> Note some syntax used:
>
> - `A[i..j]` means a slice of the array from `A[i]` to `A[j]`, inclusive.
> - `a ← b` means assign value `b` to `a` while `a = b` is a boolean expression that compares the value of `a` and `b`. (This is different from the common programming language)
> - `A[1..n] ← []` means the pseudocode declare a 1d array called `A` that is size `n` big. Same for arbitrary n-d array.
> - `A[a or b]` means the 1d array `A` has size 2, and it only be indexed by `A[a]` or `A[b]`.
> - `foreach value ← [a, b]` means a for loop that will be executed 2 times, first time with the `value ← a`, and second time with `value ← b`.
> - `(a, b) ← c` means that the value `c` is a tuple, we assign the first entry of the tuple to `a` and second entry to `b`.

(a)

we define the a function called `CountRecursive(root, marked)`, which will given the number of all possible matchings of tree which root is `root` under the constraint that `root` can not form a matching to any of its children if the `marked` is `true`.

We use a 2d array `count[1..length, true or false]`. Given that current function's `root` The `count[i, mark]` just store the calculation result of `CountRecursive(children[i], mark)`

Few things to notice:

- `ChildrenCount(node)` returns the count of the children for current node. `Children(node)` returns an 1d array of all the children of current tree. `Length(array)` returns the size of the array.
- The special case is when the current node have no children (`ChildrenCount(root) = 0`). In this case, there is only one matching (the empty set).
- The final function that satisfy the problem is the `Count(root)`

```
Count(root):
    return CountRecursive(root, false)

CountRecursive(root, marked):
    if ChildrenCount(root) = 0:
        return 1

    children ← Children(root)
    length ← Length(children)
    count[1..length, true or false] ← []

    for i ← 1 to length:
        foreach mark ← [true, false]:
            count[i, mark] ← CountRecursive(children[i], mark)

    sum ← 0
    product ← 1
    for i ← 1 to length:
        product ← product * count[i, false]

    sum ← sum + product

    if marked:
        return sum

    for i ← 1 to length:
        product ← 1
        for j ← 1 to length:
            if i = j:
                product ← product * count[j, true]
            else:
                product ← product * count[j, false]
        sum ← sum + product

    return sum
```

We see that:

- The base case when there is only one vertex (i.e. the root has no children), the time complexity is just $O(1)$
- For the first single for loop, there are V iteration, each iteration takes $O(1)$ time, and then the total time complexity for this part is just $V \cdot O(1) = O(V)$
- For the second double for loop, there are V^2 iterations, each iteration takes $O(1)$ time, and then the total time complexity for this part is just $V^2 \cdot O(1) = O(V^2)$
- For the part where we build the array `count`, there are $2V$ iterations, and it's time complexity is

  $$\sum_{i=1}^{\text{size}(V)} 2T(V_i)$$

  where the $\text{size}(V)$ just means the number of vertices in the tree.

We see that the total time complexity is just

$$T(1) = O(1)$$
$$T(V) = \sum_{i=1}^{\text{size}(V)} 2T(V_i) + O(V^2)$$

we try to unfold the recurrence:

$$T(V) = O(V^2) + \sum_{i=1}^{\text{size}(V)} \sum_{j=1}^{\text{size}(V_i)} 2T(V_{i,j}) + O(V_i^2)$$

$$T(V) = O(V^2) + \sum_{i=1}^{\text{size}(V)} \sum_{j=1}^{\text{size}(V_i)} \sum_{k=1}^{\text{size}(V_j)} 2T(V_{i,j,k}) + O(V_{i,j}^2) + O(V_i^2)$$

$$\vdots$$

$$T(V) = O(V^2) + \sum_{i=1}^{\text{size}(V)} \sum_{j=1}^{\text{size}(V_i)} \sum_{k=1}^{\text{size}(V_j)} \cdots + O(V_{i,j,k}^2) + O(V_{i,j}^2) + O(V_i^2)$$

we could actually show that $\sum_{j=1}^{\text{size}(V_i)} O(V_{i,j}^2)$ is smaller than $O(V_i^2)$, to do this, we need to see that

$$1 + \sum_{j=1}^{\text{size}(V_i)} \text{size}(V_{i,j}) = \text{size}(V_i)$$

that is, the size of tree is equal to the 1 (the root) and the sum of size of all subtree. This lead to the inequality:

$$\sum_{j=1}^{\text{size}(V_i)} V_{i,j}^2 < V_i^2$$

and that means the $\sum_{j=1}^{\text{size}(V_i)} O(V_{i,j}^2)$ is smaller than $O(V_i^2)$. Using the similar method, we could show that $\sum_{k=1}^{\text{size}(V_j)} O(V_{i,j,k}^2)$ is smaller than $O(V_{i,j}^2)$ and so on. This means that

$$T(V) = O(V^2) + \sum_{i=1}^{\text{size}(V)} \sum_{j=1}^{\text{size}(V_i)} \sum_{k=1}^{\text{size}(V_j)} \cdots + O(V^2) + O(V^2) + O(V^2)$$

and the depth of the sum is actually the height of the tree, and therefore

$$T(V) = O(V^2 h)$$

where V is the number of vertices in the tree, and h is the height of the tree.

(b)

$$T(1) = 1$$
$$T(2) = 2$$
$$T(n) = T(n-1) + T(n-2)$$

The value of $T(500)$ is `139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125`, We see that $\log_2(T(500)) > 64$, which means it couldn't fit into a 64-bit word.

(c)

From part (b), We see that we may not be able to represent arbitrary number using only 64-bit word. To make sure that this algorithm could run on 64-bit machine (where each bit could have k different values, this is 2 almost in every case, since typical computers are binary), we could represent a number by an array $A[1..n]$, where each element in the array is a 64-bit integer. The number represented by the such array $A[1..n]$ is:

$$\sum_{i=1}^{n} A[i] \cdot (k^{64})^i$$

(without loss of generality, we would assume that the $k = 2$, that is, the machine is binary from now on)

we could then define the `Add` and `Multiply` operator on two big number that represented as $A[1..n]$ and $B[1..n]$ (we could pad one number so that 2 arrays have same length) as:

```
Add(A[1..n], B[1..n]):
    result[1..(n + 1)] ← []

    for i ← 1 to n:
        (carry, sum) ← Add64(A[i], B[i])
        result[i + 1] ← result[i + 1] + carry
        result[i] ← sum

    return result

Negate(A[1..n]):
    for i ← 1 to n:
        A[i] ← -A[i]
    return A

Sub(A[1..n], B[1..n]):
    return Add(A, Negate(B))

Multiply(A[1..n], B[1..n]):
    if n = 1:
        return Multiply64(A[1], B[1])
    else:
        m ← Ceil(n / 2)
        a ← A[(m + 1)..n]; b ← A[1..m]
        c ← B[(m + 1)..n]; d ← A[1..m]

        e ← Multiply(a, c)
        f ← Multiply(b, d)
        g ← Multiply(Sub(a, b), Sub(c, d))

        e_result[1..2n] ← []
        e_result[(2m+1)..2n] ← e

        f_result[1..2n] ← []
        f_result[1..2m] ← f

        middle_result[1..2n] ← []
        middle_result[(m+1)..3m] ← Sub(Add(e, f), g)

        return Add(Add(e_result, f_result), middle_result)
```

Where `Add64` are `Multiply64` are the primitive function on the machine and they should only take $O(1)$. The `Add64` returns a tuple of `(carry, sum)`, where the `carry = (A[i] + B[i]) / 2^64` and `sum = (A[i] + B[i]) % 2^64` (the `carry` and `sum` represented here is what a machine's arithmetic unit will normally provide).

We see that the `Add` operator runs at $O(n)$ times as indicated by the single for loop. The `Multiply` operator is a modified version of **Karatsuba** multiplication algorithm that returns product of the number represented using array, the product is also expressed using array. Using similar analysis of time complexity we seen in the GPS, we could see that the time complexity for the `Multiply` is $O(n^{\log_3 2})$.

To make sure the algorithm in part (a) will work on a 64 bit machine, we could just replace the `*` and `+` in the original algorithm by the `Multiply` and `Add` defined above, and we see the time complexity in this case is:

$$T(1) = O(1)$$

$$T(V) = \sum_{i=1}^{\text{size}(V)} 2T(V_i) + O(V^2 M(n))$$

where $M(n)$ denote the time complexity for multiplication. The n is an appropriate length of the array that used to represent the number defined above. This n makes sure that no overflow will happen. Following the same calculation process in part (a), we see that

$$T(V) = O(V^2 M(n) h)$$

We could use maybe more advanced multiplication algorithm for multiplication, but for the case when we use **Karatsuba** algorithm, the time complexity is

$$T(V) = O(V^2 h n^{\log_3 2})$$

(d)

We define the a function called `MaximumWeightRecursive(root, k, marked)`, which will give the maximum weight matching of the tree which root is `root` under the constraint that

1. `root` can not form any matching to its children is the `marked` is `true`
2. the number of matching could not exceed `k`

We create a 3d array `MaximumWeight_array[1..Length(V), 1..k, true or false]`, the `MaximumWeight_array[id, k, marked]`, where `id = Vid(node)` is the id of certain node in the tree, will store the result of `MaximumWeightRecursive(node, k, marked)`

We define a function `Store(value)` that write the value into `MaximumWeight_array` and then return it input `value`. Just to clarify, the `Store(value)` function is defined inside the function `MaximumWeightRecursive`, so the `root, k, marked` used in the `Store` is just the `root, k, marked` in the outer function `MaximumWeightRecursive` (That is basically what a closure is). If you don't understand, just think `Store` as a function that will (1) store the value it got to the corresponding place in `MaximumWeightRecursive` (2) then just return the what it received.

Notice that in the function `MaximumWeightRecursive`, we will use `Store` function to store the result of `MaximumWeightRecursive` to `MaximumWeight_array` so that when we later enter the function `MaximumWeightRecursive` with exact same parameter, the function will just return the value from `MaximumWeight_array` in $O(1)$.

Few things to notice:

- `ChildrenCount(node)` returns the count of the children for current node. `Children(node)` returns an 1d array of all the children of current tree. `Length(array)` returns the size of the array. `Left(node)` and `Right(node)` gives the left child of right child of the current node (the tree in this a binary tree). `Weight(from, to)` gives you the weight of the edge where its end point is `from` and `to`. `Vid` (vertex id) is a function that maps every vertex of the tree to a number between `1` and `Length(V)`, `V` is the array of all vertices. (The function `Vid` could be implemented in $O(1)$ time)
- Notice, in the pseudocode, when the parent have only one children, we use the `children[0]` to select it instead of checking for both `Left(parent)` and `Right(parent)`. This results in cleaner code.
- `Split(n, k)` is a function that gives all possible splits that split the value `n` into `k` values where each value is non-negative, it puts all possible splits into a array. For example, `Split(4, 2)` will give you `[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]`
- There are few special case:
    - `k = 0`, in this case, we couldn't select any edge, so the maximum weight matching is just 0 in this case.
    - `ChildrenCount(root) = 0`. In this case, the current root have no children, and therefore there is no edge to choose from. So, the maximum weight matching is just 0 in this case.
- The final function that satisfy the problem is the `MaximumWeight(root, k)`

```
MaximumWeight_array[1..Length(V), 1..k, true or false] ← []

MaximumWeight(root, k):
    return MaximumWeightRecursive(root, k, false)

MaximumWeightRecursive(root, k, marked):
    if MaximumWeight_array[Vid(root), k, marked] is defined:
        return MaximumWeight_array[Vid(root), k, marked]

    Store(value):
        MaximumWeight_array[Vid(root), k, marked] = value
        return value

    children ← Children(root)

    if k = 0:
        return Store(0)
    if ChildrenCount(root) = 0:
        return Store(0)
    if ChildrenCount(root) = 1:
        if marked:
            return Store(MaximumWeightRecursive(children[1], k, false))
        else:
            return Store(max(
                MaximumWeightRecursive(children[1], k - 1, true)
                    + Weight(root, children[1]),
                MaximumWeightRecursive(children[1], k, false)
            ))

    maxWeight ← 0

    foreach (a, b) ← Split(k, 2):
        maxWeight ← Max(
            maxWeight,
            MaximumWeightRecursive(Left(root), a, false) +
                MaximumWeightRecursive(Right(root), b, false)
        )

    if marked:
        return Store(maxWeight)

    foreach (a, b) ← Split(k - 1, 2):
        maxWeight ← Max(
            maxWeight,
            MaximumWeightRecursive(Left(root), a, true) +
                MaximumWeightRecursive(Right(root), b, false) +
                Weight(root, Left(root))
            MaximumWeightRecursive(left(root), a, false) +
                MaximumWeightRecursive(Right(root), b, true) +
                Weight(root, Right(root))
        )

    return Store(maxWeight)
```

To analyze time complexity, we see that the base case when there is only one vertex (i.e. the root without children), the time complexity is just $O(1)$. Also, when the $k = 0$, the time complexity is also just $O(1)$.

We see that in the worst case, we have to calculate the `MaximumWeightRecursive(root, k, marked)` **once** for every possible combination of `root`, `k` and `marked`. The total number of possible combination is $V \cdot k \cdot 2$, and we see that in the foreach loop, the `Split(k, 2)` will create $O(k)$ number of tuples. (These are `(0, k)`, `(1, k - 1), (2, k - 2) … (k, 0)`), and in the loop, if we assume the time complexity of the recursive call is just $O(1)$ (this assumption is valid, since the time the recursive call takes is already counted when we considered all the $V \cdot k \cdot 2$ combinations). Then the time complexity in each case, which is dominated by the for each call, will just take $O(k)$ times. Therefore, since we have $O(Vk)$ combinations, and for each case we need $O(k)$ time, we need $O(Vk^2)$ time in total.