Connor Jensen

Written Assignment 01

February 24, 2024

P1. (10 pts) Proof by Induction

- Let F_1 be the Fibonacci numbers as defined in Section 1.2

  a. Assume: F_0=1, F_1=1, and $F_n = F_{n-1} + F_{n-2}$

Prove: $\sum_{i=1}^{N-2} \mathbf{F_i} = \boldsymbol{F_N - 2}$

1. **Base Case** $(\mathbf{N = 3})$: $\sum_{i=1}^{3-2} \boldsymbol{F_i} = \boldsymbol{F_3 - 2}$
2. **Left Side**: $\sum_{i=1}^{3-2} \boldsymbol{F_i} = \sum_{i=1}^{1} \mathbf{F_i} = (\boldsymbol{F_1}) = \mathbf{1}$
3. **Right Side**: $\boldsymbol{F_3 - 2 = F_2 + F_1 - 2 = (F_1 + F_0) + 1 - 2 = (2) - 1 = 1}$

Base Case is True because both the left and right side are equal to 1 when N = 3

a. Assumptions for Inductive Case (N = N + 1):

   a. N = (D + 2) => (N + 1) = D + 3
   b. $\sum_{i=1}^{N-2} F_i = \sum_{i=1}^{D} F_i = \boldsymbol{\sum_{i=1}^{D+1} F_i}$
   c. $F_N - 2 = F_{D+2} - 2 = \boldsymbol{F_{D+3} - 2}$
   d. $\boldsymbol{F_{n+1} + F_{n+2} = F_{n+3}}$

1. **Inductive Form**: $\boldsymbol{\sum_{i=1}^{D+1} F_i = F_{D+3} - 2}$

   a. $\boldsymbol{\sum_{i=1}^{D+1} F_i = \sum_{i=1}^{D} F_i + F_{D+1} = F_{D+1} + (F_{D+2} - 2) = F_{D+3} - 2}$

Inductive Case is True because both sides are equal to $\boldsymbol{F_{D+3} - 2}$ when $\boldsymbol{N = N + 1 = D - 1}$

- For N>= 1,

Prove: $\sum_{i=1}^{N}(2i - 1) = N^2$

1. **Base Case** $(\mathbf{N} = \mathbf{1})$: $\sum_{i=1}^{1}(2i - 1) = 1^2$

   2. **Left Side**: $\sum_{i=1}^{1}(2i - 1) = 2 * 1 - 1 = 1$

   3. **Right Side**: $1^2 = 1$

<u>Base Case is True because both the left and right side are equal to 1 when N = 1</u>

a. Assumptions for Inductive Case:

   a. $\sum_{i=1}^{N}(2i - 1) = N^2$

   b. $\sum_{i=1}^{N}(2i - 1) = \sum_{i=1}^{N+1}(2i - 1)$

   c. $N^2 = (N + 1)^2$

   d. $(N + 1)^2 = N^2 + 2N + 1$

1. **Inductive Form**: $\sum_{i=1}^{N+1}(2i - 1) = (N + 1)^2$

   a. $\sum_{i=1}^{N+1}(2i - 1) = \sum_{i=1}^{N}(2i - 1) + (2(N + 1) - 1) = N^2 + 2N + 1$

<u>Inductive Case is True because both sides are equal to $(N + 1)^2$ when $N = (N + 1)$</u>

P2. (5 pts) (5 pts) An algorithm takes 1 ms for input size 100. How long will it take for input size 800 if the running time is the following (assume low-order terms are negligible):

- Linear

  A: 8 ms
- O (N $\log_2$ N)

  A: 24 ms
- Quadratic

  A: 64 ms
- Cubic

  A: 512 ms

P3. (10 pts) For each of the following program fragments, give an analysis of the running time (using Big-Theta)

```
i)
def fun1(n):
  i = 1                 # C
  sumA = 0              # C
  while i < n * n:      # While loop = O(N^2)
    sumA += 1
    i += 3
```

A: Using Big-O analysis, the while loop uses (N^2 / 3) processes to complete; so, the Big-Theta notation is equal to **O(N^2)**

```
ii)
def fun2(n):
  i = 1                 # C
  sumB = 0              # C
  while i < n * n:      # While loop = O(log (N))
    sumB += 1         # C
    i *= 3            # C
```

A: Using Big-O analysis, the while loop uses (N^2 / i^2) processes to complete; so, the Big-Theta notation is equal to **O(log N)**

```
iii)
def fun3(n):
  sumC = 0                        # C
  for i in range(n):              # N
    for j in range(i*2, n**3):      # For loop = O(n**3/2i)
      for k in range(j):              # For loop = O(n**3/2i)
        sumC += 1                         # C
```

<u>A: Using Big-O analysis, the first for-loop uses N process to complete, while the subsequenct two for-loops each use (N^3/2i) processes to complete; so, the Big-Theta notation is equal to **O(N^7)**</u>

```
iv)
def fun4(n):
  sumD = 0                        # C
  for i in range(n):              # N
    for j in range(i*2, n**3):  # N**3/2i
      if j < i:               # Never happens: N
        for k in range(j):  # Never happens: N**3
          sumD += 1       # C
```

<u>A: Using Big-O analysis, the first for loop uses (N) processes to complete, the second for loop uses (N**3) processes, and the if statement never is true, so the nested loop inside will never be reached; so, the Big-Theta notation is equal to **O(N^4)**</u>

```
v)
def fun5():
  k = 0                   # C
  n = 5                   # C
  if n > 10:              # C
    k = n
  else:                       # C
    for i in range(n):
      for j in range(n):
        k += 1
```

<u>A: Using Big-O analysis since all variables in loops are stationary and thus cannot increase nor decrease, the amount of processes is constant, meaning the big-Theta notation is **O(1)**</u>

P4. (10 pts) What is the asymptotic complexity of the following functions? Justify your answer.

i) (3 pts)

```
def fun1(n):
  i = 0
  if (n > 1):
    fun1(n - 1)
  for i in range(n):
    print(" * ",end="")
```

Recurrence Relation:

$$\underline{\boldsymbol{T(N) = T(N-1) + O(N) + C}}$$

Complexity in Big-Oh: Show your process of finding the complexity from the recurrence relation.

Using Master's Theorem, 1 = 1, therefore, the complexity in Big-Oh is N*log N

ii) (3 pts)

```
def fun(a, b):
if (b == 0):
return 1
if (b % 2 == 0):
return fun(a*a, b//2)
return fun(a*a, b//2)*a
```

Recurrence Relation:

$T(N) = 2T(N/2) + O(N^2) + C$

Complexity in Big-Oh: Show your process of finding the complexity from the recurrence relation.

Using Masters Theorem, $2 < 2^2$, therefore, the complexity in Big-Oh is N^2

iii) (4 pts)

```
def func(n,start,end,aux):
      if(n==1):
      print("Move disk 1 from",start,"to",end)
      return
func(n-1,start,aux,end)
print("Move disk",n,"from",start,"to",end)
func(n-1,aux,end,start)
```

Recurrence Relation:

$T(N) = 2T(N - 1) + O(2^n) + C$

Complexity in Big-Oh: Show your process of finding the complexity from the recurrence relation.

2^n is the biggest term in the Recurrence Relation, and thus will be the Big-Theta: 2^n