CSC2710 Analysis of Algorithms — Fall 2022
Unit 3 - Recurrence Analysis

Book: §4.3 – 4.5

3.1 Mathematical Analysis of Recursive Algorithms

- We will take what we learned last week and look at how we apply this to the analysis of recursive algorithms.
- The first recursive algorithm your normally taught is how to compute $n!$.
- We define the $n!$ as:

$$n! = \begin{cases} 1, & n = 0 \\ n\,(n-1)!, & n \geq 1 \end{cases}$$

- We can formally write the algorithm as:

```
#Input: n >= 0
#Output: n! is returned
def Factorial( n ):
    if n == 0:       # base case          Line 1
        return 1                        # Line 2
    else:            # recursive case     Line 3
        return n * Factorial(n-1)       # Line 4
```

- We observe that n dictates how long it takes for the algorithm to run. As it dictates how times each line of the algorithm runs.
  - If you think about this a moment, it makes sense. A recursive algorithm is sort of like a loop.
- Lets count the operations.

Line	Cost	Count
1	c_1	t
2	c_2	1
3	c_3	$t-1$
4	c_4	$t-1$

- How do we determine the value for t?
  - Count the number of recursive calls!
- We will write our number of (additional) function calls for an input of size n as $C\,(n)$. In particular our number of calls is given by:

$$C\,(n) = C\,(n-1) + 1$$

  - This equation is called a *recurrence relation* or just *recurrence*.
- Our goal is to find a *closed form*. A closed form is an equation only in terms of n.
- There are infinitely many closed forms unless we specify an initial condition. In this case $C\,(0) = 0$.
  - This initial condition is sometimes called the *base case*.
- There is no general method that *always* works to find a closed form of a recurrence. So, how do we find the closed form of this recurrence?

- We could guess. This might work for the simpler recurrence relations.
- We instead will use a method called *back-substitution*

- In the back substitution method we express a recursive call in terms of its definition and then substitute the result in to the recurrence for $C(n)$.
  - For example, let's take computing $n!$. The recurrence is

$$C(n) = C(n-1) + 1.$$

$$\begin{aligned} C(n) &= C(n-1) + 1 \\ &= (C(n-2) + 1) + 1 && \text{(Substitute definition of } C(n-1)) \\ &= ((C(n-3) + 1) + 1) + 1 && \text{(Substitute definition of } C(n-2)) \\ &= \cdots \\ &= C(n-k) + k && \text{(Where } k \text{ is the number of calls)} \end{aligned}$$

  - Recall: Recursion stops when we reach the base case.
  - What value of k will cause the base case to be reached?
    - Answer: $k = n$ this implies,

$$C(n-k) + k = C(0) + n = n.$$

    - Therefore we say that our algorithm running time is in $\Theta(n)$.

- The basic analysis framework is modified slightly now for step (4) we must set up a recurrence relation and step (5) becomes finding the closed form of the recurrence relation.

3.1.1 Towers of Hanoi

- The Towers of Hanoi is a classic puzzle in Computer Science.
- The game goes as follows:

  Definition 1 (Towers of Hanoi Problem)**.**
  ***Input**: A board with three pegs p_1, p_2, and p_3 and a stack of n disks $\{d_1, d_2, \ldots, d_n\}$ stacked on the left most peg. They are stacked such that d_1 is the top disk and d_n is the bottom most disk. Moreover, d_1 is smallest disk and d_n is the largest. In general, disk d_j is smaller than all disks d_i for $i > j$.*
  ***Output**: A sequence of moves that move the disks from p_1 to p_3 obeying the rules:*

  1. *Only one disk can be moved at a time,*
  2. *Only the top most disk on any peg can be moved, and*
  3. *d_j can not be placed on top of d_i for any $i < j$. In other words, you can never place a smaller disk on top of a larger one.*

- **Practice**: How might we solve this for $n = 4$ disks?
- Let's look at a general algorithm for n disks.

  1. Recursively move $n - 1$ disks from p_1 to p_2 using p_3 as an auxiliary peg.
  2. Move the largest disk from p_1 to p_3.
  3. Recursively move $n - 1$ disks from p_2 to p_3 using p_1 as an auxiliary peg.

```
# Input: An integer n, src, aux, and tgt all in the set
#     of numbers from 0 to 2
# Output: The sequence of moves that moves all disks from src to tgt
def Hanoi(n, src, tgt, aux):
    # Base case: stop when we only have one disk to move
    if( n == 1 ):                                                    # Line 1
        print("Move disk 1 from peg ", src, " to peg ", tgt)         # Line 2
    else:                                                            # Line 3
        # Move n-1 disks from src to aux using tgt as scratch space
        Hanoi( n-1, src, aux, tgt )                                  # Line 4
        # Move disk n from src to tgt
        print("Move disk ", n, " from peg ", src, " to peg ", tgt)   # Line 5
        # Move the disks from aux to tgt
        Hanoi( n-1, aux, tgt, src )                                  # Line 6
```

- Notice that our algorithms runtime depends on the number of moves (calls) $C(n)$ we make for n disks.
- Let's set up that recurrence relation. The first call to HANOI kicks off the recurrence relation:

$$C(n) = \underbrace{C(n-1)}_{\text{disks from } p_1 \text{ to } p_2} + \underbrace{1}_{\text{move the disk}} + \underbrace{C(n-1)}_{\text{Move disks from } p_2 \text{ to } p_3}.$$

- We can simply this recurrence as:

$$C(n) = 2C(n-1) + 1.$$

- Whats our base case?
  - Answer: $C(1) = 1$.

- Let's look at using backwards substitution to find the closed form of this recurrence.

$$\begin{aligned}
C(n) &= 2C(n-1)+1 \\
&= 2(2C(n-2)+1)+1 && \text{(Substituting definition of } C(n-1)\text{)} \\
&= 2(2(2C(n-3)+1)+1)+1 \\
&= 2^3 C(n-3) + \underbrace{4+2+1}_{=2^3-1} && \text{(Applying algebra)} \\
&= \cdots \\
&= 2^k C(n-k) + \sum_{i=0}^{k-1} 2^i && \text{(Where } k \text{ is the number of disks processed)} \\
&= 2^k C(n-k) + 2^k - 1 && \text{(By Summation formula } \sum_{i=0}^{n} 2^i = 2^{n+1}-1\text{)}
\end{aligned}$$

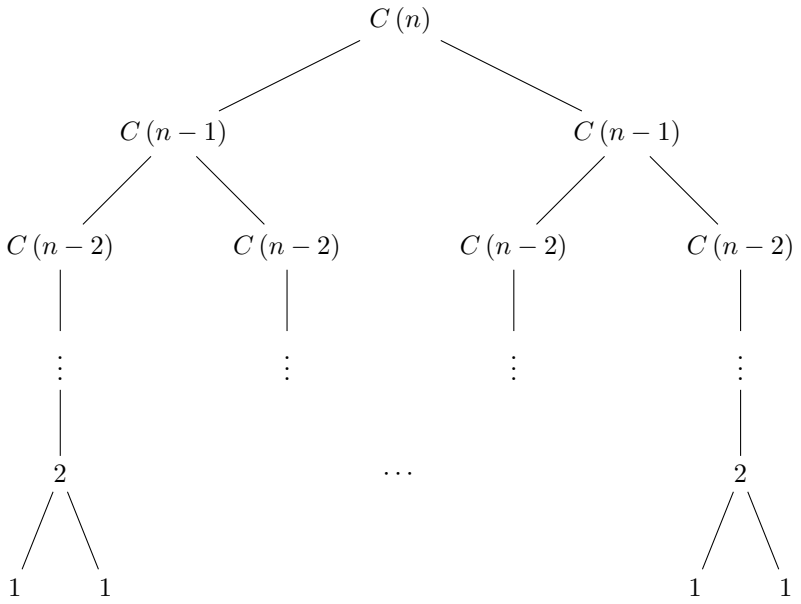
- What value of k gives us the base case?
  - Answer: $k = n-1$
- If we substitute $k = n-1$ into our recurrence we get

$$2^k C(n-k) + 2^k - 1 = 2^{n-1} C(1) + 2^{n-1} - 1 = 2^n - 1.$$

- Therefore our algorithms running time is in $\Theta(2^n)$.
- We can also express these recurrences as a tree. This might be helpful when you are trying to see a pattern.
  - For example, consider the recurrence

$$C(n) = 2C(n-1)+1.$$

We have the tree:



3.2 The Master Method

- The method we just discussed *always* works. However, most common recurrence relations we will encounter actually can be solved using a simple theorem called the *Master Theorem*.
  - The master theorem works on a recurrence relation with the general form

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

  - Formally, the theorem is:

    Theorem 1 (Master Theorem[1]). *If* $f(n) \in \Theta\left(n^d\right)$ *where* $d \geq 0$, *then*

$$T(n) \in \begin{cases} \Theta\left(n^d\right), & \text{if } a < b^d \\ \Theta\left(n^d \lg n\right), & \text{if } a = b^d \\ \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \end{cases}$$

- Given $A(n) = 2A\left(\frac{n}{2}\right) + 1$, $f(n) = 1$ so $d = 0$, $a = 2$ and $b = 2$. We know that $2 > 2^0$ so we have $\Theta\left(n^{\log_2 2}\right) = \Theta(n)$.
  - Ahhh...wasn't that much easier!

[1] This is a simplification of the Master Theorem. Theorem 4.1 in section 4.5 of CLRS is more precise but, harder to work with. This definition will work just as well in practice.

Challenge Problem: Solving Recurrences

For each of the following, assume $T(1) = 1$.

- What is the closed form of the recurrence $T(n) = T(n-1) + n$? What is its running time?
- What is the closed form of the recurrence $T(n) = 3 * T(n/2) + 2$? What is its running time? **Please note:** You cannot use the Master Method for this running time analysis, as it falls into one of the "gaps" between the (simpler) method described in these lecture notes and the method descibed in the textbook, as mentioned in the footnote above.
- What is the running time of the recurrence $T(n) = 2 * T(n/2) + (n/2)$. You *can* use the Master Method for this problem.

Challenge Problem: Counting Binary Numbers

- Recall binary digit counting problem is formally defined as:

  Definition 2 (Binary Digit Counting Problem)**.**
  Input*: An integer $n \in \mathbb{Z}^+$.*
  Output*: The b the number of bits in the binary expansion of n.*

- How might we solve this recursively?
  - We could just convert our loop to a recursive statement.
  - Recall, that loops and recursion can be shown to be equivalent. One can always convert a recursive algorithm to an algorithm based on loops and vice versa.
- The formal algorithm is:

```
import math

# Input: n is a positive integer
# Output: The number of bits in the binary expansion of n
def BinaryDigitCountRecursive( n ):
    if n == 1:                                                    # Line 1
        return 1                                                  # Line 2
    else:                                                         # Line 3
        return BinaryDigitCountRecursive( math.floor(n/2) ) + 1   # Line 4
```

- Use a recurrence relation to determine a *closed form solution* to what the algorithm computes (not its running time)
- What is the running time (in terms of Big-Oh, or Big-Theta) of this algorithm?

Challenge Problem: Printing Pairs

- Write a recursive algorithm to print all pairs in an array of numbers.
  - If $A = [3, 4, 1, 6]$, then the algorithm should output:
    $[3, 4]$
    $[3, 1]$
    $[3, 6]$
    $[4, 1]$
    $[4, 6]$
    $[1, 6]$
- Determine the running time of your algorithm, showing all work.

Challenge Problem: Generating the Power Set

- The **power set** of a set of numbers is the set of all subsets of the set.

  Definition 3 (Power Set Problem)**.**
  Input*: A set A of n integers.*
  Output*: The set of all subsets of A.*

- How many subsets does a set of size n have?
- Create a recursive algorithm to generate the power set of A. **Hint:** Store the power set as a set of sets, and generate each set individually.
  - Assume that Python functions are passed by value (copies are made when passing)
  - You can designate something as **Persistent** (or globally accessible) using another section after your **Output** before the algorithm starts. Remember to also use the Python keyword `global`
  - You can also assume you have access to the suite of all `Set` operations (`add`, `remove`, `union`, `intersection`, etc.)
- Test your solution by converting it to code and testing it out.
- Analyze the running time of your algorithm. If you used any of Python's built in functions, be sure to look up their running times during your analysis.

Challenge Problem: Largest Sum Submatrix

- In this problem, we will determine and analyze an algorithm for finding the square submatrix with the largest sum.

  Definition 4 (Largest Sum Submatrix)**.**
  Input*: An $n \times n$ matrix M*
  Output*: The sum of the elements of the square submatrix (a mtimesm sub matrix within M) that produces the largest sum.*

- There are many square submatrixes of a matrix, of varying sizes, from 1×1 to $n \times n$ and everything in between.
- This problem tasks you with finding the square submatrix that, when all the elements are added together, produces the largest sum. *it might be the entirety of M* but it is not guaranteed to be (why?).
- Determine a recursive algorithm for solving this problem and analyze its runtime. *If you feel comfortable* try doing this without writing the algorithm out formally, but simply describing it.