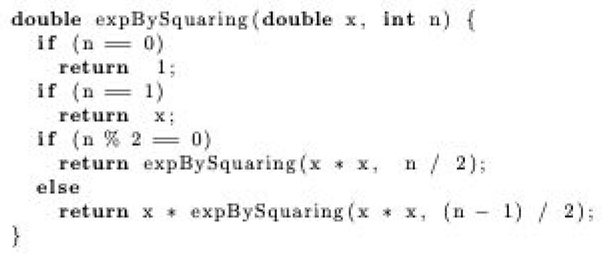
Name ;- Kiran Bagwe UNCC ID :- 801223392
Name :- MUkesh Dasari UNCC)D :- 801208218

Extracting dependency from code is an almost automatic process. You need to choose a granularity. But once that is chosen, the entire analysis follows. In the whole activity, you should express the metrics in complexity notation as a function of the parameters of the functions.

1 Fast Exponentiation

Consider this function to compute x n where n is a positive integer.



```
double expBySquaring(double x, int n) {
  if (n == 0)
    return  1;
  if (n == 1)
    return  x;
  if (n % 2 == 0)
    return expBySquaring(x * x,  n / 2);
  else
    return x * expBySquaring(x * x, (n - 1) / 2);
}
```

Question: What is the complexity of this function?
Ans: The final answer will be returned when n becomes 0 or 1. And every time the value of n is updated to its half so the complexity of the problem becomes O(log(n)).

Question: Extract the dependencies.
Ans: In this algorithm we have traced the recursion using example values as x=6 and n=6 and which should return 5^6. Below is the DAG of the algorithm. In every recursion call value of x is squared and n is halved.

$$\begin{tikzcd}
\text{expBySquaring(x=5, n=6) - Pre} \arrow[d, Rightarrow, "P\_1"] \\
\text{expBySquaring(x=25, n=3) - Pre} \arrow[d, Rightarrow, "P\_11"] \\
\text{expBySquaring(x=625, n=1) - Pre} \arrow[d, Rightarrow, "R\_11"] \\
\text{expBySquaring(x=25, 3) - Post} \arrow[d, Rightarrow, "R\_1"] \\
\text{expBySquaring(x=5, n=6) - Post}
\end{tikzcd}$$

Question: What is the width?
Ans: The width is n/2 because in every recursion function it is not dependent on any shared variable so each function call can be considered as an independent task.

Question: What is the work?
Ans: Theta(epxBySquaring(x,n))

Question: What is the critical path? What is its length?
Ans: The critical path is from P_1 and R_1 and the length will be work done or sum of all the processing time taken by each process.

2. Dense Matrix Matrix Multiplication Recursively

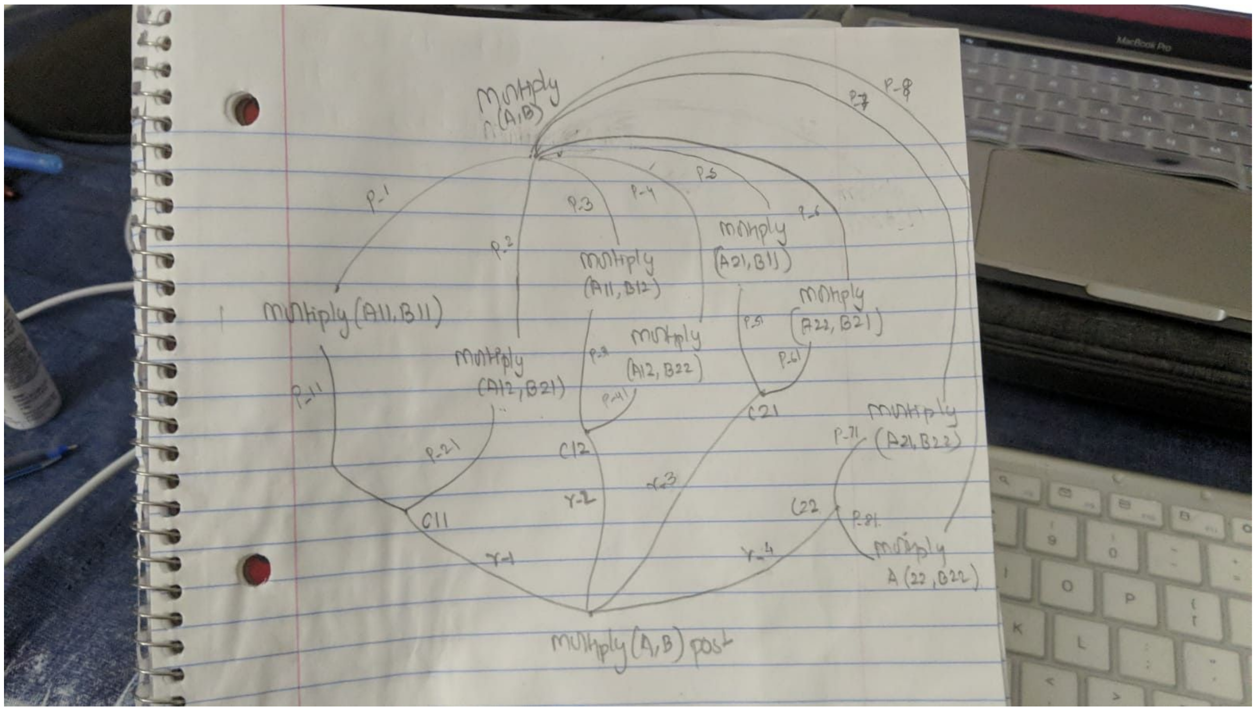
Consider this algorithm to compute C = A ∗ B when A, B, and C are n × n matrices where n is a power of 2.

```
Multiply(A, B):
  A11 = A[1..n/2][1..n/2]
  A12 = A[1..n/2][n/2..n]
  A21 = A[n/2..n][1..n/2]
  A22 = A[n/2..n][n/2..n]

  B11 = B[1..n/2][1..n/2]
  B12 = B[1..n/2][n/2..n]
  B21 = B[n/2..n][1..n/2]
  B22 = B[n/2..n][n/2..n]

  C11 = A11*B11 + A12*B21
  C12 = A11*B12 + A12*B22
  C21 = A21*B11 + A22*B21
  C22 = A21*B12 + A22*B22

  return [[C11, C12],[C21, C22]]
```

Note that the ∗ operation are done by recursively calling the Multiply function. And that the + operation is a matrix operation.

Question: What is the complexity of this function? (Hint: use Master theorem)
Answer: The complexity of this function is O(n^2)

Question: Extract the dependencies.
Answer:-



Question: What is the width?
Answer:- Since there will be 8 independent task , the width will be 8

Question: What is the work?
Answer:- the work will be the time taken by the 8 recursion calls and 4 additional calls required to add those 8 calls.

Question: What is the critical path? What is its length?
Answer:- Critical path will be sum of time taken by 1 recursion call, 1 addition call and 1 return

3 Merge Sort

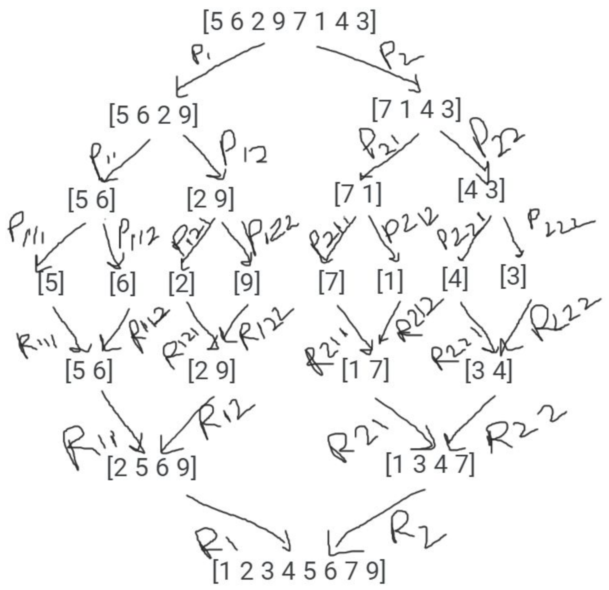
Question: Recall the merge sort algorithm. (Give the algorithm.)
Ans:
It is a classic divide and conquer based algorithm. So it has basically two main components, first it divides arrays into sub arrays and sorts those and second it merges the sub arrays to result in a final sorted array.

```
Function mergeSort(Array, Length)
    If Length == 1 return Array
    leftArray = mergeSort(0, Length/2)
    rightArray = mergeSort(Length/2 + 1, Length)
    Return merge(leftArray, rightArray)

Function merge(leftArray, rightArray)
    B = new Array
    While leftArray =! Empty AND rightArray != Empty
        If leftArray[0] < rightArray[0]
            B.insrtAtLast(leftArray[0])
            Remove leftArray[0]
        Else
            B.insrtAtLast(rightArray[0])
            Remove rightArray[0]
    While leftArray != Empty
        B.insrtAtLast(leftArray[eachElement])
    While rightArray != Empty
        B.insrtAtLast(rightArray[eachElement])
    Return B
```

Question: What is the complexity of this function?

Ans: Merge sort runs in O(n log(n)) as the function mergeSort works in O(log(n)) and function merge works in O(n). SO adding both it becomes O(n log(n)).

Question: Extract the dependencies. (Hint: instead of using loop iterations as a task, you can use function calls and function return as tasks. Think that merge sort is recursive! Remember that when working with functions, a name in two different functions can represent different underlying variable/memory locations.)
Ans:
Here each recursive function does not depend on any shared variable or memory. So this is why we can consider each task as an independent task.



Question: Do all tasks have the same processing time?
Ans: Here in every recursive function the array is divided into half and merged. So considering each recursive call as a task its processing time can not be the same.

Question: What is the width?
Ans: Each function call is independent so total function calls will be the width.

Question: What is the work?
Ans: Theta(mergeSort(log n to base 2))

Question: What is the critical path? What is its length?
Ans:P1 --- > R1

Question: How does the schedule of such an algorithm look like when P = 4? (What I mean is that whatever the values of n, the schedules have “shapes”. What “shape” does any schedule for this problem have? The sketch of what a Gantt chart would look like answers the question.)

Ans: In recursive functions each task is independent which means that does not depend on any shared variable. So, it is possible to apply parallelism on recursive functions.