it is composed by 10 questions;
- The quiz should be taken on Canvas (Module 5), and it is not a timed quiz:
 - You can take as long as you want to answer it (a quiz taken in less than one hour is usually a too short time);
- The quiz is open book, open notes, and you can even use any language Interpreter to answer it;
- Yet, the quiz is evaluated and you are allowed to submit it only once.

Your task:

- Go to Canvas, answer the quiz and submit it within the deadline.



MERRIMACK COLLEGE

This quiz counts towards the Quizzes grade and the deadline is Next Monday.

” **Welcome to CSC 6013**

- **Do In-class Exercise E#5 until Friday;**
- **Do Quiz Q#5 (available Friday) until next Monday;**
- **Do Coding Project P#5 until next Monday.**

Next Week - Decrease-and-conquer algorithms



MERRIMACK COLLEGE