EE4371 - Assignment 2

Om Shri Prasath, EE17B113

Q1. Order the following functions by asymptotic growth rate.

- $4n\log(n)+2n$
- 2^{10}
- $2^{\log(n)}$
- $3n+100\log(n)$
- $4n$
- 2^n
- n^2+10n
- n^3
- $n\log(n)$

Solution :

The order of asymptotic growth rate for general functions are given by following thumb rules :

1. $a\cdot f(n) < b\cdot f(n)$ for $a<b$ where a,b are constants
2. constant < logarithmic < polynomial < exponential is the general order
3. Only the higher order terms needs to be considered

Also $2^{\log n}=n$ since the $\log$ here is of base 2

Using the above rules, we get :

$$\boxed{2^{10} < 2^{\log(n)} < 3n+100\log(n) < 4n < n\log(n) < 4n\log(n)+2n < n^2+10n < n^3 < 2^n}$$

Q2. In each of the following situations, indicate whether $f=O(g)$, or $f=\Omega(g)$, or both (in which case $f=\Theta(g)$). Justify your answer.

	$f(n)$	$g(n)$
(i)	$n-100$	$n-200$
(ii)	$\log 2n$	$\log 3n$
(iii)	$n^{0.1}$	$(\log n)^{10}$
(iv)	$n2^n$	3^n

Solution :

The definition for the relations between $f(n)$ and $g(n)$ are as follows :

$$f(n)=O(g(n)) \iff \exists\, c,n_o>0 \text{ s.t. } \forall\, n\geq n_o, f(n)\leq c\cdot g(n)$$

$$f(n)=\Omega(g(n)) \iff \exists\, c,n_o>0 \text{ s.t. } \forall\, n\geq n_o, f(n)\geq c\cdot g(n)$$

$$f(n)=\Theta(g(n)) \iff \exists\, c_1,c_2,n_o>0 \text{ s.t. } \forall\, n\geq n_o,\ \ c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$$

We will use the above definitions for deciding the relation between $f(n)$ and $g(n)$.

(i) $f(n)=n-100, g(n)=n-200$

We see that $\forall\, n\in\mathbb{R}, n-200\leq n-100 \implies f(n)=\Omega(g(n))$

We see that for the inequality, $n-100\leq c(n-200)$ to be satisfied,

$(c-1)n-200c+100\geq 0$. Let c = 2 $\implies n-300\geq 0 \implies n\geq 300$

For $n_o=300, c=2$, $f(n)\leq c\cdot g(n) \implies f(n)=O(g(n))$

Thus, for $n_0=300, c_1=1, c_2=2, c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$

The conditions for Θ is satisfied. Thus $\boxed{f=\Theta(g)}$

(ii) $f(n)=\log 2n, g(n)=\log 3n$

We see that $\forall\, n>0, \log 2n\leq\log 3n \implies f(n)=O(g(n))$

We see that for the inequality, $\log 2n\geq c\log 3n$ to be satisfied

$(1-c)\log n+\log 2-c\log 3\geq 0$

Let $c=0.5 \implies 0.5\log n+\log\frac{2}{\sqrt{3}}\geq 0 \implies n\geq\frac{3}{4}$

For $n_o=\frac{3}{4}, c=0.5$, $f(n)\geq c\cdot g(n) \implies f(n)=\Omega(g(n))$

Thus, for $n_o=\frac{3}{4}, c_1=0.5, c_2=1$, $c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$

The conditions for Θ is satisfied. Thus $\boxed{f=\Theta(g)}$

(iii) $f(n)=n^{0.1}, g(n)=(\log n)^{10}$

Let $n_o=2^{1000}, c=1$, we see that $f(n)\geq c\cdot g(n)\,\forall\, n\geq n_o$

Also, $\lim_{n\to\infty}\frac{n^{0.1}}{(\log n)^{10}}=\infty$, which implies that the value of $n^{0.1}$ is greater than $(\log n)^{10}$ asymptotically.

Thus the conditions for Ω is satisfied. Also, there is no n_o, c to satisfy the condition for O. Thus $\boxed{f=\Omega(g)}$

(iv) $f(n)=n2^n, g(n)=3^n$

Let $n_o=3, c=1$, we see that $f(n)\leq c\cdot g(n)\,\forall\, n\geq n_o$

Also, $\lim_{n\to\infty} n\left(\frac{2}{3}\right)^n=0$, which implies the value of $n2^n$ is greater than 3^n asymptotically.

Thus the conditions for O is satisfied. Also, there is no n_o,c to satisfy the conditions for Ω. Thus $\boxed{f=O(g)}$

Q3. Describe an efficient algorithm for finding the ten largest elements in a sequence of size *n*. What is the running time of your algorithm?

Solution : $O(n)$ Time Complexity

We will use the following algorithm for finding the **k** largest elements in an array. For k = 10, we will get the solution for the above question.

Let the array `arr` be of length **n(>=k)**

- We first find the (n-k)th largest element in the array, which is found using the Random QuickSelect algorithm as follows :
  - We first select a *pivot* in the given array randomly which is between two given indices *l* and *r*
  - We push elements which are less than the *pivot* element to the left of it and the elements which are greater than the *pivot* element to the right of it.
  - After the above operation, the *pivot* element will be in the correct position of the array. Thus, we can do the following based on the position of *pivot*
    - If *pivot* position is equal to (n-k), we return that element.
    - If *pivot* position is greater than (n-k), we recursively search in the left part of the array (*l*,*pivot-1*)
    - If *pivot* position is less than (n-k), we recursively search in the right part of the array (*pivot+1*,*r*)
  - The time complexity of the above algorithm is $O(n)$ on average. The proof is given as follows :
    - Assuming we split the array into half on average (due to the random pivot select), we see that the size of the array we need to compute on reduces by half in each step. The maximum number of steps we take is $\log n$ where n is the size of the array.
    - Thus the time taken would be as follows : (Note : $2^{\log n}=n$)

$$T(n)=(n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}\ldots)_{\log n\text{ times}}=n\left(\frac{1-\left(\frac{1}{2^{\log n}}\right)}{1-\frac{1}{2}}\right)=2n\left(1-\frac{1}{n}\right)=2n-2=O(n)$$

- The average time complexity is $O(n)$ assuming we split the array equally on average due to the random pivot selection. But the worst-case time complexity is $O(n^2)$ which happens when we repeatedly select the least or greatest element of the subarray as pivot.
- After finding the (n-k)th largest element in the array, we again iterate through the array and take only elements which are greater than or equal to the (n-k)th array. This gives us the required k greatest numbers in the array. This operation is $O(n)$ time complexity.

Thus the overall time complexity of the algorithm is $O(n)+O(n)=O(n)$ time complexity on average.

In [1]:

```python
"""
Python code to get the 10 largest elements from an array
"""

# Importing random library to randomly select pivot
from random import randint

"""
Function to sort the array around a random pivot

Input -> arr - Array, l - Left Index, r - Right Index
Output -> Array which is sorted around pivot and the pivot index
"""


def randPartition(arr, l, r):
    # Selecting the random pivot
    m = randint(l, r)

    # Swap the pivot to the end for easier sorting
    arr[r], arr[m] = arr[m], arr[r]

    # Variable to store current swap position
    i = l

    # Pivot element value
    x = arr[r]

    # Loop to sort the array around the pivot
    for j in range(l, r):
        # If element is smaller than pivot,
        if arr[j] <= x:
            # Swap element to ith position
            arr[i], arr[j] = arr[j], arr[i]
            # Increment i
            i += 1
    # Swap the pivot to the correct position
    arr[r], arr[i] = arr[i], arr[r]

    # Return the pivot position
    return i


"""
Function to find the kth largest element

Input -> arr - Array, l - Left Index, r - Right Index, k
Output -> k th largest element
"""


def kthLargestElement(arr, l, r, k):

    # Check if k is less than size of array
    if k > 0 and k <= r - l + 1:

        # Apply randomPartition and get the pivot position
        p = randPartition(arr, l, r)

        # Compare pivot position to k and act accordingly
        if p - l == k - 1:
            return arr[p]
        elif p - l < k - 1:
            return kthLargestElement(arr, p + 1, r, k - p + l - 1)
        else:
            return kthLargestElement(arr, l, p - 1, k)


"""
Function to return k largest element

Input -> arr - Array, k
Output -> k largest elements
"""


def kLargestElements(arr, k):

    # Get the (n-k)th largest element
    k_l = kthLargestElement(arr, 0, len(arr) - 1, len(arr) - k + 1)

    # Variable to store answer
    ans = []

    for i in arr:
        # Store values in answer which are >= (n-k)th element
        if i >= k_l:
            ans.append(i)

    return ans


# Checking the function
arr = [3, 2, 1, 4, 5, 9, 8, 7, 10, 6, 11, 12, 15, 14, 13]

print("10 largest elements :", kLargestElements(arr, 10))
```

```
10 largest elements : [6, 7, 8, 10, 13, 11, 12, 15, 14, 9]
```

Q4. Use the divide and conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

Solution : $O(n^{1.6})$ Time Complexity

We will be using the Karatsuba divide and conquer algorithm for multiplication.

- Given two binary numbers x and y to be multiplied, we will write x as $2^{(n/2)}a+b$ and y as $2^{(n/2)}c+d$
- Now we can write $x\cdot y$ as $2^n(a\cdot c)+2^n(a\cdot c)+2^{(n/2)}(a\cdot d+b\cdot c)+b\cdot d$
- Now we recursively call the function for the products $a\cdot c$, $a\cdot d$, $(a+b)\cdot(c+d)$
- The term $a\cdot d+b\cdot c$ can be calculated as $(a+b)\cdot(c+d)-a\cdot c-b\cdot d$
- The stopping condition is when we have 1 digit products, where we just directly return the result

The time complexity is given as follows : We do 3^k steps at each level, and the last level is of size $\log n$. Thus the final time complexity is $3^{\log n}=n^{\log 3}\approx O(n^{1.6})$

In [2]:

```python
"""
Function to multiply two binary numbers


Input -> Two binary numbers x,y
Output -> Binary number which is the product of x and y

"""


def binary_multiply(x, y):

    # Get the length of the array
    n_x = len(x)
    n_y = len(y)

    n = max(n_x, n_y)

    # To make the length of the arrays same

    x = "0" * (n - n_x) + x
    y = "0" * (n - n_y) + y

    # If the elements are of size 1, we directly return the result
    if n == 1:

        if x == "1" and y == "1":
            return "1"
        else:
            return "0"

    # Making the length of operands even to make implementation easier
    while n % 2:
        x = "0" + x
        y = "0" + y
        n += 1

    # Calculating a,b,c,d

    a = x[: n // 2]
    b = x[n // 2 :]

    c = y[: n // 2]
    d = y[n // 2 :]

    # Calculating a+b and c+d
    a_b = bin(int(a, 2) + int(b, 2))[2:]
    c_d = bin(int(c, 2) + int(d, 2))[2:]

    # Recursively calculating ac,(a+b)(c+d),bd
    ac = binary_multiply(a, c)
    ab_cd = binary_multiply(a_b, c_d)
    bd = binary_multiply(b, d)

    # Calculating the product x*y using ac,ad,bc,bd
    ans = bin(
        (int(ac, 2) << n)
        + ((int(ab_cd, 2) - int(ac, 2) - int(bd, 2)) << n // 2)
        + int(bd, 2)
    )[2:]
    return ans


# Checking the function
x = "10011011"
y = "10111010"

x_int = int(x, 2)
y_int = int(y, 2)

print("General Multiplication Result\t:", bin(x_int * y_int)[2:])
print("Multiplication using D&C\t:", (binary_multiply(x, y)))
```

```
General Multiplication Result   : 111000010011110
Multiplication using D&C        : 111000010011110
```

Q5. You are given a unimodal array of n distinct elements, meaning that its entries are in increasing order up until its maximum elements, after which its elements are in decreasing order. Give an algorithm to compute the maximum element of a unimodal array that runs in $O(\log n)$ time.

Solution : $O(\log n)$ Time Complexity

We will use binary search method to find the maximum element of a unimodal array as follows :

1. We define two pointers `l = 0`, `r = n-1` where `n` = length of the array
2. Until `l <= r`, we calculate `mid = (l+r)//2`
   A. If the element at mid is greater than both its nearby elements, then the element at m is maximum
   B. If the element at mid is lesser than the right element, then we shift `l = mid+1`
   C. If the element at mid is greater than the right element, then we shift `r = mid-1`
   D. If `l==r`, we return the element at l
   E. If `l==r-1`, we return max of the element at l or r

This algorithm runs in $O(\log n)$ time, since the array needed to be searched is cut into half at every step.

In [3]:

```python
"""
Function to find maximum number in an unimodal array


Input -> Unimodal Array arr
Output -> Maximum number in array

"""


def max_unimodal(arr):

    # Get the length of the array
    n = len(arr)

    # Define l and r
    l = 0
    r = n - 1

    # Loop to apply the above algorithm
    while l < r:

        if l == r - 1:
            return max(arr[l], arr[r])
        mid = (l + r) // 2

        if arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1]:
            return arr[mid]

        if arr[mid] < arr[mid - 1] and arr[mid] > arr[mid + 1]:
            r = mid - 1
        else:
            l = mid + 1

    if l == r:
        return arr[l]


a = max_unimodal([1, 2, 3, 4, 5, 2, 1, 0, -1, -2, -3, -4])

print("Largest element =", a)
```

```
Largest element = 5
```

Q6. Towers of Hanoi : Given a game board with three pegs and a set of disks of different diameter all stacked from smallest to largest on the leftmost peg, moves all of the disks to the rightmost peg following these two rules. First, only one disk may be moved at a time. Second, a larger diameter disk may never be placed on a smaller disk. Any number of disks can be used. Implement this in Python.

Solution : $O(2^n)$ Time Complexity

We follow the steps as follows :

1. We shift the n-1 disks to the middle peg
2. We shift the largest disk to the last peg
3. We then shift back the n-1 disks to the last peg

This is a recursive method where for solving for n-1, we call the solution for n-2 and so on. This makes the solution $O(2^n)$

In [10]:

```python
"""
Python code to solve Tower of Hanoi
"""

from copy import deepcopy


"""
Function to draw the current position of disks

Input -> disks - Array containing the disks in each pegs
Output -> Print the position of disks in pegs
"""


def draw(disks):
    # Variable holding number of disks used
    global num_disks

    # Loop to print either a disk or a bar (|)
    for j in range(num_disks - 1, -1, -1):
        for i in range(3):
            if len(disks[i]) < j + 1:
                print("|", end="\t")
            else:
                print(disks[i][j], end="\t")
        print()
    print("\n")


"""
Function to solve Tower of Hanoi for given number of disks

Input -> disks - Array containing the disks in each pegs
         n - Number of disks
         start_peg - Index pointing to start peg
         end_peg - Index pointing to end peg
         temp_peg - Peg used to shift n-1 disks

Output -> Print the position of disks in pegs
"""


def solve_hanoi(disks, n, start_peg, end_peg, temp_peg):

    # If only one disk remains we push it to end peg
    if n == 1:
        disks[end_peg].append(disks[start_peg].pop(-1))
        draw(disks)
        return

    # Else, we first shift n-1 disks to temp peg
    solve_hanoi(disks, n - 1, start_peg, temp_peg, end_peg)

    # We then shift the remaining disk to the end peg
    disks[end_peg].append(disks[start_peg].pop(-1))
    draw(disks)

    # We then shift the n-1 disks from temp to the end peg
    solve_hanoi(disks, n - 1, temp_peg, end_peg, start_peg)


"""
Function to setup the Tower of Hanoi and solve it

Input -> n - Number of disks
Output -> Solves tower of hanoi and displays each step
"""


def hanoi(n):

    # Creating the array of disks to represent pegs
    disks = [[i for i in range(n, 0, -1)], [], []]
    draw(disks)
    print("Starting Solve for n=", n, "\n")
    # Solving the Tower of Hanoi
    solve_hanoi(disks, n, 0, 2, 1)


# Checking the code
num_disks = int(input("Enter the number of disks :"))
print()
# Using deepcopy so that num_disks is not affected in recursion
hanoi(deepcopy(num_disks))
```

```
Enter the number of disks :3

1       |       |
2       |       |
3       |       |


Starting Solve for n= 3

|       |       |
2       |       |
3       |       1


|       |       |
|       |       |
3       2       1


|       |       |
|       1       |
3       2       |


|       |       |
|       1       |
|       2       3


|       |       |
|       |       |
1       2       3


|       |       |
|       |       2
1       |       3


|       |       1
|       |       2
|       |       3
```