fit@hcmus

PROGRAMMING TECHNIQUES

Week 7: Recursion (2)

Nguyễn Hải Minh - 03/2024

Content

- Types of Recursion
- Work through examples to get used to the recursive process
- Removing Recursion
- Applications of Recursion

TYPE OF RECURSION

- Direct Recursion & Indirect Recursion
- Linear Recursion, Binary Recursion & Multiple Recursion
- Tail Recursion vs Non-tail Recursion
- Nested Recursion

Types of Recursion

- Direct

```
int Fact(int x){
  if(x == 0)
    return 1;
  else
    return x * Fact(x-1);
}
```

- Indirect (Mutual)

```
bool isEven(int x){
   if(x == 0)
      return true;
   else
      return isOdd(x-1);
}

bool isOdd(int x){
   return !isEven(x);
}
```

Types of Recursion

- Linear Recursion

```
int Fact(int x){
  if(x == 0)
    return 1;
  else
    return x * Fact(x-1);
}
```

- Binary Recursion

```
int Fibo(int n){
  if(n < 2)
    return n;
  else
    return Fibo(n-1) +
                    Fibo(n-2);
}
```

Types of Recursion

- Tail Recursion

```
void PrintNum(int n)
{
  cout << n << endl;
  if(n > 0)
    PrintNum(n-1);
}
```

- Non-tail Recursion

```
void PrintNum(int n)
{
  if(n > 0)
  {
    PrintNum(n-1);
    cout << n << endl;
  }
}
```

Content

- Nested Recursion

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(2n)) & \text{if } n \leq 4 \end{cases}$$

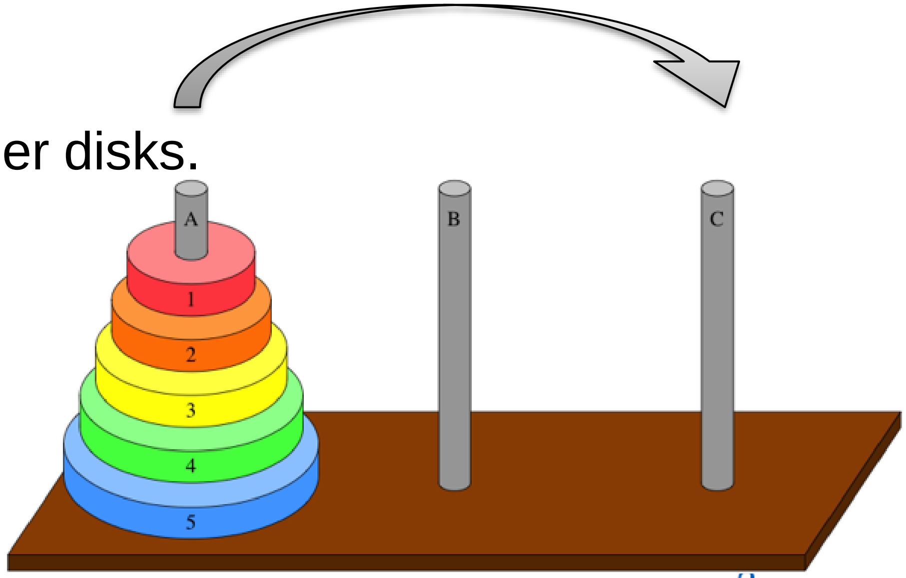
```
int Func_h(int n)
{
  if(n ==0) return 0;
  if(n > 4) return n;
  return Func_h(2+ Func_h(2*n));
}
```

WORK THROUGH EXAMPLES

- Tower of Hanoi
- Operations on Linked List
- Mystery Recursive Call

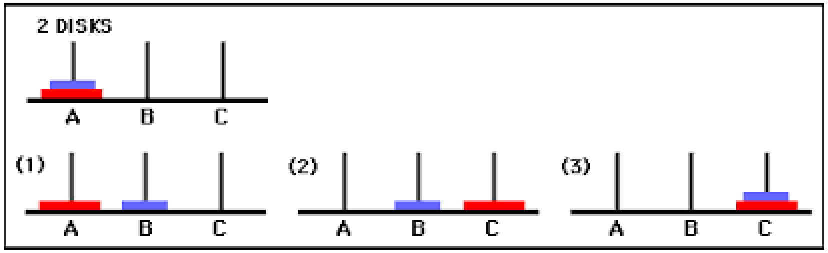
Tower of Hanoi

- Given a set of three pegs A, B, C, and n disks, with each disk a different size (disk 1 is the smallest, disk n is the largest)
- Initially, n disks are on peg A, in order of decreasing size from bottom to top.
- The goal is to move all n disks from peg A to peg C
- 2 rules:
  1. You can move 1 disk at a time.
  2. Smaller disk must be above larger disks.



Tower of Hanoi

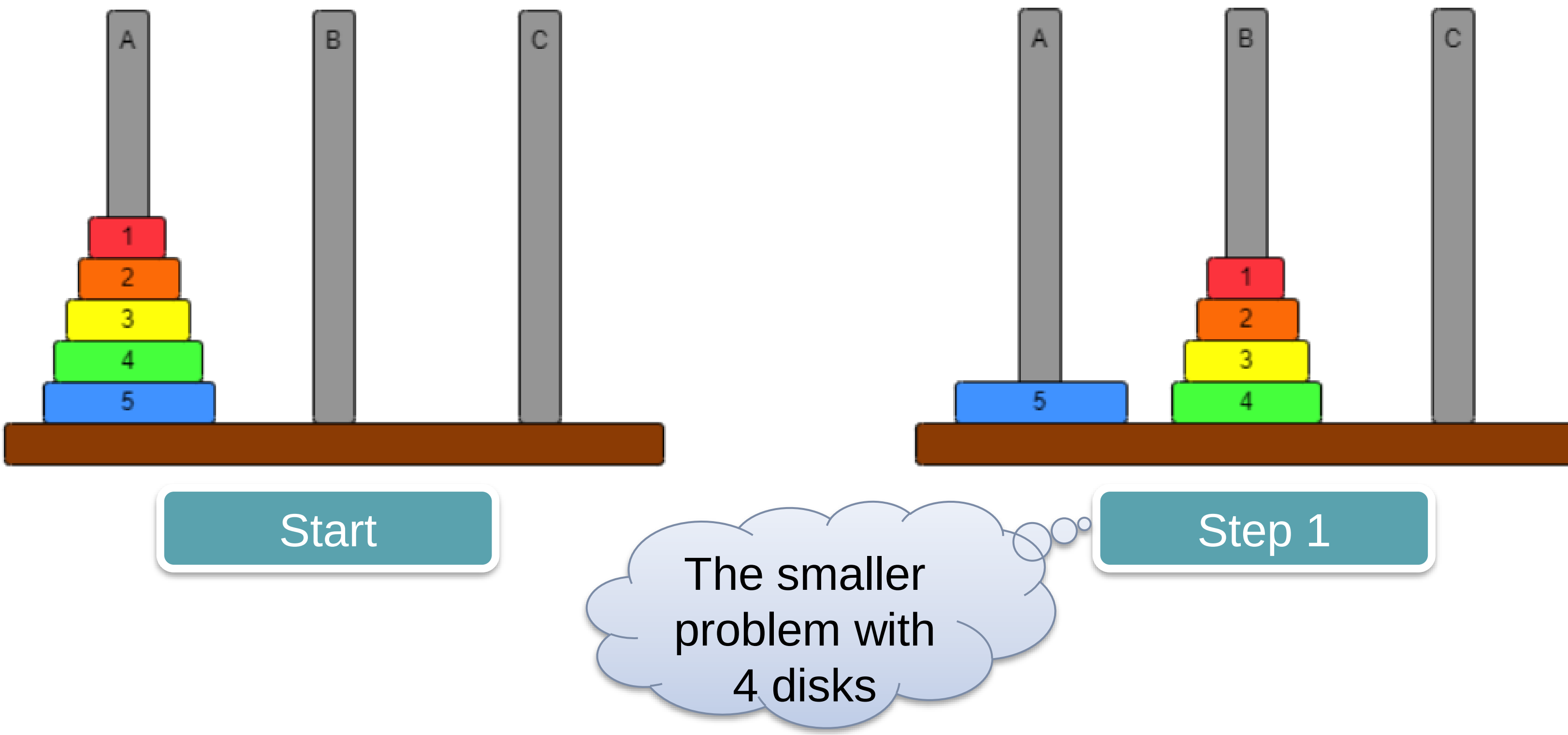
- How to solve this problem recursively?
  - The easiest case: (base case)
    - $n = 1$: just move disk 1 from A to C
  - When $n = 2$: 3 steps (using B as the spare peg)
    - Move disk 1 from A to B
    - Move disk 2 from A to C
    - Move disk 1 from B to C



  - When $n = k$:
    - Move disk $1, 2, \ldots, k - 1$ from A to B
    - Move disk k from A to C
    - Move disk $1, 2, \ldots, k - 1$ from B to C

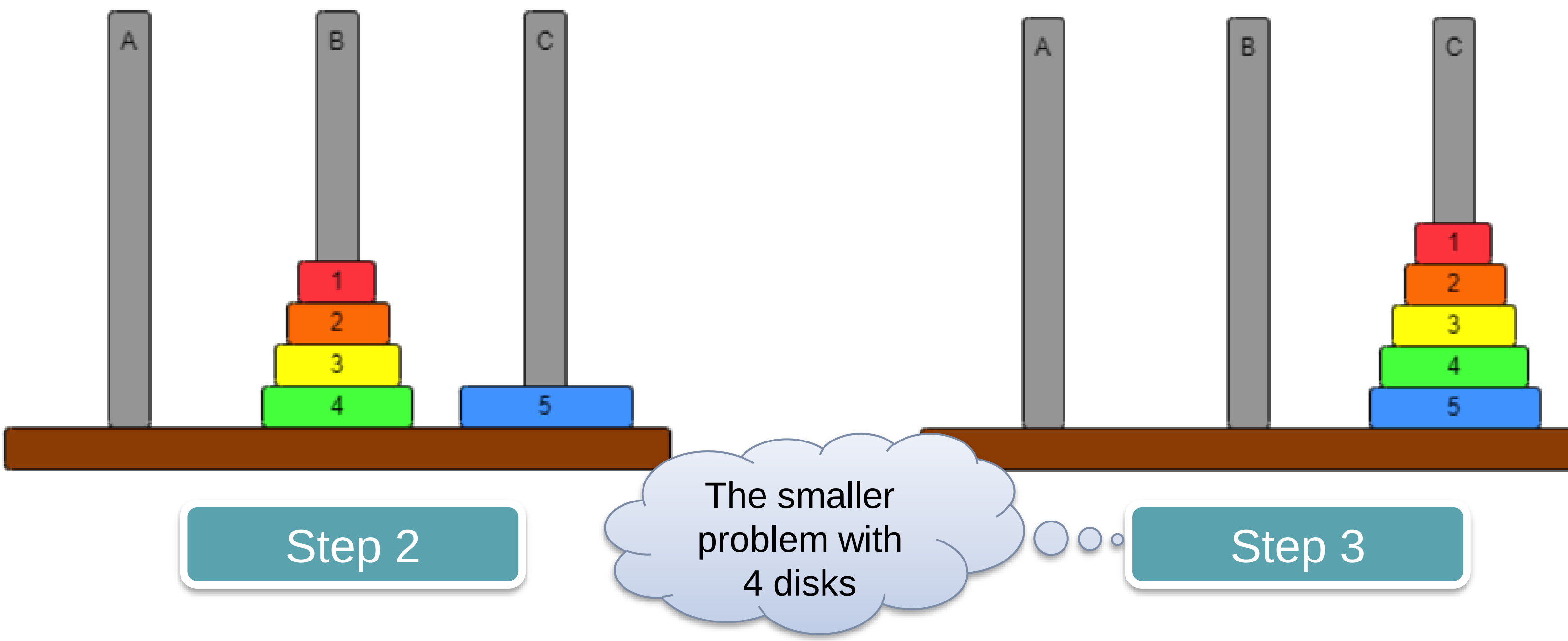
Tower of Hanoi

- How to solve this problem recursively?
  - n = 5:
    - Step 1: Move disk $1, 2, \ldots, 4$ from A to B



Tower of Hanoi

- How to solve this problem recursively?
  - n = 5:
    - Step 1: Move disk $1, 2, \ldots, 4$ from A to B
    - Step 2: Move disk 5 from A to C
    - Step 3: Move disk $1, 2, \ldots, 4$ from B to C



Tower of Hanoi – Recursive Solution

- Determine what the **stopping condition** should be first:
  - *when n is 1*
- What should be done when **this condition is reached**?
  - *Move disk 1 to goal peg*
- What should be done **otherwise**?
  - *3 steps process*:
    - Move $n - 1$ disks to spare peg
    - Move disk n to goal peg
    - Move $n - 1$ disks from spare peg to goal peg

Tower of Hanoi – Recursive Solution

```
void TowerOfHanoi(int n, char from, char to, char tmp)
{
  if (n == 1){
     cout << "Move disk 1 from Peg " << from
          << " to Peg " << to << endl;
     return;
  }
  TowerOfHanoi(n - 1, from, tmp, to);
  cout << "Move disk " << n << " from Peg " << from
       << " to Peg " << to << endl;
  TowerOfHanoi(n - 1, tmp, to, from);
}
```

Base case

Step 1

Step 2

Step 3

→ Function call:

```
TowerOfHanoi(4,'A','C','B');
```

Operations on Linked List

- Now, we will select simple problems that in *reality* should be solved *using iteration* and not recursion.
- But it should give you an understanding of how to design using recursion *(which we will need to understand for next course)*
- **Linked List Recursive Examples:**
  1. **Display** the content of a linear linked list
  2. **Insert** at the end of a linear linked list
  3. **Remove** a specific item from a linear linked list

Display the content of a LLL

- If we were to do this **iteratively**:

```
void Display(node* pHead) {
   while (pHead) {
      cout << pHead->data->title << endl;
      pHead = pHead->pNext;
   }
}
```

Why is it ok in this case to change pHead?

- Look at the stopping condition
  - with recursion we will replace the `while` with an `if....`and replace the traversal with a function call

Display the content of a LLL

- If we were to do this **recursively**:

```
void Display(node* pHead) {
   if (pHead) {
      cout << pHead->data->title << endl;
      Display(pHead->pNext);
   }
}
```

- Now, change this to display the list backwards
  - Discuss the code you'd need to do THAT **recursively**
  - How about display the list backwards using iteration

Insert at the end of a LLL

- Again, this should be done iteratively!
- But, as an exercise determine what the **stopping condition** should be first:
  - *when the pHead pointer is* ***`NULL`***
- what should be done when this condition is reached?
  - *allocate memory and save the data*
- what should be done **otherwise**?
  - *call the function recursively with the* ***`next pointer`***

Insert at the end of a LLL

Iterative

```
void Append(node* & pHead, const video & d) {
    if (!pHead){
        pHead = new node;
        pHead->data = … //save the data
        pHead->pNext = NULL;
    }
    else {
        node* pCurr = pHead;
        while (pCurr->pNext) {
            current = pCurr->pNext;
        }
        pCurr->pNext = new node;
        current = pCurr->pNext;
        pCurr->data = … //save the data
        pCurr->pNext = NULL;
    }
}
```

Insert at the end of a LLL


```
void Append(node* & pHead, const video & d) {
   if (!pHead){
      pHead = new node;
      pHead->data = … //save the data
      pHead->pNext = NULL;
   }
   else {
      Append(pHead->pNext, d);
   }
}
```

Recursive A

pass by reference is used to implicitly connect up the nodes

→ This is much shorter (but less efficient)
→ Notice the stopping condition (!pHead)

Insert at the end of a LLL

```
node* Append(node* pHead, const video & d) {
   if (!pHead){
      pHead = new node;
      pHead->data = … //save the data
      pHead->pNext = NULL;
   }
   else {
      pHead->pNext = Append(pHead->pNext, d);
   }
   return pHead;
}
```

Must use the returned value

Recursive B

pass by value is used, we need to explicitly connect up the nodes

Remove an item from a LLL

- Again, this should be done iteratively!
- But, as an exercise determine what the **stopping condition** should be first:
  - *when the pHead pointer is `NULL`*
  - *when a match (the item to be removed) is found*
- what should be done when this condition is reached?
  - *deallocate memory*
- what should be done **otherwise**?
  - *call the function recursively with the `next pointer`*

Remove an item from a LLL

```
int Remove(node* & pHead, const video & d)
{
   if (!pHead) return 0;  //match not found!
   if (strcmp(pHead->data->title, d->title)==0) {
      node* temp = pHead->pNext;
      delete [] pHead->data->title;
      delete pHead->data;
      delete pHead;
      pHead = temp;
      return 1;
   }
   return Remove(pHead->pNext,d);
}
```

Remove an item from a LLL

- Does this reconnect the nodes?
- How does it handle the special cases of
  a) Empty list
  b) Deleting the first item
  c) Deleting elsewhere

More Examples with Linked List

- Now in class, let's design and implement the following **recursively**
  4) Count the number of items in a linear linked list
  5) Delete all nodes in a linear linked list
- Why would recursion not be the proper solution for push, pop, enqueue, dequeue?

Mystery Recursive Call

- What is the output for the following program fragment?
- Called: `f(5)`

```
int f(int n)
{
   cout << n << endl;
   if (n == 0) return 4;
   else if (n == 1) return 2;
   else if (n == 2) return 3;
   n = f(n-2) * f(n-4);
   cout << n <<endl;
   return n;
}
```

Mystery Recursive Call

- What is the output of the following program or write INFINITE if there are indefinite recursive calls?
- Called: `cout << watch(-7)`

```
int watch(int n)  {
   if (n > 0)
      return n;
   cout << n <<endl;
   return watch(n+2)*2;
}
```

REMOVING RECURSION

Removing Recursion

- Sometimes, we need to convert a recursive algorithm into an iterative one if:
  - The language does not support recursion
  - The recursive algorithm is expensive
- There are 2 techniques to remove recursion:
  - Iteration
  - Stacking

Replacing Recursion with Iteration

- The Simple Method:
  1. Convert all recursive calls into tail calls (stop it you can't)
  2. Introduce a loop around the function body
  3. Convert tail calls into continue statements
  4. Tidy up

Replacing Recursion with Iteration

- Example: Calculate Factorial of a number

```
int Fact(int n){
  if(n < 2)
    return 1;
  return n * Fact(n-1);
}
```

Non-tail recursion

```
int Fact(int n, int acc=1){
  if(n < 2)
    return 1 * acc;
  return Fact(n-1, acc * n);
}
```

Convert to Tail recursion

Replacing Recursion with Iteration

- Example: Calculate Factorial of a number

```
int Fact(int n, int acc=1){
   while(true)
   {
      if(n < 2)
         return 1 * acc;
      (n, acc) = (n-1, acc * n);
      continue;
   }
}
```

Introduce a loop

Replace recursive call by the original function’s agurment list

Replacing Recursion with Iteration

- Example: Calculate Factorial of a number

```
int Fact(int n, int acc=1){
  while(n > 1)
  {
    n = n - 1;
    acc = acc * n;
  }
  return acc;
}
```

Tidy up!

Replacing Recursion with Stack

- Push the parameters that are passed to the recursive function onto a **stack**.
- In other words, we are replacing the program stack by our own stack.
- Example:
  - The `strange` function in last week's topic.

Replacing Recursion with Stack

- The `strange` function

```
void strange() {
    int t;
    cin >> t;

   if (t != 0){
       strange();
       cout << t << " ";
    }
}
```

```
void strange() {
    int t;
    cin >> t;
    stack s;
    while (t != 0) {
        Push(s,t);
        cin >> t;
    }
    while (!IsEmpty(s))
    {
        t = Pop(s);
        cout << t << " ";
    }
}
```

Practice Exercises

1. Make a copy of a linear linked list, recursively
2. Merge two sorted linear linked lists, keeping the result sorted, recursively

APPLICATIONS OF RECURSION

Recursion Applications

1. Mathematical Calculations: Fibonacci sequence, factorials, exponentials, etc.
2. List Traversal: linked list, tree, graph, etc.
3. Backtracking Algorithms: in searching (maze, 8-puzzle, N-queens, etc.)
4. Divide and Conquer
5. Dynamic Programming

Divide and Conquer

- An algorithm design paradigm based on multi-branched recursion.
  - Call themselves recursively one or more times to deal with closely related subproblems.
- Many useful algorithms:
  - Merge sort
  - Binary search
  - Powering a number
  - Fibonacci numbers
  - Matrix multiplication
  - …

Divide and Conquer

1. ***Divide*** the problem (instance) into subproblems.
   - Smaller instances of the same problem.
   - If the problem is small enough: base case.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

Power a number

- Calculate a^n
  - Recursion or iteration:

$$a^0 = 1; a^n = a \cdot a^{n-1}$$

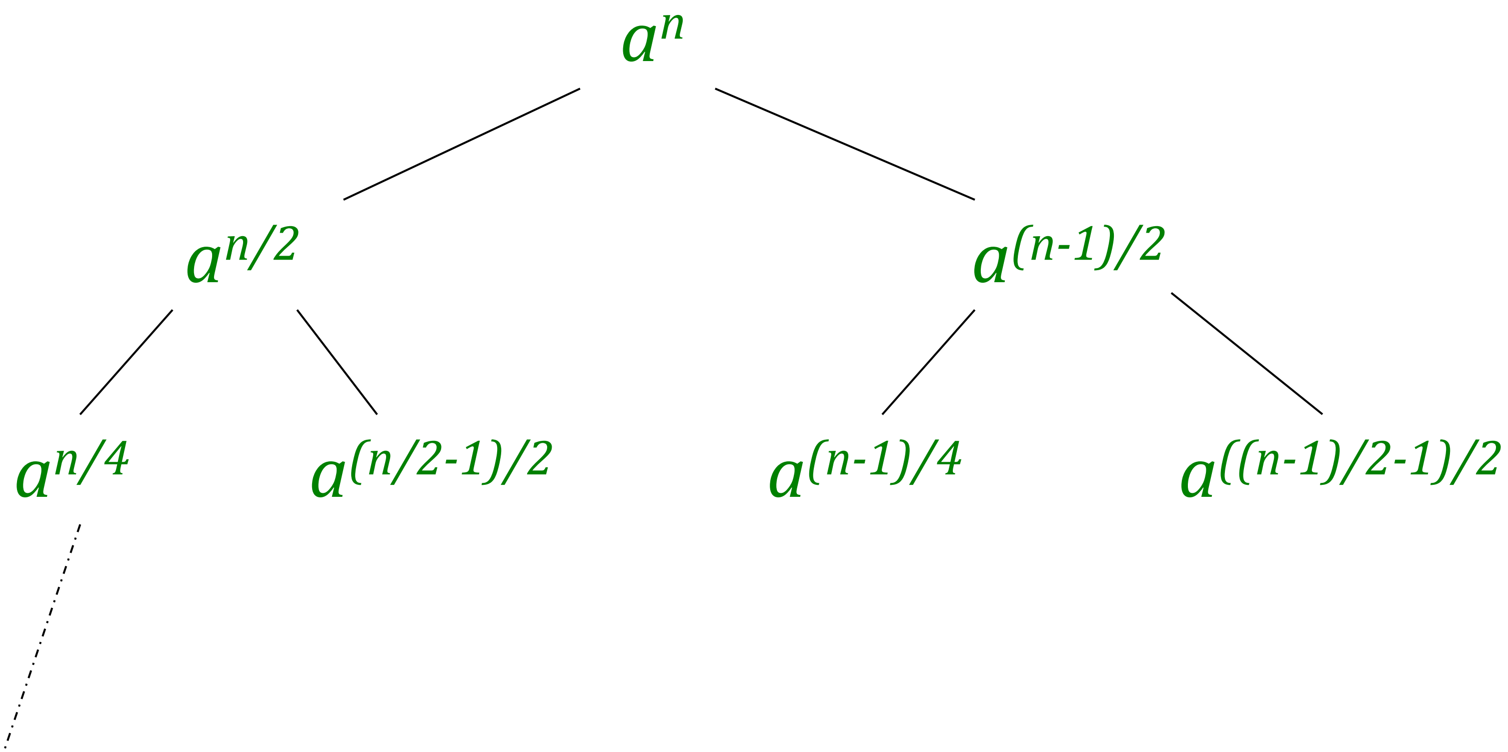
  - Divide and conquer:

$$a^0 = 1$$

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a \cdot a^{(n-1)/2} \cdot a^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

Power a number

- Divide and conquer:



Power a number

- Recursion

```
int Power(int a, unsigned int n) {
  if (n == 0)
    return 1;
   else
     return a * Power(a, n - 1);
}
```

Power a number

- Divide and conquer:

```
int Power(int a, unsigned int n) {
    int p;
     if (n == 0)
        return 1;
     else if (n % 2 == 0){
        p = Power(a, n / 2);
        return p * p;
    }
     else{
        p = Power(a, (n - 1) / 2);
        return a * p * p;
    }
}
```

Power a number recursively

- Calculate 3^{11} using recursion

$3 * 3^{10}$

Power a number recursively

- Calculate 3^{11} using recursion

$3 * 3^{10}$

$3 * 3^{9}$

Power a number recursively

- Calculate 3^{11} using recursion

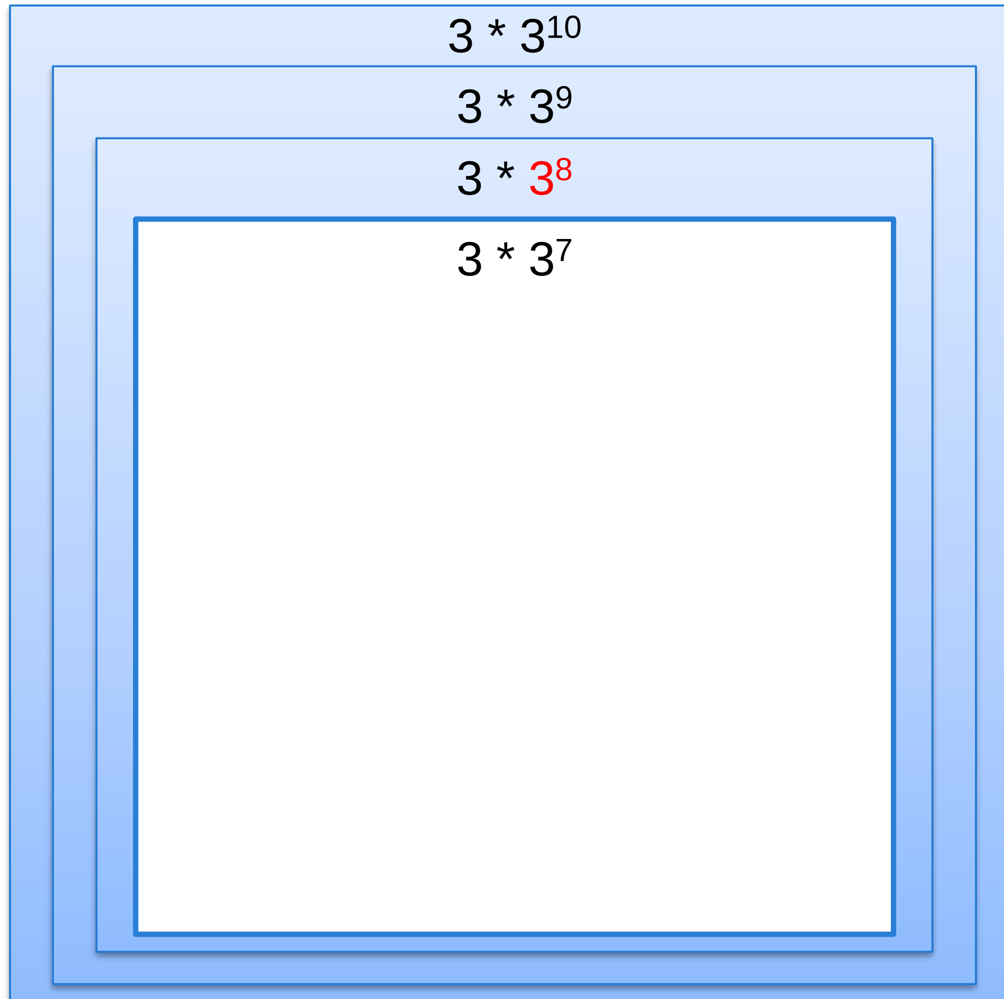
$3 * 3^{10}$

$3 * 3^{9}$

$3 * 3^{8}$

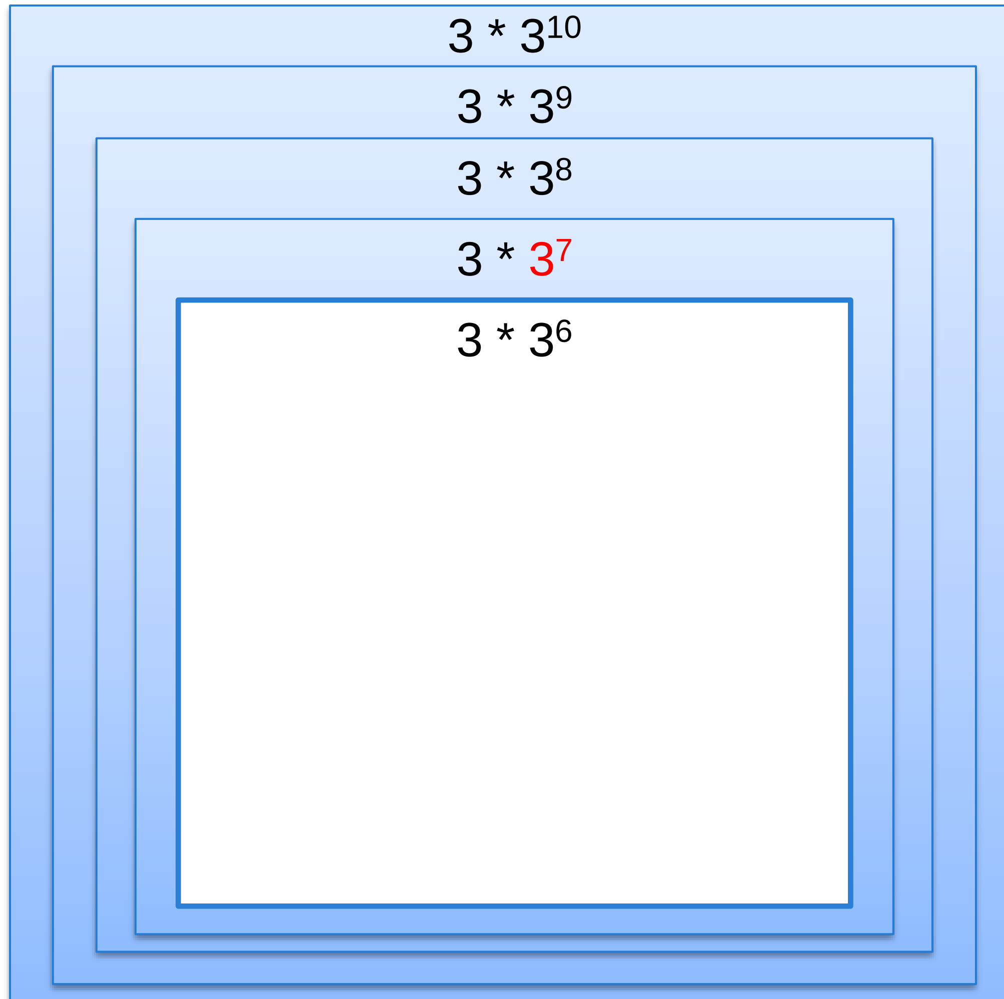
Power a number recursively

- Calculate 3^{11} using recursion



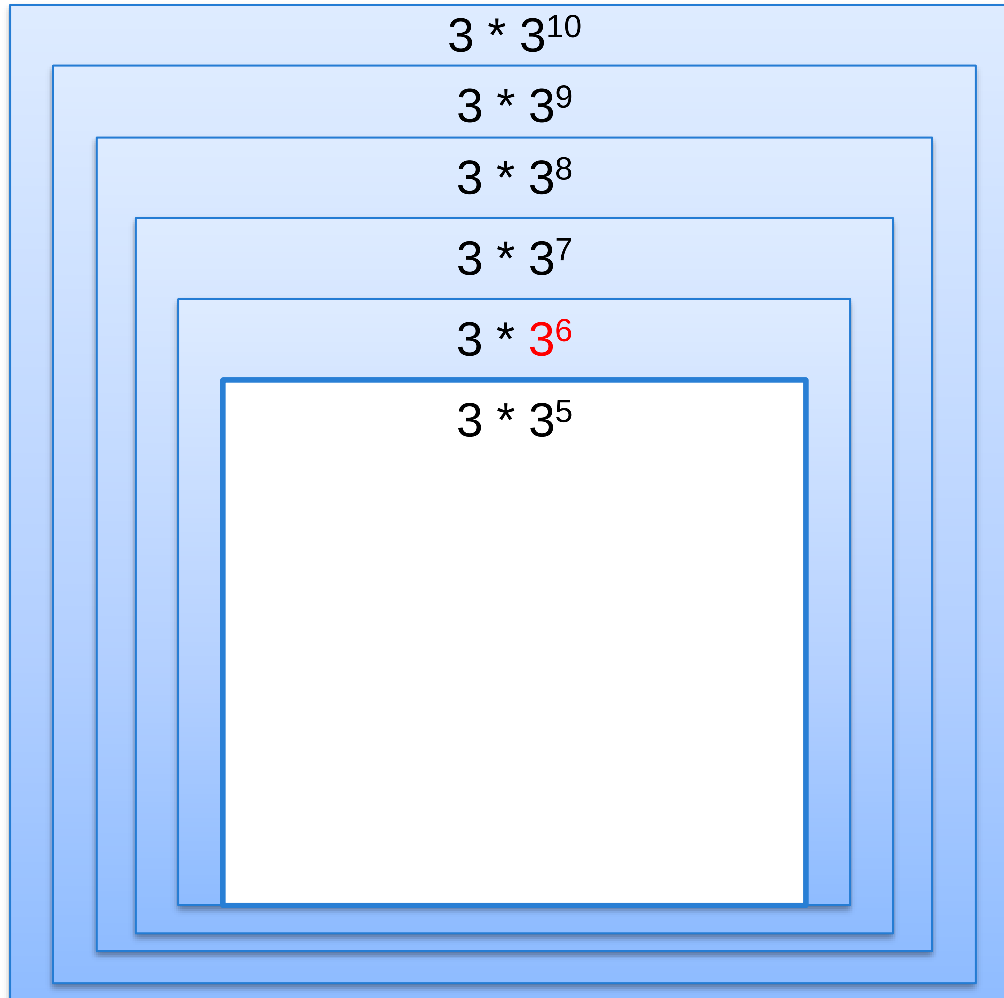
Power a number recursively

- Calculate 3^{11} using recursion



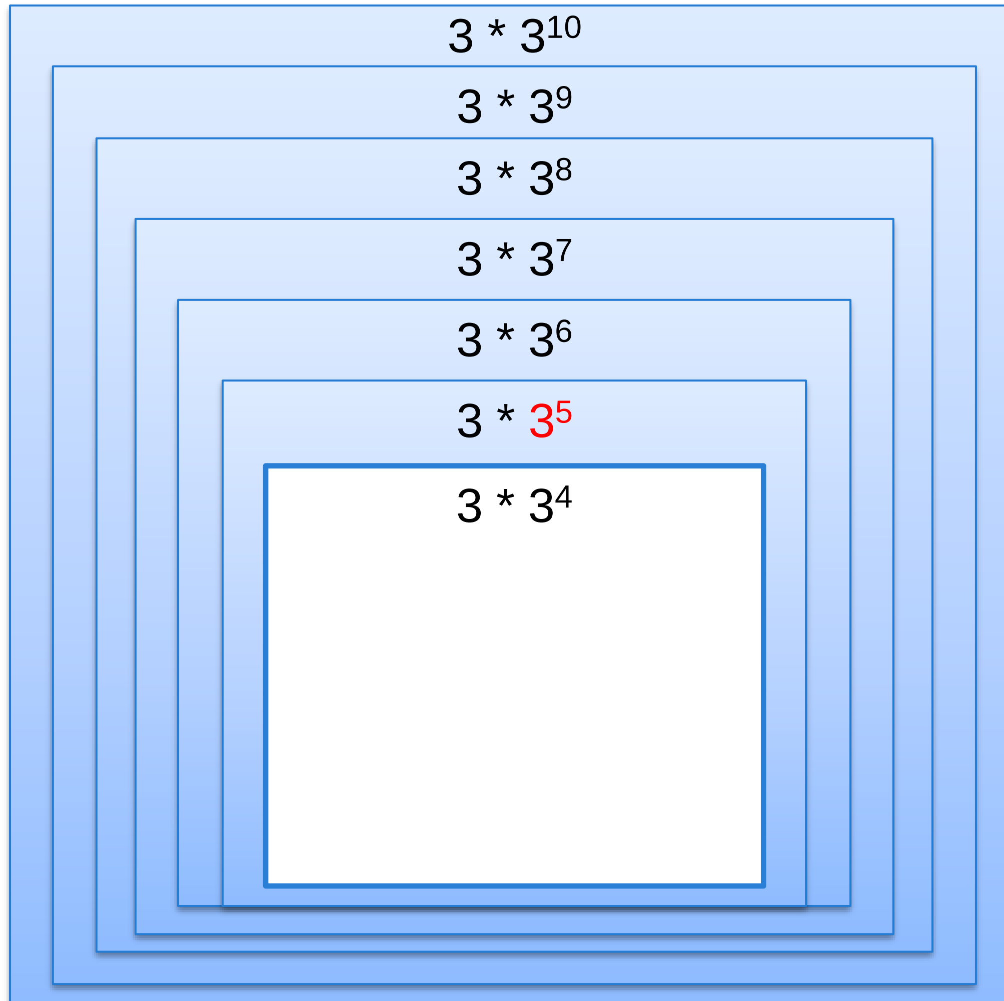
Power a number recursively

- Calculate 3^{11} using recursion



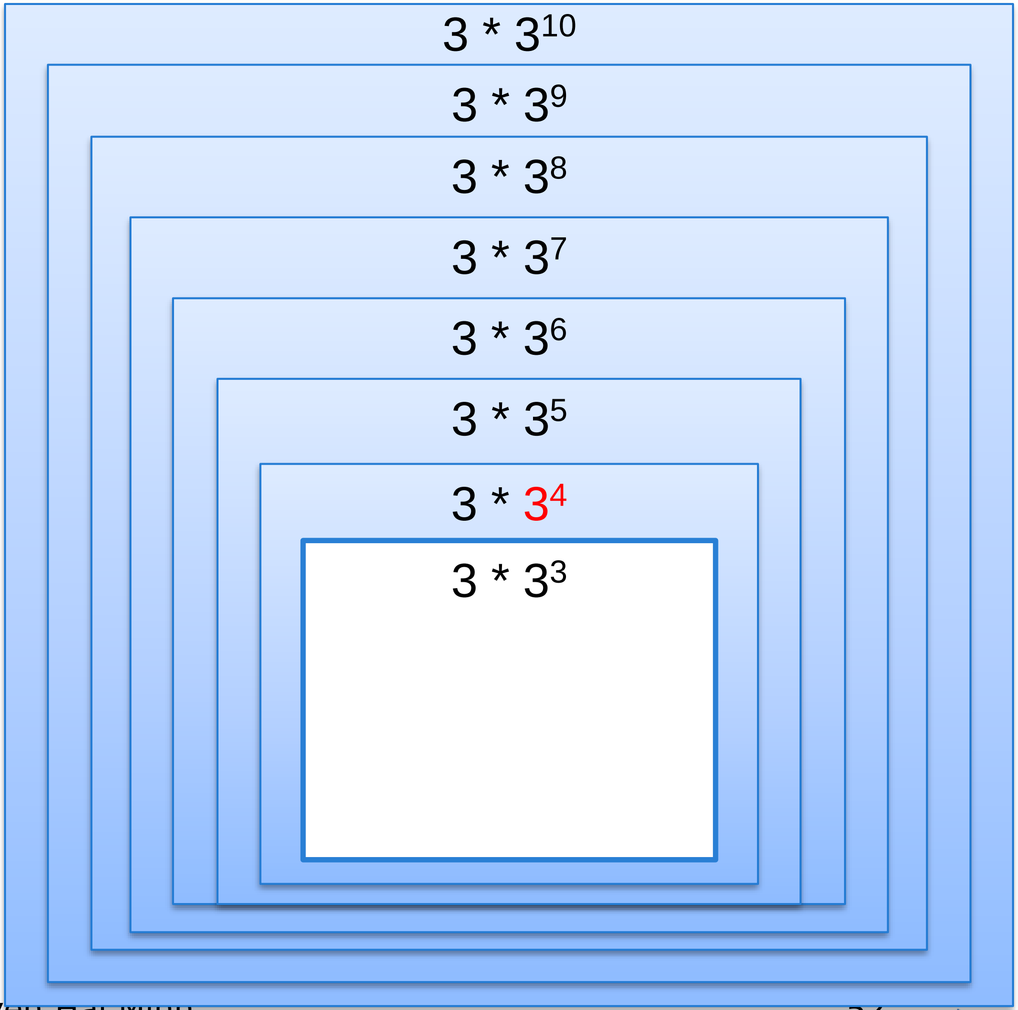
Power a number recursively

- Calculate 3^{11} using recursion



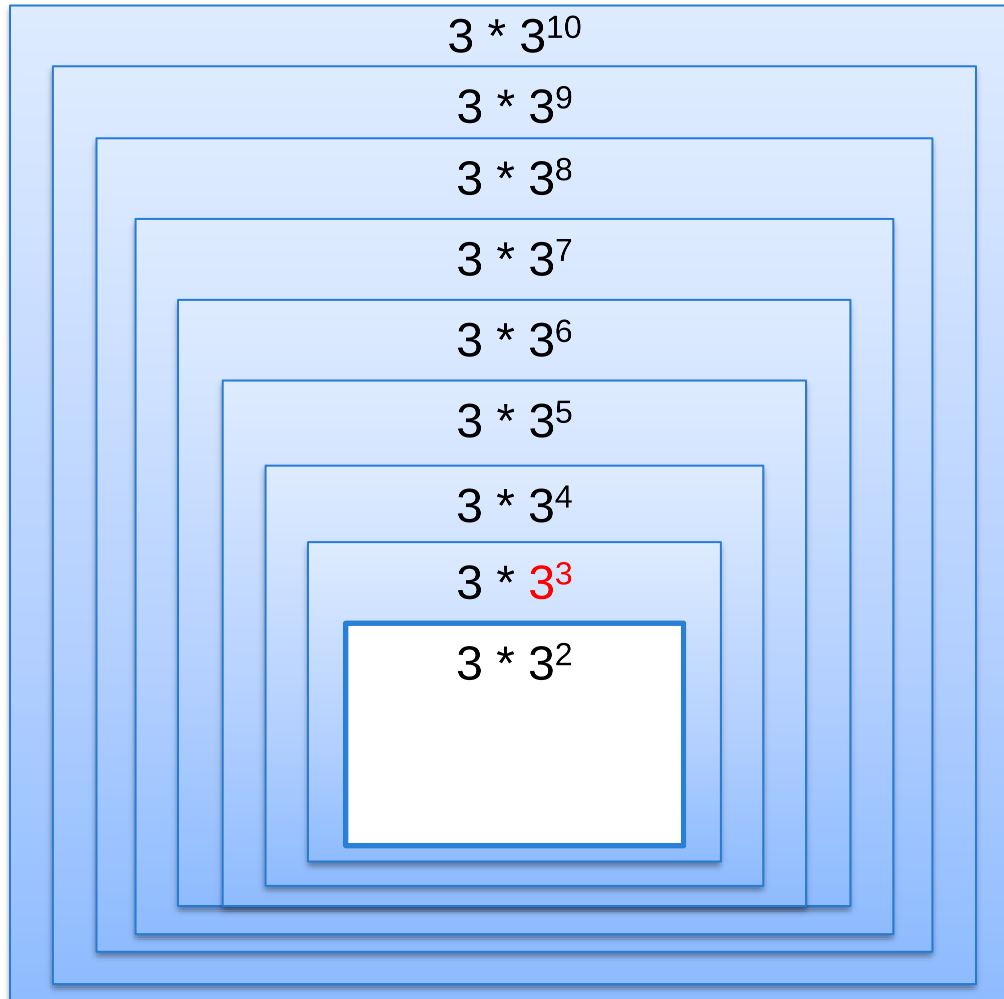
Power a number recursively

- Calculate 3^{11} using recursion



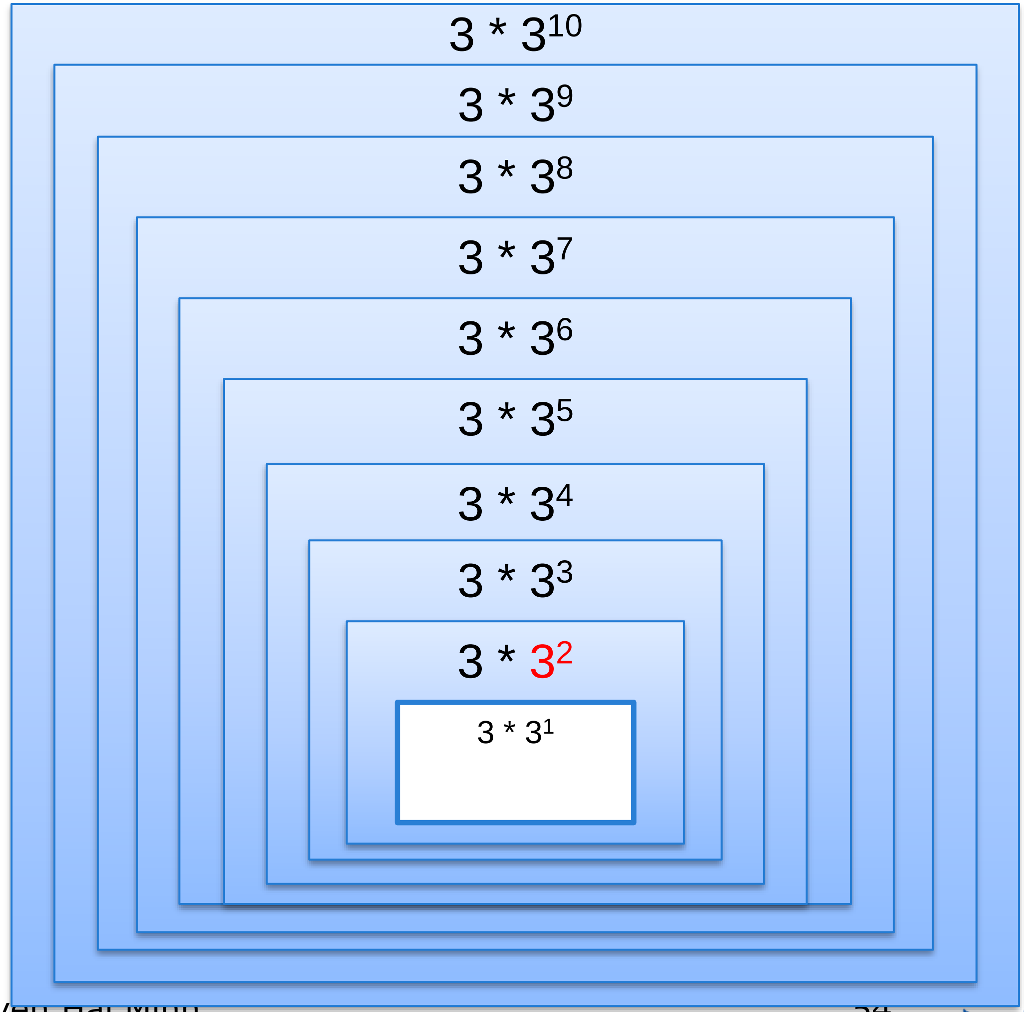
Power a number recursively

- Calculate 3^{11} using recursion



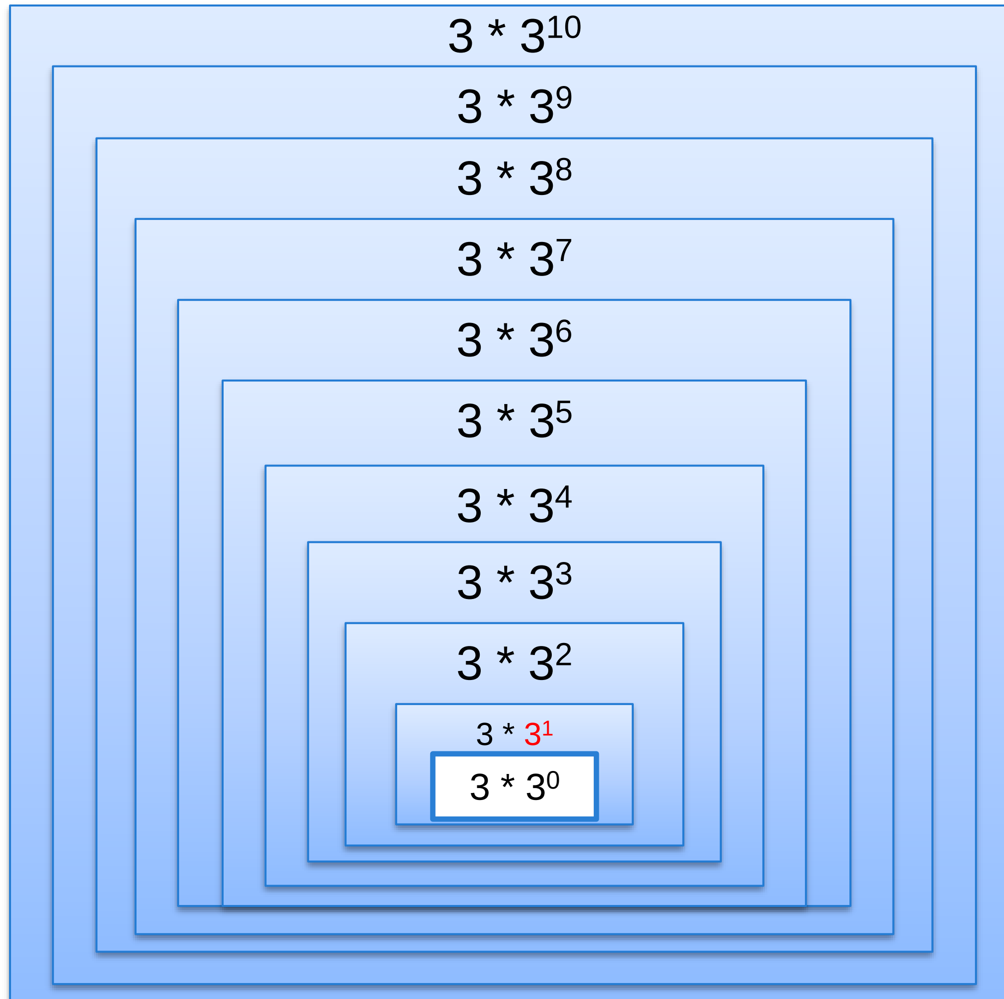
Power a number recursively

- Calculate 3^{11} using recursion



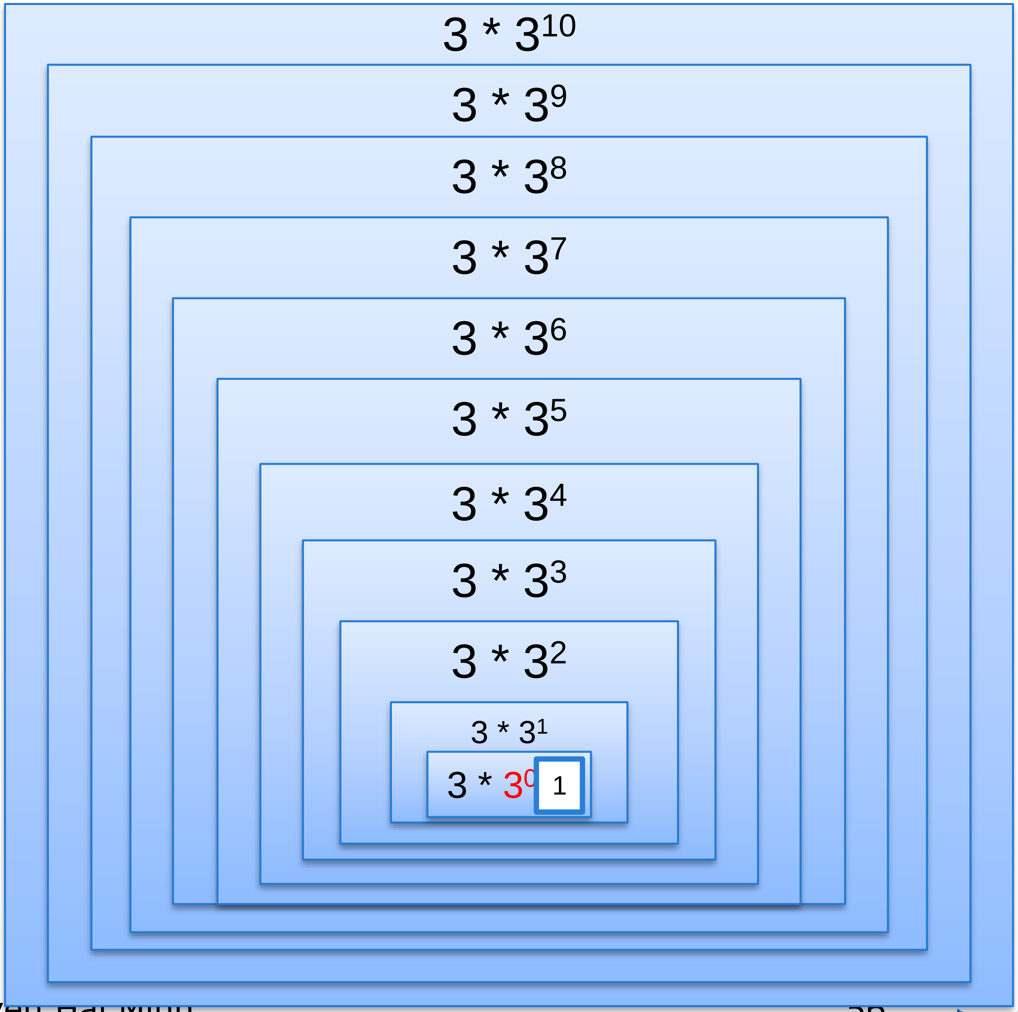
Power a number recursively

- Calculate 3^{11} using recursion



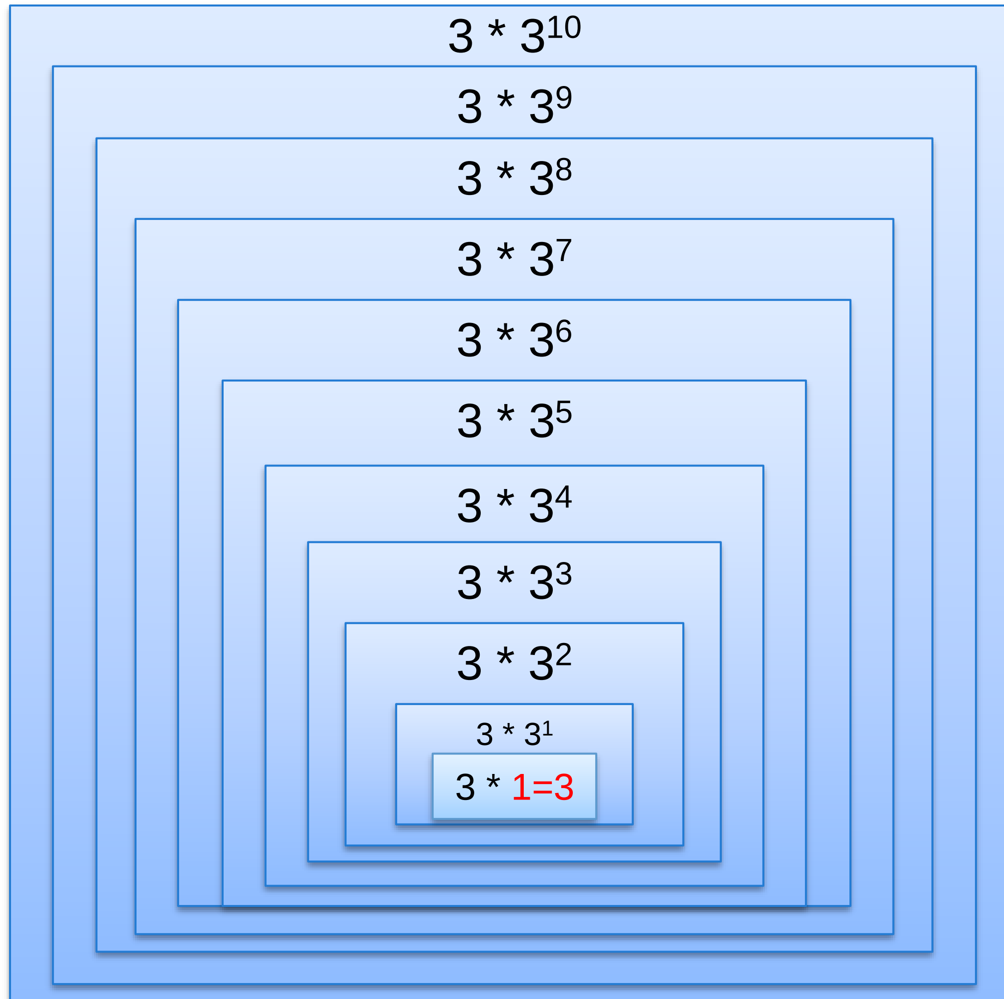
Power a number recursively

- Calculate 3^{11} using recursion



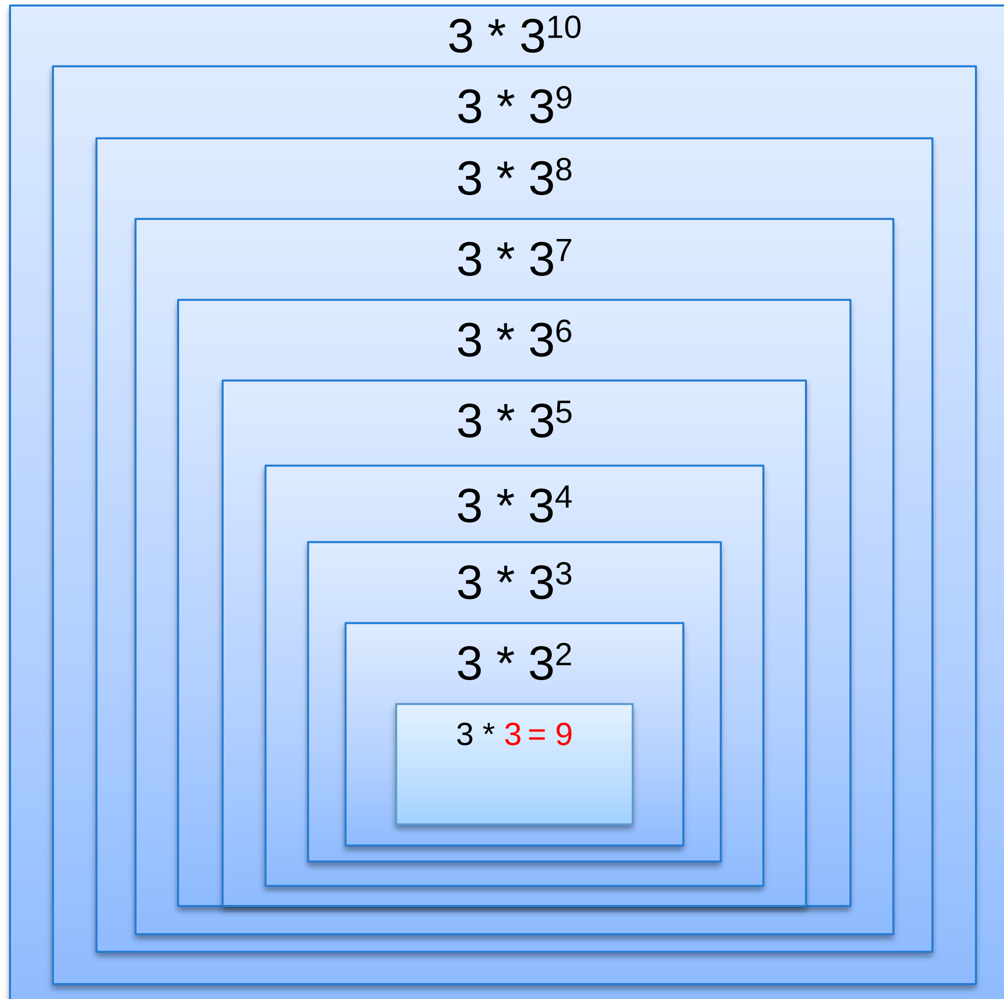
Power a number recursively

- Calculate 3^{11} using recursion



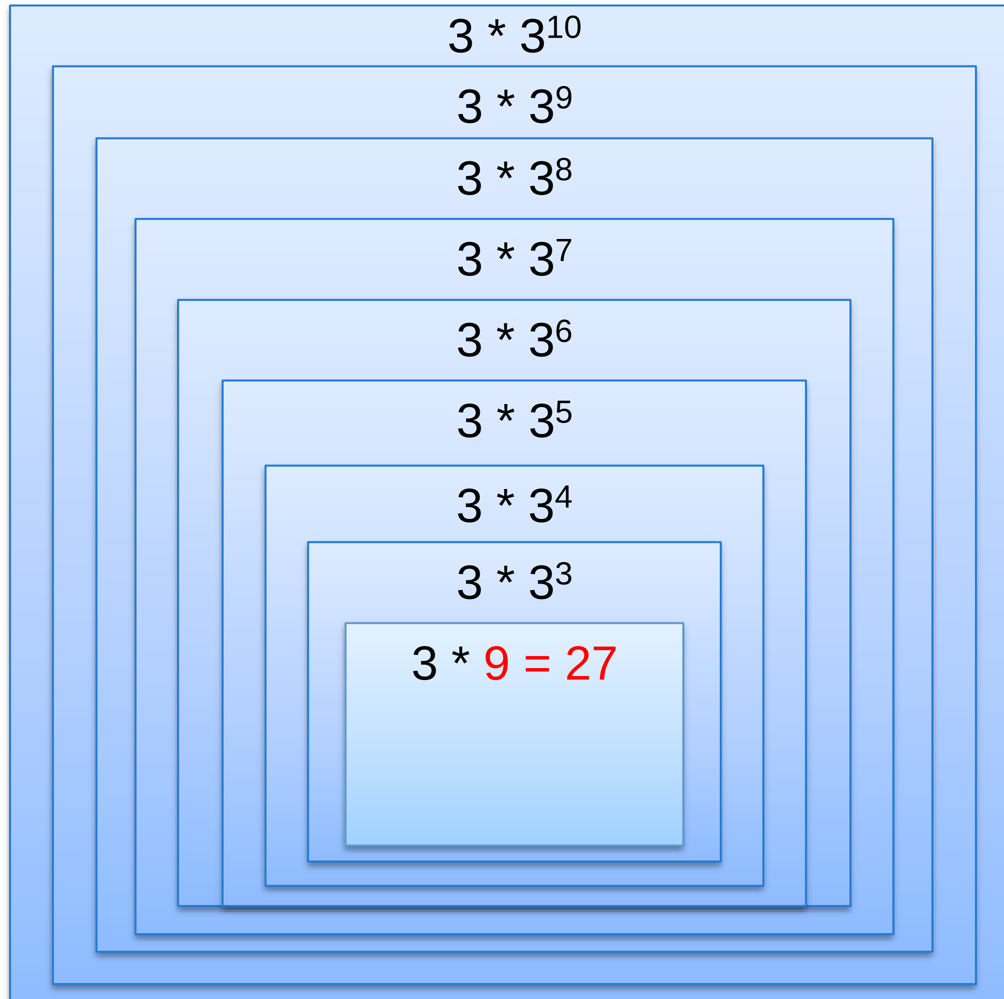
Power a number recursively

- Calculate 3^{11} using recursion



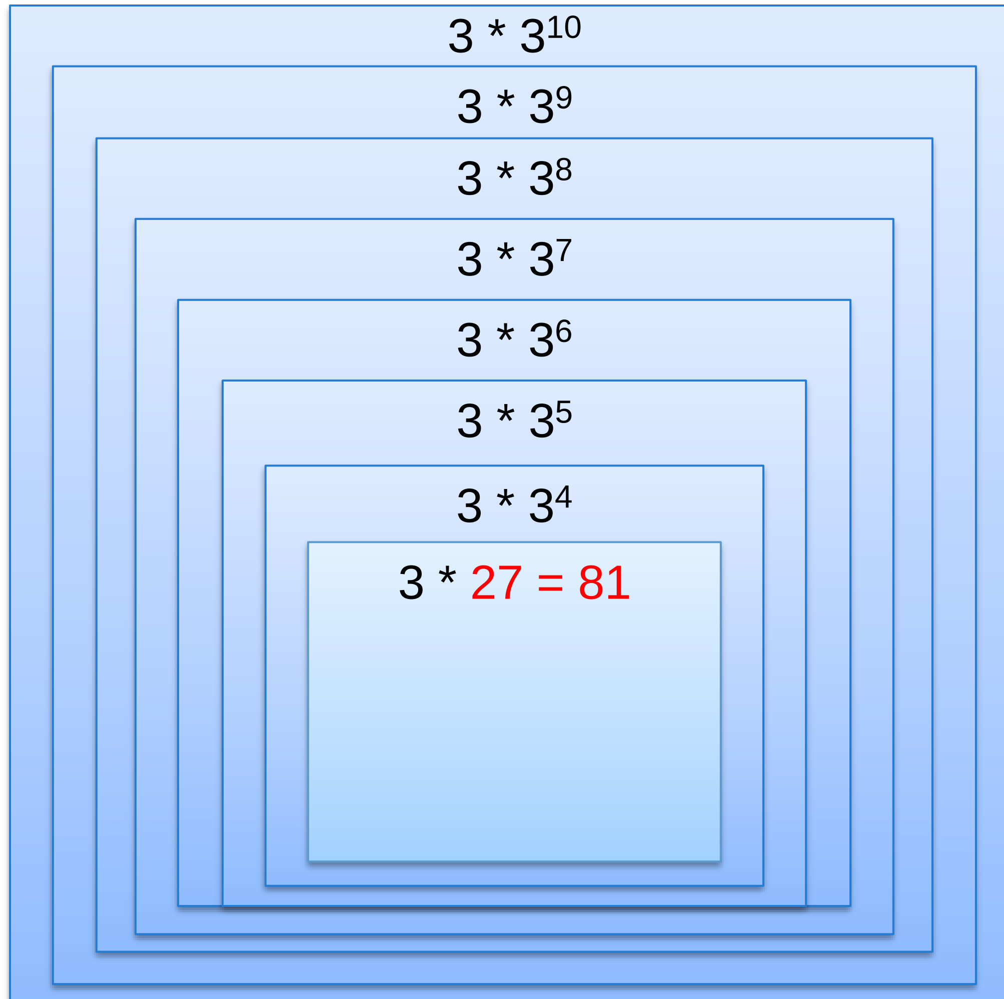
Power a number recursively

- Calculate 3^{11} using recursion



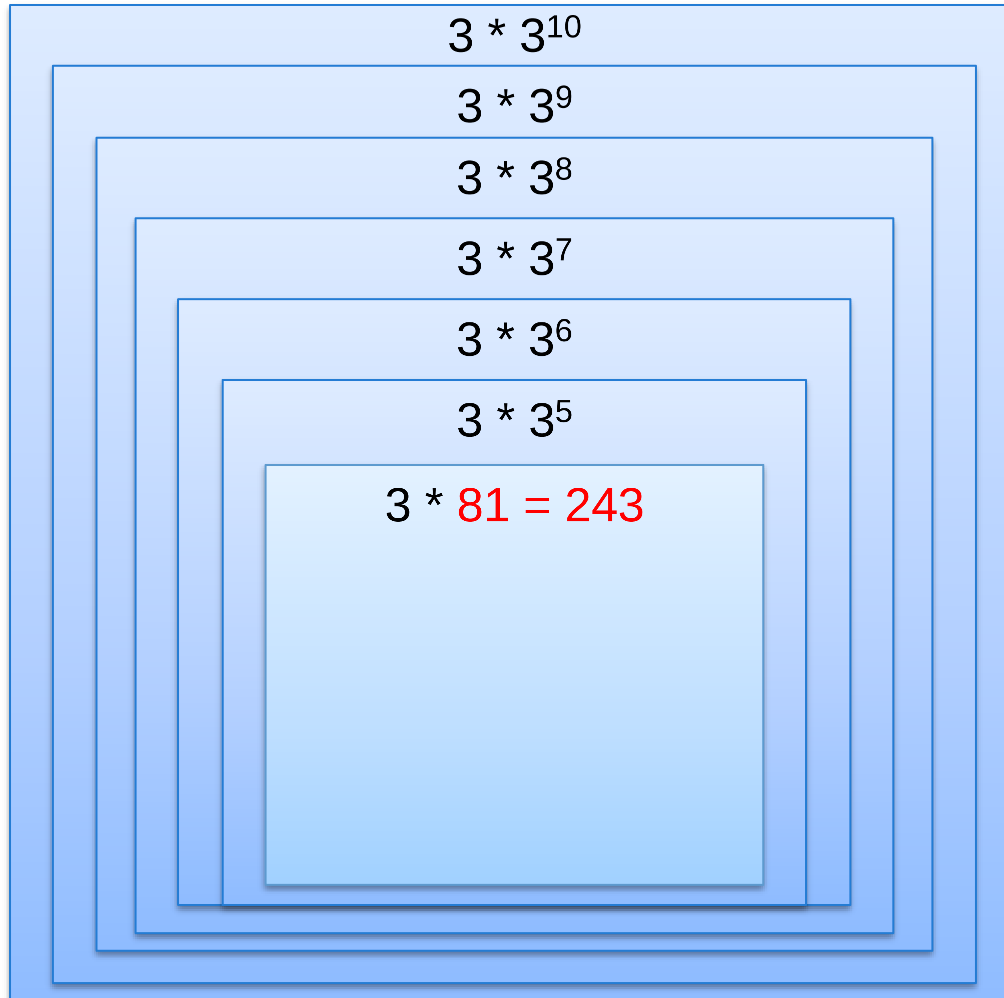
Power a number recursively

- Calculate 3^{11} using recursion



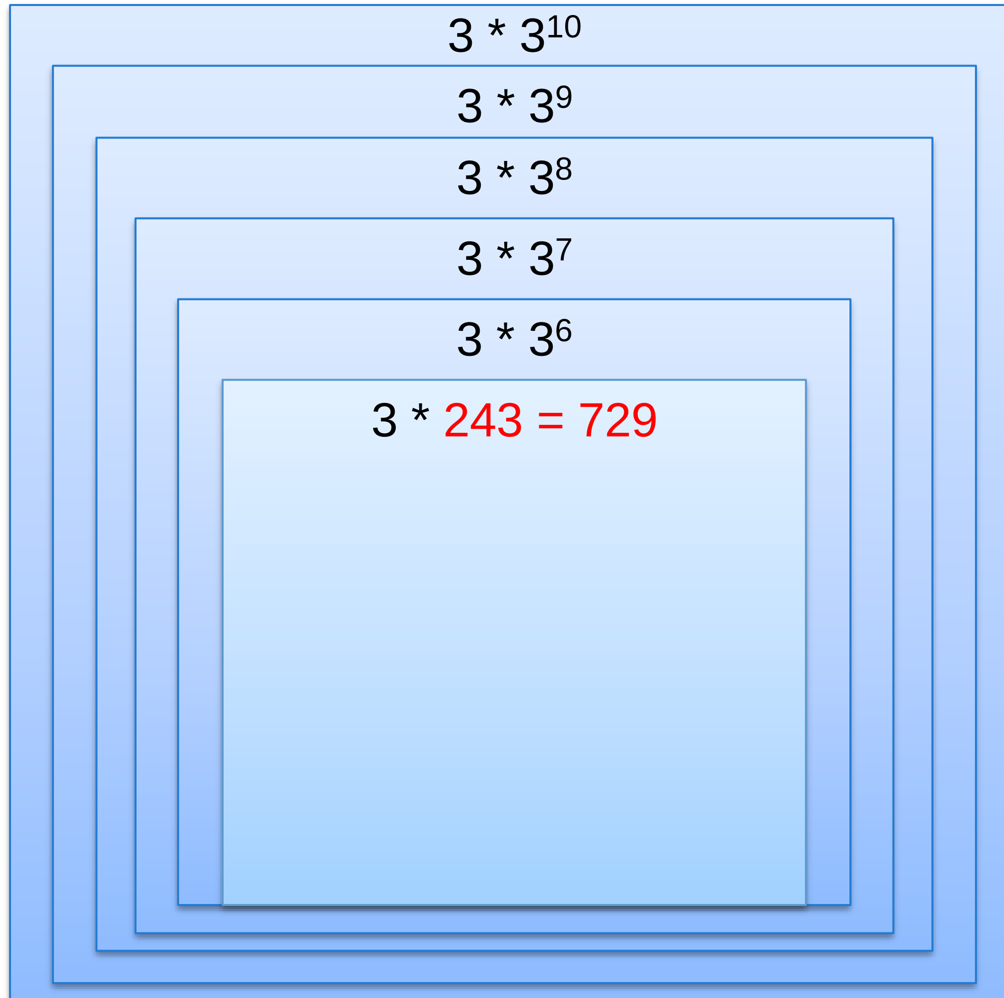
Power a number recursively

- Calculate 3^{11} using recursion



Power a number recursively

- Calculate 3^{11} using recursion



Power a number recursively

- Calculate 3^{11} using recursion

$3 * 3^{10}$

$3 * 3^{9}$

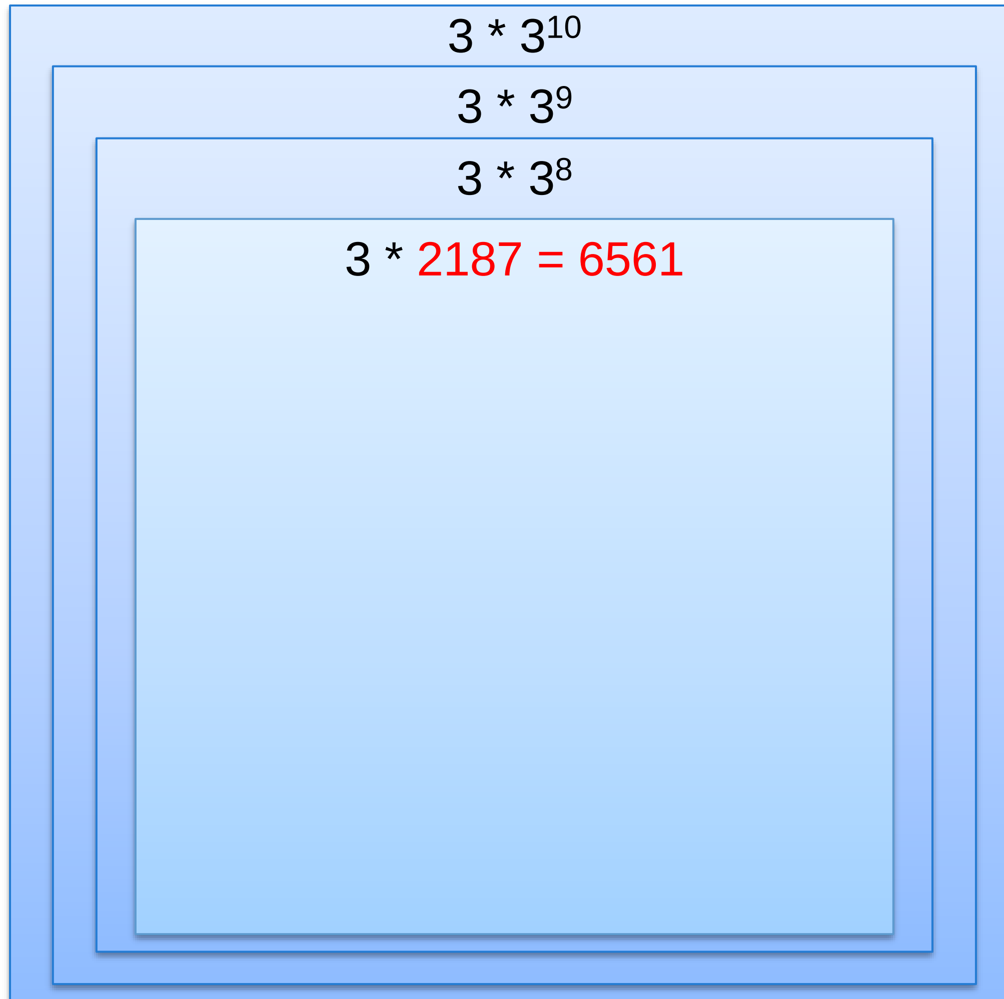
$3 * 3^{8}$

$3 * 3^{7}$

$3 * 729 = 2187$

Power a number recursively

- Calculate 3^{11} using recursion



Power a number recursively

- Calculate 3^{11} using recursion

$3 * 3^{10}$

$3 * 3^{9}$

3 * 6561 = 19683

Power a number recursively

- Calculate 3^{11} using recursion

$3 * 3^{10}$

$3 * 19683 = 59049$

Power a number recursively

- Calculate 3^{11} using recursion

$3 * 59049 = 177147$

Divide and conquer
Powering a number

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

$$3 * 3^2 * 3^2$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

$$3 * 3^2 * 3^2$$

$$3^1 * 3^1$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

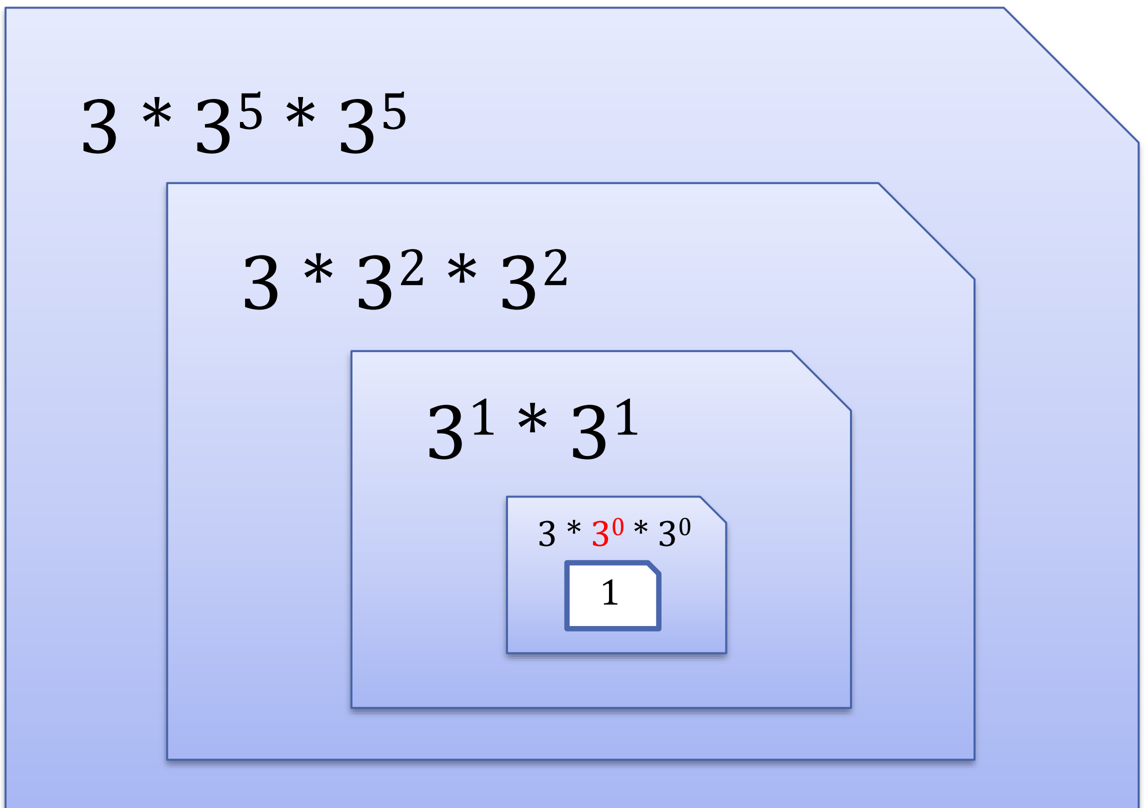
$$3 * 3^2 * 3^2$$

$$3^1 * 3^1$$

$$3 * 3^0 * 3^0$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:



Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

$$3 * 3^2 * 3^2$$

$$3^1 * 3^1$$

$$3 * 1 * 1 = 3$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

$$3 * 3^2 * 3^2$$

$$3 * 3 = 9$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 3^5 * 3^5$$

$$3 * 9 * 9 = 243$$

Power a number with D&C

- Calculate 3^{11} using divide-and-conquer:

$$3 * 243 * 243 = 177{,}147$$

Dynamic Programming

- More Reading…

Next week topic

- Sorting
- Quiz:
  - Review Quiz: Recursion


Q&A