Recursion and Iteration

Department of Electrical and Computer Engineering
McMaster University

January 2020

Recursion

Definition (Recursive Algorithm)

A **recursive** algorithm is one which solves a problem by representing it as an instance of the same problem with a smaller input size.

Definition (Recursive Algorithm)

A **recursive** algorithm is one which solves a problem by representing it as an instance of the same problem with a smaller input size.

Example: Factorial function

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 \\ &= n \cdot (n-1)! \\ &= n \cdot (n-1) \cdot (n-2)! \\ &\quad \cdots\cdots \end{aligned}$$

Factorial function

```c
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

Factorial function

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

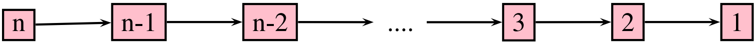
Calling sequence:

n → n-1 → n-2 → .... → 3 → 2 → 1

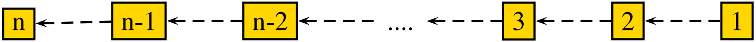
Factorial function

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

Calling sequence:



Return sequence:



Factorial function - Run Time

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

- The running time to complete `factorial(n)` (denoted $T(n)$) equals the running time to complete `factorial(n-1)` (i.e., $T(n-1)$) plus a constant number of operations.

Factorial function - Run Time

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

- The running time to complete `factorial(n)` (denoted $T(n)$) equals the running time to complete `factorial(n-1)` (i.e., $T(n-1)$) plus a constant number of operations.

$$
\begin{aligned}
T(n) &= T(n-1) + c \\
&= T(n-2) + c + c \\
&= T(n-3) + c + c + c \\
&\;\;\vdots \\
&= T(1) + c + \cdots + c + c = T(1) + c(n-1) \\
&= 1 + c(n-1) = cn - c + 1 = \Theta(n)
\end{aligned}
$$

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

- **Space Complexity** (refers to the maximum amount of memory used at any time during execution)
  - At each method call an **activation record (AR)** is pushed onto the stack of ARs.
  - Each AR needs a **constant amount** of memory (of bytes).
  - How big will the stack of ARs grow?

```
int factorial (int n){
    if(n==1)
        return(1);
    else
        return(n*factorial(n-1));
}
```

- **Space Complexity** (refers to the maximum amount of memory used at any time during execution)
  - At each method call an **activation record (AR)** is pushed onto the stack of ARs.
  - Each AR needs a **constant amount** of memory (of bytes).
  - How big will the stack of ARs grow?n **ARs** on top of AR for main().
  - Maximum amount of memory needed (maximum number of bytes) is $S(n) = \Theta(n)$.
  - Space Complexity is $\Theta(n)$.

Rules of Recursion

- **Base case**: (stopping condition) must have at least one case which can be solved without recursion.
- **Making progress**: at each step there must be some progress toward the base case.
  - $\rightarrow$ makes sure stopping condition is eventually met
- **Design rule**: during the design of the recursive algorithm, assume that all simpler recursive calls work.
- **Compound interest**: do not solve the same instance of the problem in more than one recursive call.

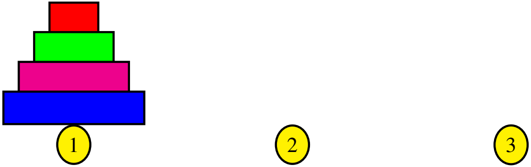
Why use recursion?

- Some problems are more easily solved using recursion.
  - → It is easier to see (and describe) the reduction step rather than trace the whole path of steps bottom up.
- In such cases the recursive solution leads to more compact code, easier to write, understand and debug.

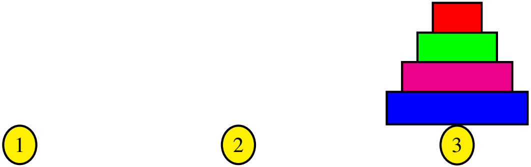
McMaster University

- **Rules**:
  - Move the disks from pile 1 to pile 3, using pile 2 as a temporary holding location.
  - Move one disk at a time.
  - A disk may not be placed on a smaller one.

Start Position:



Finish Position:



Towers of Hanoi

Question: How many moves does it take to do this if we have n disks?
Answer:

Question: How many moves does it take to do this if we have n disks?
Answer:

- With 1 disk ... takes 1 move.
- With 2 disks ... takes 3 moves.
- With 3 disks ... takes 7 moves.

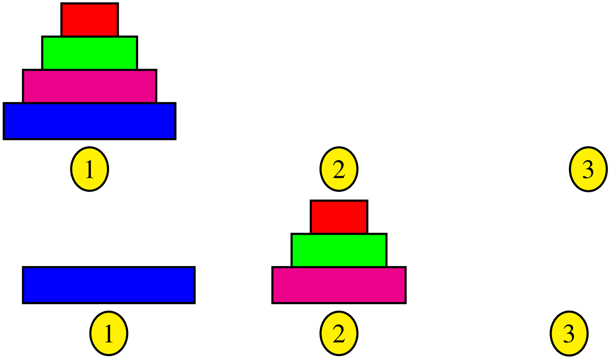
What is the pattern ?

If you consider all the intermediate steps this can be a complicated problem ...

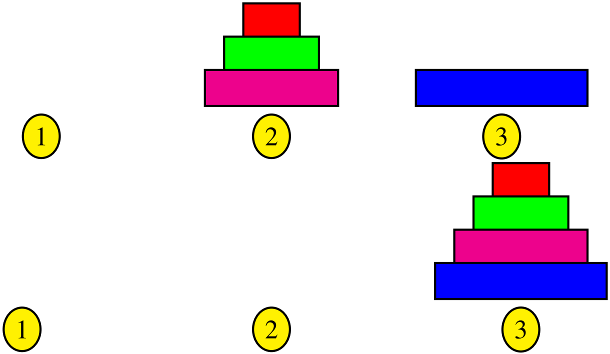
Insight!

- No matter how many disks you start with, in order to move biggest disk to pile 3, need to have $n-1$ disks in pile 2 and then move the largest disk.

Example: Consider $n = 4$ disks,



Example: Consider $n = 4$ disks,

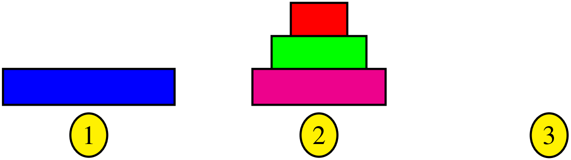


Towers of Hanoi

1. Move $n-1$ disks from pile 1 to pile 2, using pile 3 as temporary holding location
2. Move 1 disk from pile 1 to pile 3
3. Move $n-1$ disks from pile 2 to pile 3, using pile 1 as temporary holding location

1. Move $n-1$ disks from pile 1 to pile 2, using pile 3 as temporary holding location
2. Move 1 disk from pile 1 to pile 3
3. Move $n-1$ disks from pile 2 to pile 3, using pile 1 as temporary holding location

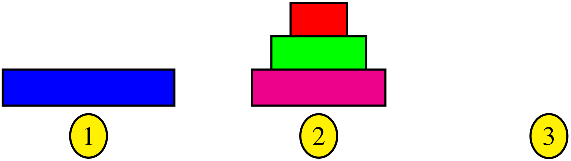
```
void towers(int n, int start, int goal, int temp){
    if(n==1)     /* Base Case */
        System.out.printf("Move from %d to %d \n",start,goal);
    else if(n>1){
        towers(n-1,start,temp,goal)
        System.out.printf("Move from %d to %d \n",start,goal);
        towers(n-1,temp,goal,start);
    }
}
```



Run Time:

- → Base Case: $T(1) = 1$
- → Recursive formula for $n > 1$

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \end{aligned}$$



Run Time: $T(1) = 1$

$$\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1 \\
&= 4(2T(n-3)+1)+2+1 = 2^3T(n-3)+4+2+1 \\
&= \cdots\cdots \\
&= 2^{n-1}T(1)+2^{n-2}+\cdots+2^2+2+1 \\
&= 2^{n-1}\cdot 1+2^{n-2}+\cdots+2^2+2+1 = 2^n-1 \\
&= \Theta(2^n)
\end{aligned}$$

Towers of Hanoi - Run Time

- **Run Time**
  - $\Theta(2^n)$ – **extremely inefficient**

- **Run Time**
  - $\rightarrow$ $\Theta(2^n)$ – **extremely inefficient**
- Consider solving the problem using the fastest supercomputer to date at about 34×10^{15} moves per second.
- Consider $n = 60$.

$$T(n) = 2^{60} - 1 \approx \left(2^{10}\right)^6 \approx 1000^6 = 10^{18}$$

$$10^{18}/(34 \times 10^{15}) \approx 29.5 \text{ seconds}$$

McMaster University

- **Run Time**
  - $\rightarrow$ $\Theta(2^n)$ – **extremely inefficient**
- Consider solving the problem using the fastest supercomputer to date at about 34×10^{15} moves per second.
- Consider $n = 60$.

$$T(n) = 2^{60} - 1 \approx \left(2^{10}\right)^6 \approx 1000^6 = 10^{18}$$

$$10^{18}/(34 \times 10^{15}) \approx 29.5 \text{ seconds}$$

- Consider $n = 100$.

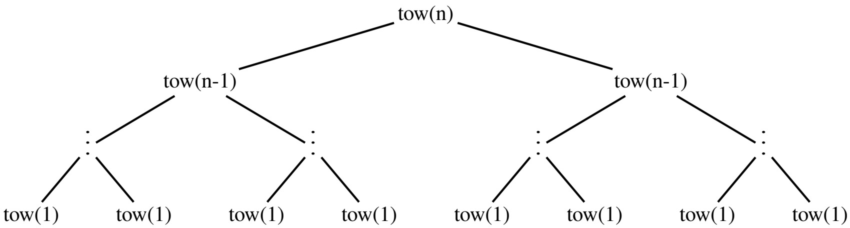
$$T(n) = 2^{100} - 1 \approx \left(2^{10}\right)^{10} \approx 1000^{10} = 10^{30}$$

$$10^{30}/(34 \times 10^{15}) \approx 29.5 \times 10^{12} \text{ seconds} > 900,000 \text{ years}$$

McMaster University

Build a **recursion tree**:

- Each node represents a method call. The children of a node represent the recursive invocations called from the parent.
- The root node corresponds to **towers(n)**. The root has two children, each corresponding to one invocation of **Towers(n-1)**.
- The tree continues in this manner.
- The leaves (nodes without any children) correspond to the base cases.
- To determine $T(n)$ count the number of operations at all tree nodes.



Towers of Hanoi Complexity using Recursion Tree

- The tree has n levels: level 0, level 1, level 2, $\cdots$, level $n-1$.
- The number of nodes on each level s is 2^s.
- The number of operations per node: 1 move.
- $T(n)$ is the total number of moves, i.e., the total number of nodes = nodes at level 0 + nodes at level 1 + nodes at level 2 + ... + nodes at level $n-1$:

$$T(n) = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{n-1} = 2^n - 1.$$

Towers of Hanoi - Space Complexity

- Let $S(n)$ denote the maximum amount of memory used at any time during the execution of **towers(n)**.
  - Recursive formula: $S(n) = S(n-1) + c_1$
  - Base Case: $S(1) = c_2$
  - Solving the recursion leads to $S(n) = c_1(n-1) + c_2 = \Theta(n)$.

- Consider a **different way** to compute $S(n)$.
  - Let us determine how big will the stack of ARs (activation records) grow.
  - Notice that the stack of ARs grows and shrinks multiple times. Peaks are reached every time a base case is reached.
  - **When a peak is reached** the stack contains ARs for **towers(n), towers(n-1), towers(n-2),** $\cdots$ **, towers(1)**. Thus, it contains n **ARs** in all.
  - Since each AR uses a constant amount of memory, we conclude that $S(n) = \Theta(n)$.

Exponentiation

Question: How do you compute X^n for X and n integers?
Solution 1: The direct approach Just apply the definition of exponentiation directly,

$$X^n = \underbrace{X \cdot X \cdot X \cdots X}_{n \text{ factors}}$$

Exponentiation

Question: How do you compute X^n for X and n integers?
Solution 1: The direct approach Just apply the definition of exponentiation directly,

$$X^n = \underbrace{X \cdot X \cdot X \cdots X}_{n \text{ factors}}$$

```
long Pow(long x, int n){
    int i;
    long  temp=1;
    for(i=0;i<n;i++)
        temp *= x;
    return(temp);
}
```

Exponentiation

Question: How do you compute X^n for X and n integers?
Solution 1: The direct approach Just apply the definition of exponentiation directly,

$$X^n = \underbrace{X \cdot X \cdot X \cdots X}_{n \text{ factors}}$$

```
long Pow(long x, int n){
    int i;
    long  temp=1;
    for(i=0;i<n;i++)
        temp *= x;
    return(temp);
}
```

Run Time: $\Theta(n)$ multiplications.

Exponentiation

Solution 2: Recursive Algorithm Notice that,

$$X^n = \begin{cases} (X^2)^{n/2} & : \quad n \text{ even} \\ X \cdot (X^2)^{(n-1)/2} & : \quad n \text{ odd} \end{cases}$$

Exponentiation

Solution 2: Recursive Algorithm Notice that,

$$X^n = \begin{cases} (X^2)^{n/2} & : \quad n \text{ even} \\ X \cdot (X^2)^{(n-1)/2} & : \quad n \text{ odd} \end{cases}$$

```
long Pow(long x, int n){
     if(n==0)
          return(1);
     if(iseven(n))
          return(Pow(x*x,n/2));
     else
          return(Pow(x*x,n/2)*x);
}
```

```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$X^{62} =$$

$$=$$

$$=$$

Exponentiation

```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$
\begin{aligned}
X^{62} &= (\underbrace{X^2})^{31} \\
&= \\
&=
\end{aligned}
$$

Exponentiation


```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$
\begin{aligned}
X^{62} &= (\underbrace{X^2})^{31} = X^2(\underbrace{(X^2)^2})^{15} \\
&= \\
&=
\end{aligned}
$$

```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$X^{62} = (\underbrace{X^2}_{})^{31} = X^2(\underbrace{(X^2)^2}_{})^{15} = X^2(X^2)^2(\underbrace{((X^2)^2)^2}_{})^7$$

$$=$$

$$=$$

Exponentiation

```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$
\begin{aligned}
X^{62} &= (\underbrace{X^2}) ^{31} = X^2(\underbrace{(X^2)^2})^{15} = X^2(X^2)^2(\underbrace{((X^2)^2)^2})^7 \\
&= X^2(X^2)^2((X^2)^2)^2(\underbrace{(((X^2)^2)^2)^2})^3 \\
&=
\end{aligned}
$$

Exponentiation

McMaster University

```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$
\begin{aligned}
X^{62} &= (\underbrace{X^2}_{})^{31} = X^2(\underbrace{(X^2)^2}_{})^{15} = X^2(X^2)^2(\underbrace{((X^2)^2)^2}_{})^7 \\
&= X^2(X^2)^2((X^2)^2)^2(\underbrace{(((X^2)^2)^2)^2}_{})^3 \\
&= X^2(X^2)^2((X^2)^2)^2(((X^2)^2)^2)^2((((X^2)^2)^2)^2)^2
\end{aligned}
$$

Exponentiation


```
long Pow(long x, int n){
    if(n==0)
        return(1);
    if(iseven(n))
        return(Pow(x*x,n/2));
    else
        return(Pow(x*x,n/2)*x);
}
```

Example:

$$
\begin{aligned}
X^{62} &= (\underbrace{X^2})^{31} = X^2(\underbrace{(X^2)^2})^{15} = X^2(X^2)^2(\underbrace{((X^2)^2)^2})^7 \\
&= X^2(X^2)^2((X^2)^2)^2(\underbrace{(((X^2)^2)^2)^2})^3 \\
&= X^2(X^2)^2((X^2)^2)^2(((X^2)^2)^2)^2((((X^2)^2)^2)^2)^2
\end{aligned}
$$

- **Run Time**
  - → $\Theta(\log n)$ (problem size reduced by $1/2$ each call)

Iteration

Definition (Iterative Algorithm)

An **iterative** algorithm is one which solves a problem by computing the required value using the current input and the results of previous inputs usually in a loop.

- In application, these types of routines differ from recursive ones since they do not call themselves.

Definition (Iterative Algorithm)

An **iterative** algorithm is one which solves a problem by computing the required value using the current input and the results of previous inputs usually in a loop.

- In application, these types of routines differ from recursive ones since they do not call themselves.

Example: Factorial function

```
int factorial (int n){
    int count,r=1;
    for(count=1;count<=n;count++)
        r=r*count;
    return(r);
}
```

McMaster University

Definition (Iterative Algorithm)

An **iterative** algorithm is one which solves a problem by computing the required value using the current input and the results of previous inputs usually in a loop.

- In application, these types of routines differ from recursive ones since they do not call themselves.

Example: Factorial function

```
int factorial (int n){
    int count,r=1;
    for(count=1;count<=n;count++)
        r=r*count;
    return(r);
}
```

- **Run Time**: $\Theta(n)$

McMaster University

Definition (Iterative Algorithm)

An **iterative** algorithm is one which solves a problem by computing the required value using the current input and the results of previous inputs usually in a loop.

- In application, these types of routines differ from recursive ones since they do not call themselves.

Example: Factorial function

```
int factorial (int n){
    int count,r=1;
    for(count=1;count<=n;count++)
        r=r*count;
    return(r);
}
```

- **Run Time**: $\Theta(n)$
- **Space Complexity**: $O(1)$
  - $\rightarrow$ rather than $O(n)$ of recursive algorithm.

Binary Search

It is possible to write the binary search of the previous topic as a recursive function ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

Search for `K` in a pre-sorted list of length `n`.

```
int binsrch(list x[], int srch, int left, int right){
    int mid= (left+right)/2;          /* Returns floor of division */

    if(x[mid]==srch)
        return(mid);
    else if(x(mid)>srch && mid>left)
        return binsrch(x,srch,left,mid-1);
    else if (x(mid)<srch && right>mid)
        return binsrch(x,srch,mid+1,right)
    else
        return UNSUCCESSFUL;
}
```

- **Run Time**:
- **Space Complexity**:

Recursive Binary Search

- **Run Time**: $\Theta(\log n)$
- **Space Complexity**:

Recursive Binary Search

- **Run Time**: $\Theta(\log n)$
- **Space Complexity**: $\Theta(\log n)$
  - → Non-recursive binary search has $\Theta(1)$ space complexity.

Multiple Recursion

- When the recursive method calls itself **more than once**.
  - → **Example**: Towers of Hanoi has **exponential time complexity**
- Does multiple recursion leads always to exponential time complexity?

Multiple Recursion

- When the recursive method calls itself **more than once**.
  - → **Example**: Towers of Hanoi has **exponential time complexity**
- Does multiple recursion leads always to exponential time complexity?
  - → If multiple recursive calls reduce the problem size by only **by a constant amount** $\Rightarrow$ exponential running time.
  - → When number of recursive calls is constant and each recursive call reduces the problem **by a constant ratio** - may lead to efficient algorithms (depending on running time of reduction step).
    - **Divide and Conquer** strategy - to be discussed later (e.g., MergeSort).

Iterative versus Recursive Algorithms

- Recursive algorithms usually require more memory and time than iterative but are easier to interpret.
  - Function calls can be expensive and take a lot of time to complete.
  - Need to make sure that all the rules of recursion are met.
  - Using recursion for the numerical computation of simple functions is usually not a good idea.
- Every recursive solution has a non-recursive alternative

Next Topic ...

- Hashing