CSE 2321: Notes on Recurrence Relations 1

1. **Tower of Hanoi Puzzle**

   In class, we looked at the Tower of Hanoi problem. See the link below:
   https://www.mathsisfun.com/games/towerofhanoi.html

   The Tower of Hanoi problem can be solved recursively: To move a stack of n disks to tower 3, first move a stack of $n-1$ disks to tower 2, then move the bottom disk to tower 3, and finally move the stack of $n-1$ disks to tower 3.

   The number of total moves M needed to move a stack of n disks can be modeled with a recurrence relation, as shown below:

   $$M(1) = 1, \text{ (base case)}$$

   $$M(n) = 2M(n-1) + 1, \text{ if } n > 1$$

   Below, we find a closed form for the recurrence relation, using the "chug-and-plug" method.

$$\begin{aligned}
(1)\ M(n) &= 2M(n-1) + 1 \\
&= 2[2M(n-2) + 1] + 1 \\
(2)\ &= 2^2 M(n-2) + 2 + 1 \\
&= 2^2[2M(n-3) + 1] + 2 + 1 \\
(3)\ &= 2^3 T(n-3) + 2^2 + 2 + 1 \\
&= 2^3[2T(n-4) + 1] + 2^2 + 2 + 1 \\
(4)\ &= 2^4 T(n-4) + 2^3 + 2^2 + 2 + 1 \\
&\cdots \\
(k)\ &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1
\end{aligned}$$

   The process stops when $n - k = 1$ or $k = n - 1$. Plugging this into the last equation above gives us,

$$\begin{aligned}
T(n) &= 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1
\end{aligned}$$

   This is a geometric series with $r > 1$, so asymptotically it acts like its largest term; therefore it is:

   $$\Theta(2^n)$$

2. **Recursive Binary Search**

The code for the recursive version of binary search is shown below.

```
#nums is a list of integers sorted in ascending order
#key is an integer to search for in nums
#lo is the leftmost index to search from in nums
#hi is the rightmost index to search to in nums

#the function returns True if key is in nums, False otherwise

def binary_search_helper(nums, key, lo, hi):
  if lo > hi:
    return False
  else:
    mid = (lo + hi) // 2
    if nums[mid] == key:
      return True
    elif nums[mid] < key:
      return binary_search_helper(nums, key, mid + 1, hi)
    else:
      return binary_search_helper(nums, key, lo, hi - 1)

def binary_search(nums, key):
  return binary_search_helper(nums, key, 0, len(nums) - 1)
```

The following is the recurrence for binary search for the worst-case running time of binary search. It takes constant time to search a list with one element. If there is more than one element in the list, it takes constant time to calculate the middle index and compare the key to the value at the middle index, and it takes time $T(n/2)$ to search the remaining half of the list.

$$T(1) = c_1$$
$$T(n) = T\left(n/2\right) + c$$

Below, we find a closed form for the recurrence.

$$\begin{aligned}
(1)\ T(n) &= T\left(n/2\right) + c \\
&= T\left(n/2^2\right) + c + c \\
(2)\ &= T\left(n/2^2\right) + 2c \\
&= T\left(n/2^3\right) + c + c + c \\
(3)\ &= T\left(n/2^3\right) + 3c \\
&\cdots \\
(k)\ &= T\left(n/2^k\right) + kc
\end{aligned}$$

The process stops when $(n/2^k) = 1$ or $k = \log_2(n)$, so

$$T(n) = T\left(n/2^k\right) + kc = T\left(1\right) + c\log_2(n) = c_1 + c\log_2(n) = \Theta(\log(n))$$