Recursion

1.1 Reductions

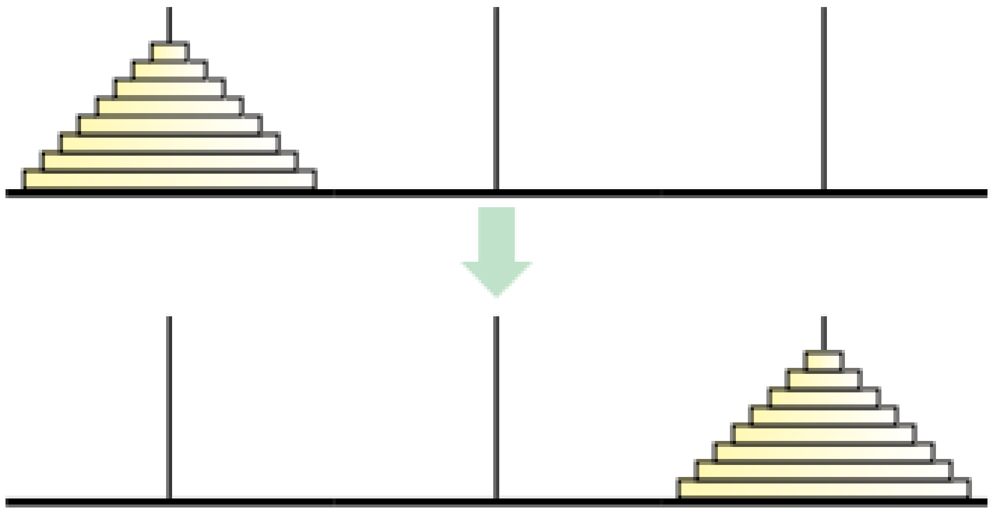
- Reduce one problem X to another problem Y
- Solving for Y is a black box for X, meaning X is independent of Y.
- We can assume that the black box solves Y correctly.

1.2 Simplify and Delegate

- Recursion is a type of reduction.
- If the problem can be solved directly, then solve it directly.
- If not, then reduce the problem to a simpler version of the same problem.
- In other words, it is either simple enough to be solved in one step, or it is complex and you need to simplify the problem and "give it to someone else".
    - The book calls "someone else" the *Recursion Fairy*.
    - AKA the *Induction Hypothesis*.
- Because we are simplifying/reducing the problem with recursion, we need a **base case** that can be solved without any further reduction.
    - We cannot have an infinite sequence.

1.3 Tower of Hanoi

- Tower of Hanoi Problem: How can we move a tower of n disks from one peg to another, using a third spare peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?



- Strategy:
    - We want to move the entire tower from one peg to another.

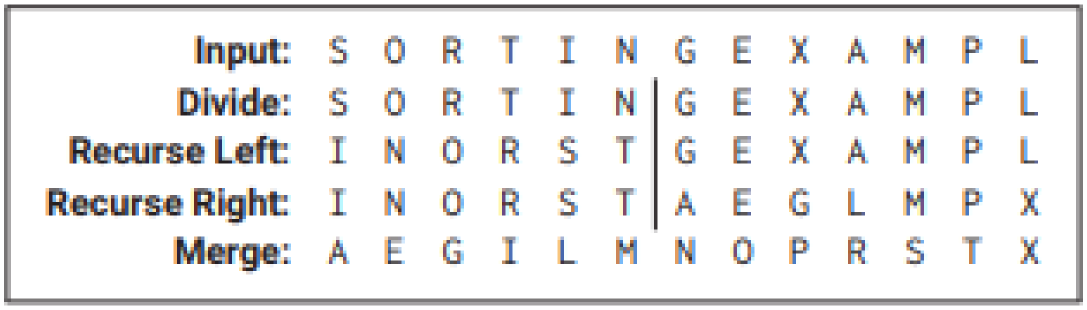
        - We can't move the bottom disk because the smaller disks are on top.
    - We need to move $n-1$ disks off from the n-th disk to a placeholder disk.
    - Then, we move the n-th disk to its destination.
    - Finally, we move the $n-1$ disks from the placeholder to the destination.
- Our strategy allowed us to reduce the problem from n disks to $n-1$ disks.
    - We solved it for n, but we don't know how to solve it for $n-1$. This is okay because how $n-1$ gets solved is a black box for us.
    - We can hand it off the problem to the *Recursion Fairy* and we don't have to worry about it anymore.
- When $n=0$, the problem is *trivial* (omg leiss reference) and we don't have to reduce the problem anymore.
- We **shouldn't** unearth the recursive sequence and try to see how it all works out. Our **only** task is to reduce the problem to a simpler version of it, or to solve it directly if it is possible.
- In terms of Induction, our base case is $n=0$, and for any $n \geq 1$, the Inductive Hypothesis implies that our algorithm correctly moves the top $n-1$ disks.
- Psuedocode for the Recursive Hanoi Algorithm:

```
Hanoi(n, src, dst, tmp):
if n>0
        Hanoi(n-1, src, tmp, dst)
        move disk n from src to dst
        Hanoi(n-1, tmp, dst, src)
```

- If $n>0$, then first we move $n-1$ disks from the starting(src) peg to the placeholder(tmp) peg, using the destination(dst) peg as a temporary peg. Then, we move the n-th disk from our starting peg to our destination peg with a single, simple move. Finally, we move back all the $n-1$ disks from the temporary peg to the destination peg by calling `Hanoi()` recursively.
- Let $T(n)$ denote the amount of moves it takes to transfer n disks. This is equivalent to the running time of our algorithm.
    - Our vacuous base case implies that $T(0)=0$, and the more general recursive algorithm implies that $T(n)=2T(n-1)+1$ for any $n \geq 1$.
    - We can easily guess that the general equation for T is $T(n)=2^n-1$ by writing out the first few values of n.
        - Prove this is true with a simple induction proof.
    - If we had $n=64$ disk, moving a tower of 64 disks would take $2^{64}-1=18,446,744,073,709,551,615$ moves. At a single move per second, it would take around 585 billion years to complete the algorithm.

1.4 Mergesort

1. Divide the input array into two parts that are roughly equal in size.
2. Recursively mergesort each part.
3. merge each sorted part into a single array.



Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.5. A mergesort example.

- We delegate the second step to the Recursion Fairy.
- The merge step is a different independent subroutine.
- The mergesort algorithm is recursive.
- Pseudocode:

```
MergeSort(A[1..n]):
if n>1
        m <- floor(n/2)
        MergeSort(A[1..m])
        MergeSort(A[m+1..n])
        Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
        i <- 1; j <- m+1
        for k <- 1 to n
                if j>n
                        B[k] <- A[i]; i <- i+1
                else if i>m
                        B[k] <- A[j]; j <- j+1
                else if A[i] < A[j]
                        B[k] <- A[i]; i <- i+1
                else
                        B[k] <- A[j]; j <- j+1
        for k <- 1 to n
                A[k] <- B[k]
```

Proof of Mergesort:

- **Lemma 1.1.** *Merge* correctly merges the subarrays $A[1..m]$ and $A[m+1..n]$, assuming those subarrays are sorted in the input.
- **Proof:** Let $A[1..n]$ be any array and m any integer such that the subarrays $A[1..m]$ and $A[m+1..n]$ are sorted. We prove that for all k from 0 to n, the last $n-k-1$ iterations of the main loop correctly merge $A[i..m]$ and $A[j..n]$ into $B[k..n]$. The proof procceds by induction on $n-k+1$, the number of elements remaining to be merged.

  If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. This is the base case. Otherwise, there are four cases to consider for the k-th iteration of the main loop.
  - If $j > n$, then subarray $A[j..n]$ is empty, so $\min(A[i..m] \cup A[j..n]) = A[i]$.
  - If $i > m$, then subarray $A[i..m]$ is empty, so $\min(A[i..m] \cup A[j..n]) = A[j]$.
  - Otherwise, if $A[i] < A[j]$, then $\min(A[i..m] \cup A[j..n]) = A[i]$.
  - Otherwise, we **must** have $A[i] \geq A[j]$, and $\min(A[i..m] \cup A[j..n]) = A[j]$.
    In all four cases, $B[k]$ is correctly assigned the smallest element of $A[i..m] \cup A[j..n]$. In two cases with the assignment $B[k] \leftarrow A[i]$, the induction hypothesis implies that the last $n-k$ iterations of the main loop correctly merge $A[i+1..m]$ and $A[j..n]$ into $B[k+1..n]$. Similarly, in the other two cases the rest of the subarrays are correctly merged. □
- **Theorem 1.2.** *MergeSort* correctly sorts any input array $A[i..n]$.
- **Proof:** We prove the theorem by induction on n. If $n \leq 1$, the algorithm correctly does nothing. Otherwise, the induction hypothesis implies that our algorithm correctly sorts the two smaller subarrays $A[1..m]$ and $A[m+1..n]$, after which they are corrected merged into a single sorted array, by Lemma 1.1. □
- We obtain the following recurrence relation for `MergeSort()`:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n).$$

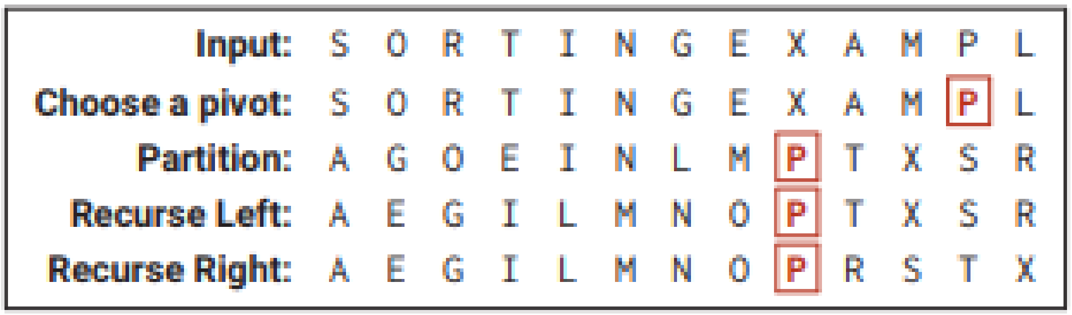
  Using a technique called *domain transformations*, we can simplify the recurrence to $T(n) = 2T(n/2) + O(n)$. The "all levels equals" case of the recursion tree method immediately implies the solution $T(n) = O(n \log n)$.
  - We can verify the solution $T(n) = O(n \log n)$ by induction.
  - We are going to go over *domain transformations* and recursion trees later in this chapter.

1.5 Quicksort

- We split the array **before** recursion, which is more difficult. However, merging the sorted arrays is trivial.

1. Choose a pivot element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.

3. Recursively quicksort the first and last subarrays (The array with smaller elements and the array with bigger elements).



Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	$\boxed{\text{P}}$	L
Partition:	A	G	O	E	I	N	L	M	$\boxed{\text{P}}$	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	$\boxed{\text{P}}$	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	$\boxed{\text{P}}$	R	S	T	X

Figure 1.7. A quicksort example.

- Pseudocode:

```
QuickSort(A[1..n]):
        if(n>1)
                Choose a pivot element A[p]
                r <- Partition(A,p)
                QuickSort(A[1..r-1])
                QuickSort(A[r+1..n])
```

```
Partition(A[1..n], p):
        swap A[p] <-> A[n]
        l <- 0
        for i <- 1 to n-1
                if A[i]<A[n]
                        l <- l+1
                        swap A[l] <-> A[i]
        swap A[n] <-> A[l+1]
        return l+1
```

- In `Partition()`, the input p is the index of the pivot in the unsorted array. The array is partitioned and returns the new index of the pivot element.
    - The variable l counts the number in the array that are less than the pivot element.
- Proving Quicksort: First prove Partition is correct, then prove that QuickSort is correct assuming Partition is correct. The proof is not given in the textbook and is left as an exercise.
    - To prove Partition, we must prove the following loop invariant: At the end of each iteration of the main loop, everything in the subarray $A[1..l]$ is less than $A[n]$, and nothing in the subarray $A[l+1..i]$ is less than $A[n]$.
- `Partition()` runs in $O(n)$ time.

- `QuickSort()` runs on $T(n) = T(r-1) + T(n-r) + O(n)$.
    - It depends on r, the *rank* of the chosen pivot.
- If we can always choose the median element in the array, then $r = \lceil n/2 \rceil$, and the two subproblems would be as close to the same size as possible. The recurrence relation would become

$$T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$$

  and we'll have $T(n) = O(n \log n)$ using the recursion tree method.
- We can locate the median element in linear time, but the algorithm is complex and sometimes impractical.
- Most of the time we'll just choose the first or last element of the array as the pivot.
    - Then, r can take any value $1 \leq r \leq n$ so we will have

$$T(n) = \max_{1 \leq r \leq n} \big(T(r-1) + T(n-r) + O(n)\big).$$

- The worst case is when the two subproblems are completely unbalanced and either $r = 1$ or $r = n$, and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is thus $T(n) = O(n^2)$.
- We can take the median of three elements, usually first, middle, and last as the pivot. In practice this is more efficient, we will still have $r = 2$ or $r = n - 1$ in the worst case. The recurrence relation is then $T(n) \leq T(1) + T(n-2) + O(n)$ which still has the solution $T(n) = O(n^2)$.
- The pivot element should usually fall somewhere in the middle of the array with rank between $n/10$ and $9n/10$. Thus, the average-case running time should be $O(n \log n)$.
    - Considered as best case running time, but it is not because it makes assumptions about the data being sorted.

1.6 The Pattern

- Mergesort and Quicksort follows the **divide and conquer** pattern:

1. **Divide** the problem into several independent smaller instances of the same problem.
2. **Delegate** each smaller instance to the "Recursion Fairy" (Don't worry about the smaller instances and just trust that they are solved correctly).
3. **Combine** the solutions for the smaller instances into the final solution from the initial instance.

- If the size of the instance falls below some constant threshold, then we can just solve the problem directly in constant time with brute force.
- Proof of a Divide and Conquer solution will almost always use induction.
- Analyzing the running time will require setting and solving a recurrence relation, which can be solved using a recursion tree most of the time.

1.7 Recursion Trees

- Used to visualize and solve divide-and-conquer recurrences.

- A rooted tree with a node for every recursive subproblem.
  - The value of each node is the running time of that subproblem, excluding recursive calls
- Overall running time of the algorithm is the sum of all the values in the tree.
- Let's say we have a divide-and-conquer algorithm with the following assumptions:
  - Time spent on non-recursive work is $O(f(n))$ time.
  - It makes r recursive calls, each on a problem of size n/c.
- The running time of the algorithm would then have the recurrence relation $$
  T(n)=r\ T(n/c) + f(n)

- The recursion tree for $T(n)$ then has a root with value $f(n)$ and r children.
  - These children have their own recursion tree for $T(n/c)$, and each of them has r children with recursion trees for $T(n/c^2)$.
- We can generalize the running time of each recursive call of the recursion tree to $T(n/c^d)$, where d is the depth (think of it as how far we are from the initial recursive call).
- We can draw this out like so:

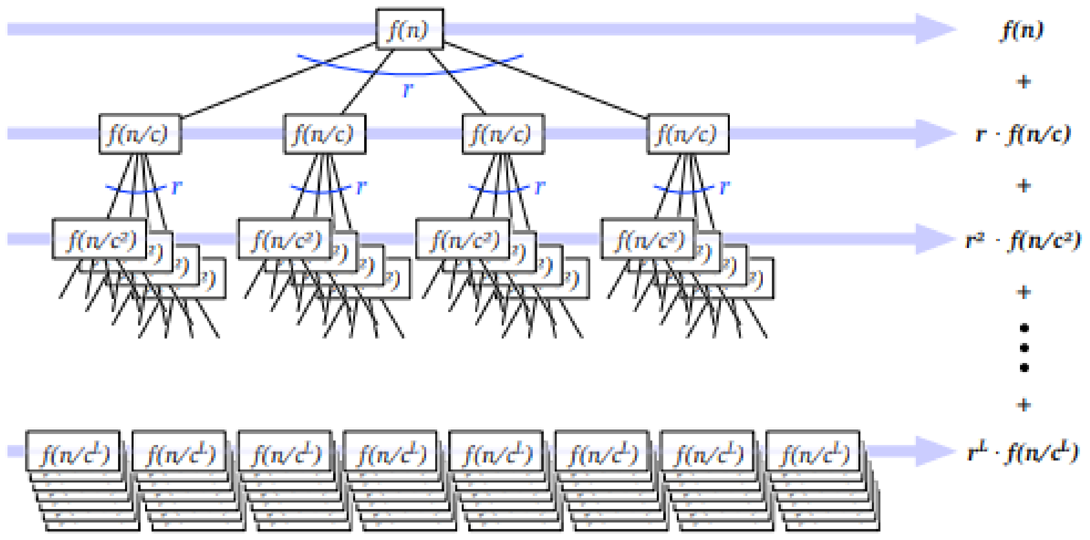


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

- The leaves are the base cases of the recurrence.
- We can assume $T(n) = 1$ for all $n \leq n_0$, where n_0 is an arbitrary positive constant (our base case).
- $T(n)$ is the sum of all the values in the recursion tree, so we can find it by getting the sum of the entire tree level by level. Lets say i is the ith level of the tree, where $0 \leq i \leq L$. Each level of the tree has r^i nodes and they each have the value $f(n/c^i)$. So we can calculate the running time to be:

$$T(n) = \sum_{i=0}^{L} (r^i) \cdot f(n/c^i)$$

- Our base case $n_0 = 1$ implies $L = \log_c n$ because $n/c^L = n_0 = 1$. Then, the number of leaves in the tree is exactly $r^L = r^{\log_c n} = n^{\log_c r}$. Thus, the last term in the summation is $n^{\log_c r} \cdot f(1) = O(n^{\log_c r})$, because $f(1) = O(1)$.
- There are three common cases where the summation is easy to evaluate:
  - If the series *decays exponentially* (**Decreasing**), every term is a constant factor smaller than the previous term. Then, $T(n) = O(f(n))$. The sum is dominated by the value at the root.
  - If all the terms in the series are **equal**, we immediately have $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$.
  - If the series *grows exponentially* (**Increasing**), every term is a constant factor that is larger than the previous one. So, $T(n) = O(n^{\log_c r})$. The sum is dominated by the number of leaves at the bottom of the tree.
- The levels of the recursion tree for the mergesort recurrence $T(n) = 2T(n/2) + O(n)$ are equal. This immediately implies $T(n) = O(n \log n)$

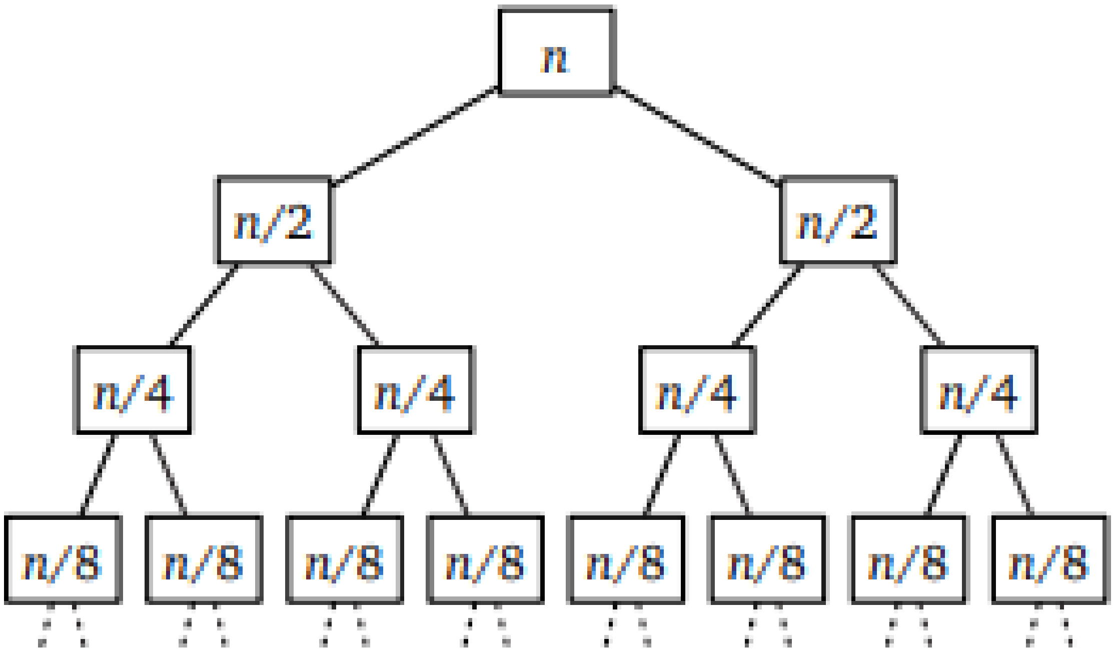


Figure 1.10. The recursion tree for mergesort

- The Recursion tree can also be used for algorithms where the recursive subproblems have different sizes.
- We can ignore floors and ceilings in running time recurrences by using domain transformation.

1.8 Linear-Time Selection

1.9 Fast Multiplication

- Multiplying two numbers x and y:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

where $x = 10^m a + b$ and $y = 10^m c + d$.

- We reduce our main problem to four sub-problems, where we find the sub-products $ac,\ bc,\ ad,\ \text{and}\ bd$.
  - Solving these four subproducts would give us a running time of $O(n^{\log_2 4}) = O(n^2)$
  - According to Karatsuba, we can find the middle coefficient $(bc + ad)$ using only one more recursive multiplication rather than two using the algebraic identity

$$ac + bd - (a-b)(c-d) = bc + ad$$

  which allows us to reduce the running time to $O(n^{\log_2 3}) \approx O(n^{1.58496})$.
- Pseudocode:

```
FastMultiply(x,y,n):
        if n=1
                return x*y
        else
                m = ceil(n/2)
                a = floor(x/10^m); b = x mod 10^m
                c = floor(y/10^m); d = y mod 10^m
                e = FastMultiply(a,c,m)
                f = FastMultiply(b,d,m)
                g = FastMultiply(a-b, c-d, m)
                return 10^(2m)e+10^m(e+f-g)+f
```

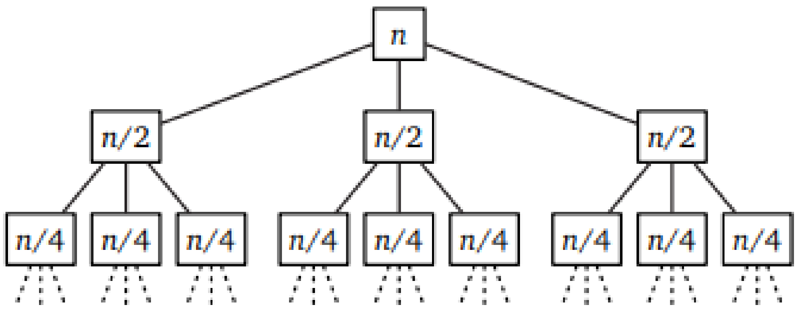


Figure 1.15. The recursion tree for Karatsuba's divide-and-conquer multiplication algorithm

1.10 Exponentiation

- Naively, we can calculate a^n by using a for-loop that multiplies a by itself n times.
- We can use a divide-and-conquer method instead, using a simple recursive formula

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \ \cdot \ a & \text{otherwise} \end{cases}$$

- Pseudocode:

```
PingalaPower(a, n):
        if n = 1
                return a
        else
                x = PingalaPower(a, floor(n/2))
                if n is even
                        return x*x
                else
                        return x*x*a
```