Recursion

CSE 2320 – Algorithms and Data Structures
University of Texas at Arlington

Updated: 2/21/2018

Background & Preclass Preparation

- Background (review):
  - Recursive functions
    - Factorial – must know how to write a recursive solution.
    - Fibonacci
  - C - function call, recursive function execution, local variables

- See next page for a list of problems to think of before class.

Preclass Preparation

- Think about these problems and write your recursive solutions on paper and bring it to class for your own reference. If stuck, write down where and what confuses you. Ask clarification questions in class.
  - Print array
  - compute the sum of all the elements in an array
  - find the index of the smallest element in an array
  - recursive implementations for selection sort and insertion sort. In particular, first think about how to replace their outer loop with recursion (and keep the inner one still as a loop). (Next you can think about replacing the inner loop with recursion and writing everything with just recursion, no loops.)
- How would you solve (even without recursion) the N-queens pb?
  - Place N queens on an NxN chess board so that they do not attach each other.
  - See this wikipage for images: https://en.wikipedia.org/wiki/Eight_queens_puzzle
    - Do not try to read and understand the solutions from there.

Objectives

- Understand
  - Recursive function execution (given the code).
  - How to approach a problem when looking for a recursive solution to it.
  - (C code issues that may come up (and cause bugs) when writing recursive functions.) – Self study

  - Difference between tail-recursion and non-tail recursion.

  - How to write the time complexity recurrence formula for recursive functions.
    - Solving the recurrences is in the next presentation.

Recursion

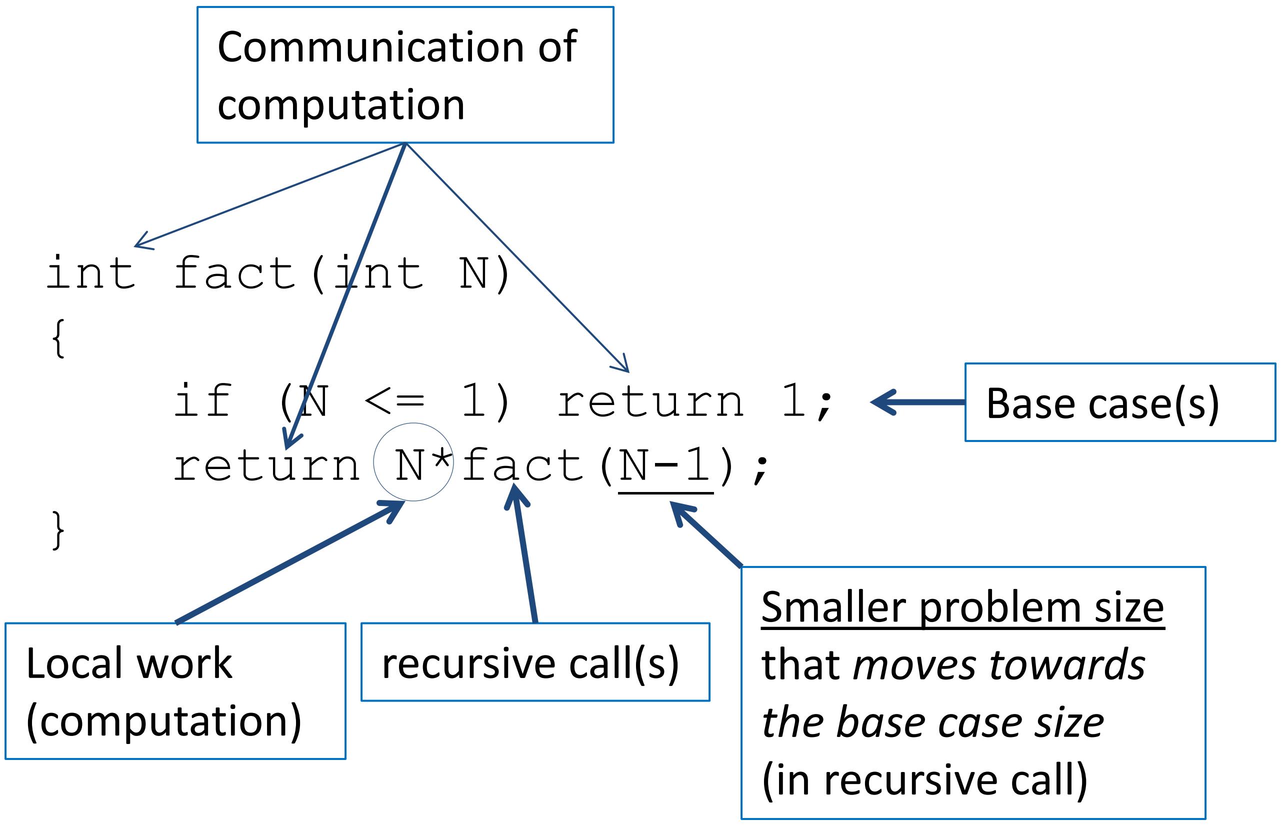
- Recursion is a fundamental concept in computer science.
  - In all recursive concepts, there are one or more **base cases**.
- **Recursive [math] functions**: functions that call themselves.
  - Example: N! = N * (N-1)!
    - Base case: N = 0

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Used as benchmark for compiler optimization for recursion.

- **Recursive data types**: data types that are defined using references to themselves.
  - Example: Nodes in the implementation of linked lists.
    - Base case: NULL
- **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.
  - Example: binary search, (also mergesort, functions for trees)
    - Base case: one or no element in collection.
- Draw fractals:
  - http://web.cs.ucdavis.edu/~amenta/s12/fractalPlant.pdf
  - http://interactivepython.org/runestone/static/pythonds/Recursion/pythondsintro-VisualizingRecursion.html

Components of recursive functions



```
int fact(int N)
{
    if (N <= 1) return 1;
    return N*fact(N-1);
}
```

Communication of computation
Base case(s)
Local work (computation)
recursive call(s)
Smaller problem size that *moves towards the base case size* (in recursive call)

What is the ***problem size***?

The variable that controls /influences how much work will be done. E.g.:
- N for factorial,
- Size of array to be sorted …

Each component is needed! What happens if you remove it?

Proof by induction can be used to show it finishes and computes the correct result.

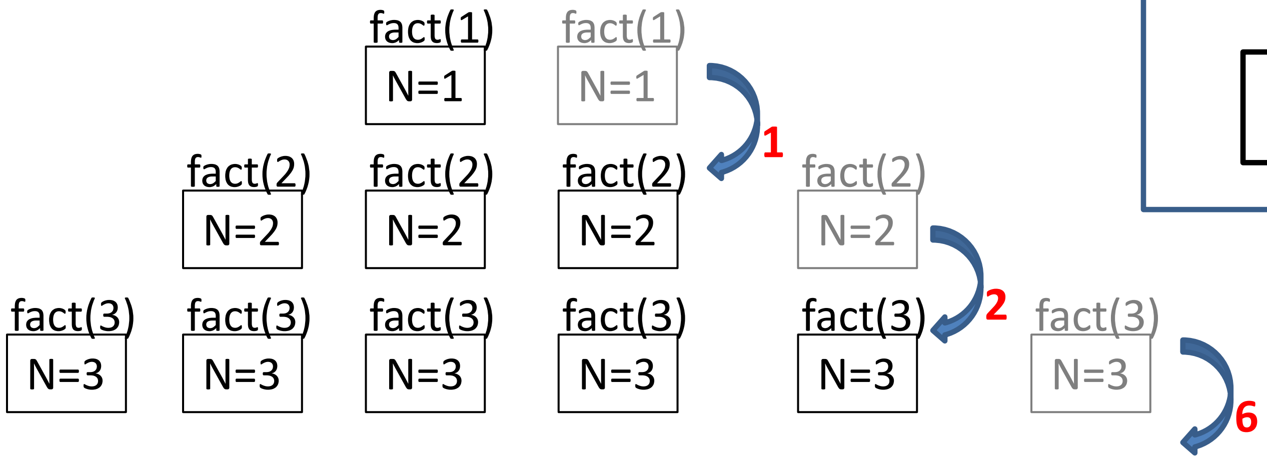
You can see correctness of factorial in extra materials at the end.

Recursive Function Execution

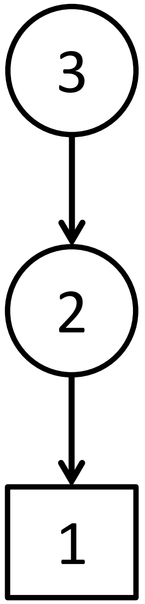
Recursive:

```
int fact(int N)
{
   if (N <= 1) return 1;
   return N*fact(N-1);
}
```

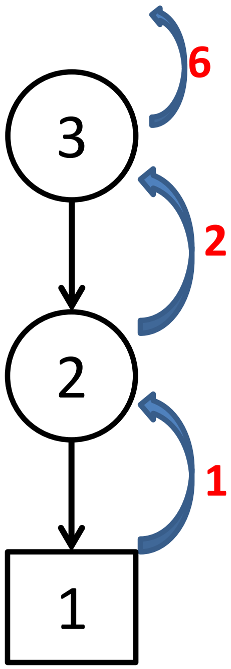
Show fact(3) execution (and call stack):



Tree showing the recursive function calls for fact(3):

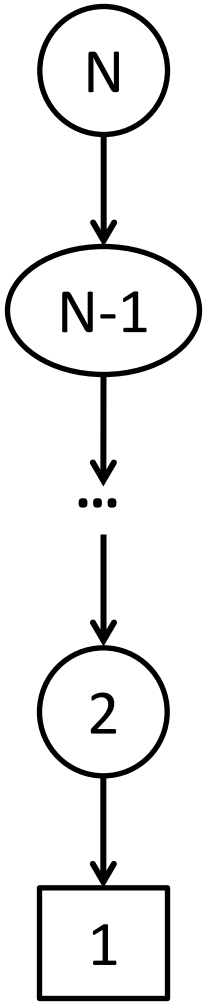


Tree showing the recursive function calls for fact(3) and the return values:



Function Call Tree for fact(N)

Function call tree convention: write problem size, N, in the node. (N is the function argument)



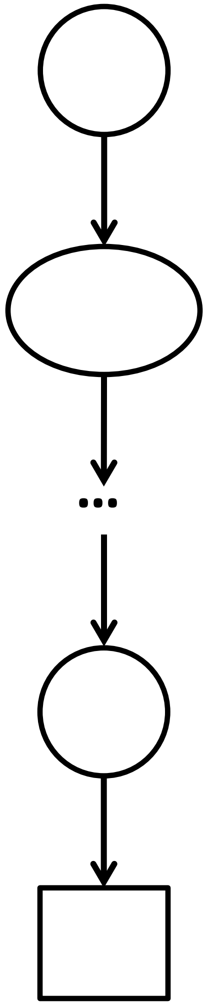
```
int fact(int N)
{
    if (N <= 1) return 1;
    return N*fact(N-1);
}
```

Time complexity of fact(N) ? T(N) = …

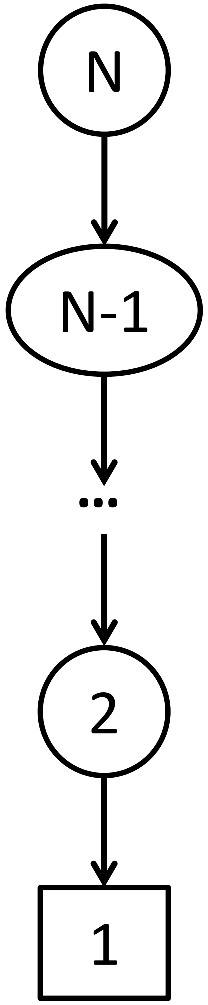
T(N) =

Trees for fact(N)

Time complexity tree:



Function call tree:



Function call tree convention: write problem size, N, in the node.

Time complexity convention: write T(N) outside the node and local cost in the node.

```
int fact(int N)
{
    if (N <= 1) return 1;
    return N*fact(N-1);
}
```

Time complexity of fact(N) ? T(N) = …

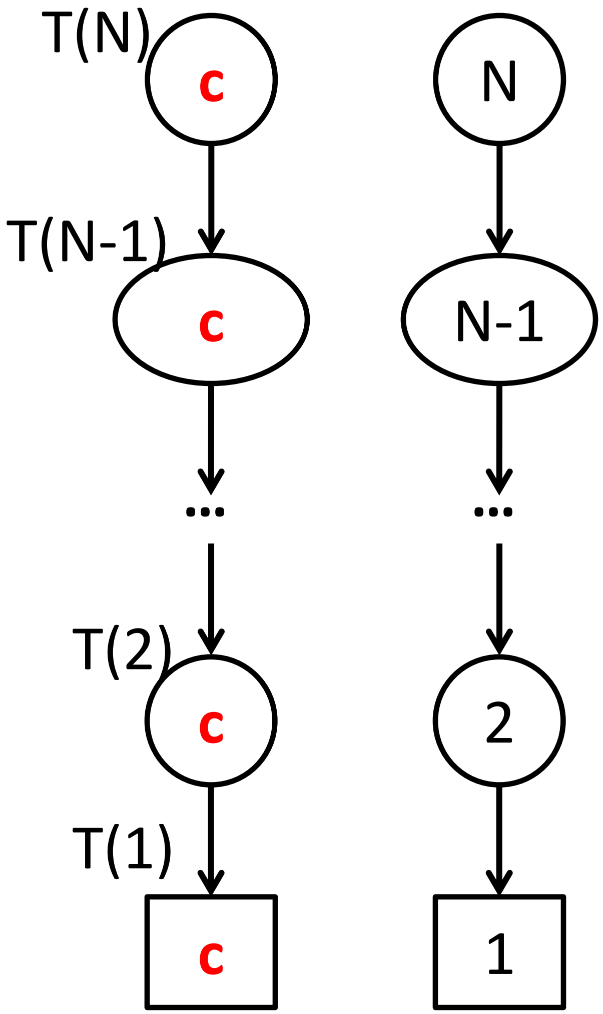
T(N) = T(N-1) + c
T(1) = c

(It works just as well with T(1) = d, but
we will use same constant cost, c, for both
local cost in recursive case and cost of base case)

Trees for fact(N)

Time complexity tree:

Function call tree:



Function call tree convention: write problem size, N, in the node.

Time complexity convention: write T(N) outside the node and local cost in the node.

```
int fact(int N)
{
    if (N <= 1) return 1;
    return N*fact(N-1);
}
```

Time complexity of fact(N) ? T(N) = …

T(N) = T(N-1) + c
T(1) = c

Addressing the inefficiency of recursive functions: Tail-recursion

Tail-recursion

- There is only one recursive call.
- The recursive call is returned directly, not used in a computation.
  - E.g. tail recursion: `return factorial(…);`
  - E.g. not tail recursion: `return N*factorial(…);`
- Where/how will the work be done?

Tail-Recursive Function Execution Worksheet

Tail-recursive (pass and return the answer):

```
int fact_pr(int N, int res){
    if (N <= 1) return res;
    res = res * N;
    return fact_pr(N-1, res);
}
// Wrapper function (sets parameters).
int fact_pr_wrapper(int N) {
    return  fact_pr(N, 1);
}
```

A function is TAIL-recursive if:

- It has just one recursive call
- Has no work left to do after the recursive call.

Show fact_pr(3,1) execution and stack.

Tail-Recursive Function Execution

Answers

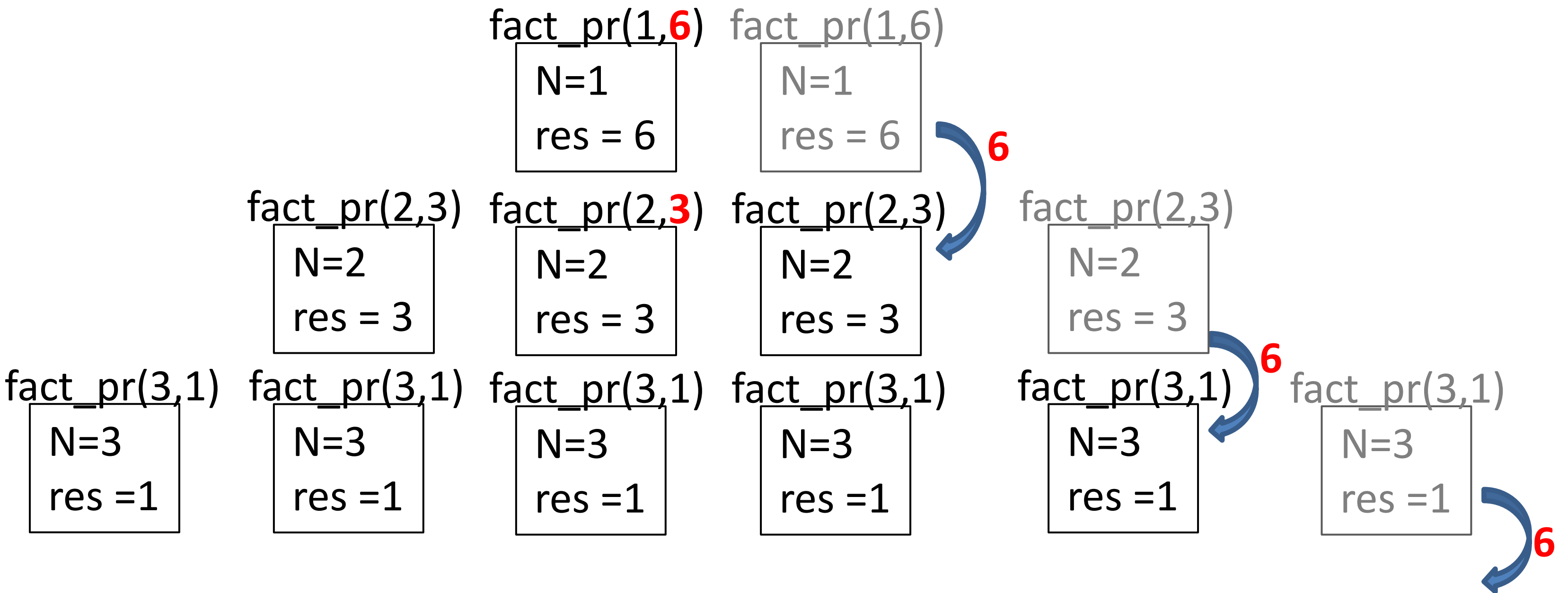
Tail-recursive (pass and return the answer):

```
int fact_pr(int N, int res){
    if (N <= 1) return res;
    res = res * N;
    return fact_pr(N-1, res);
}
// Wrapper function (sets parameters).
int fact_pr_wrapper(int N) {
    return  fact_pr(N, 1);
}
```

A function is TAIL-recursive if:

- It has just one recursive call
- The last instruction is just the recursive call.

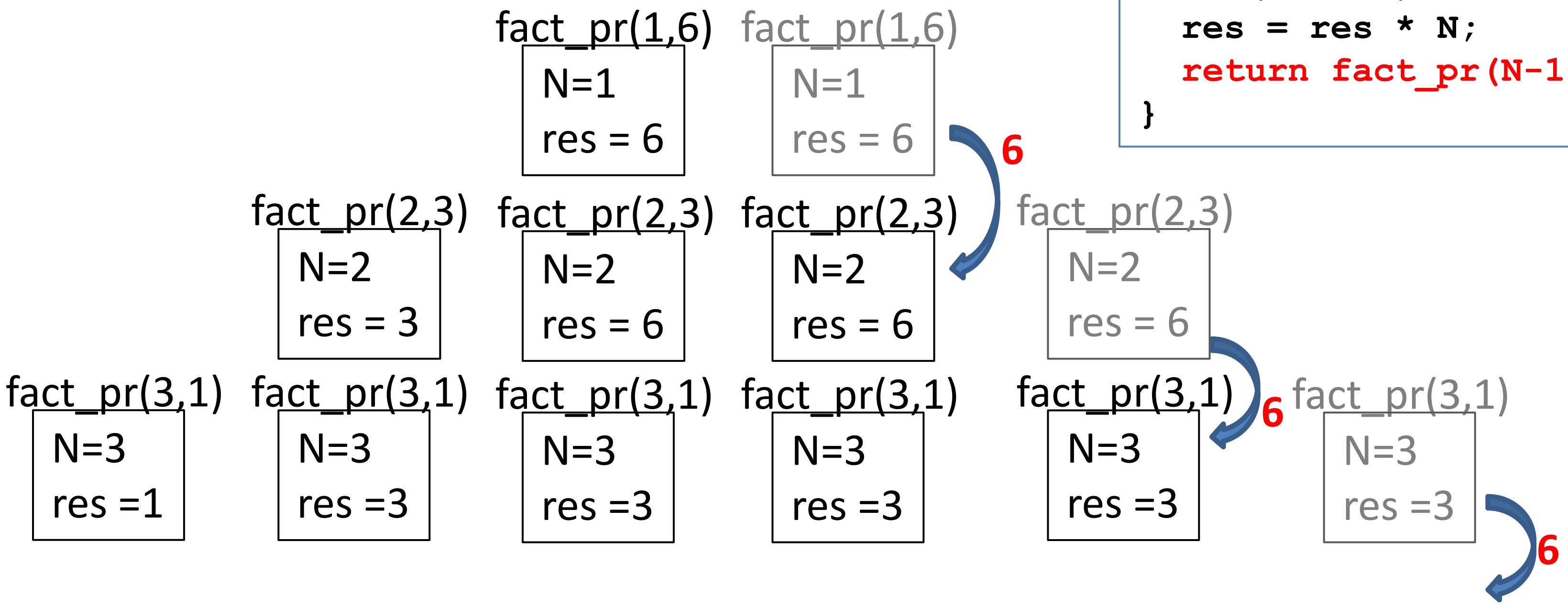
Note: it uses **res** to pass the current computation and the `return` to pass the final result.



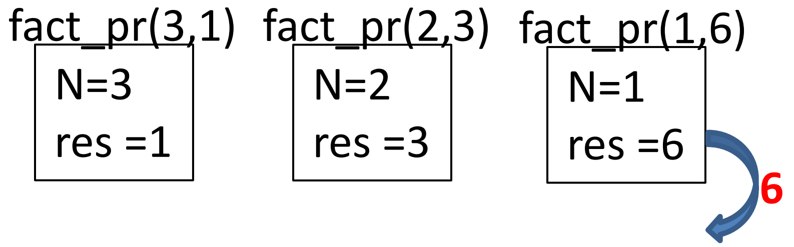
Compiler optimization: no frame stack for tail recursive

Tail-recursive (pass and return the answer):

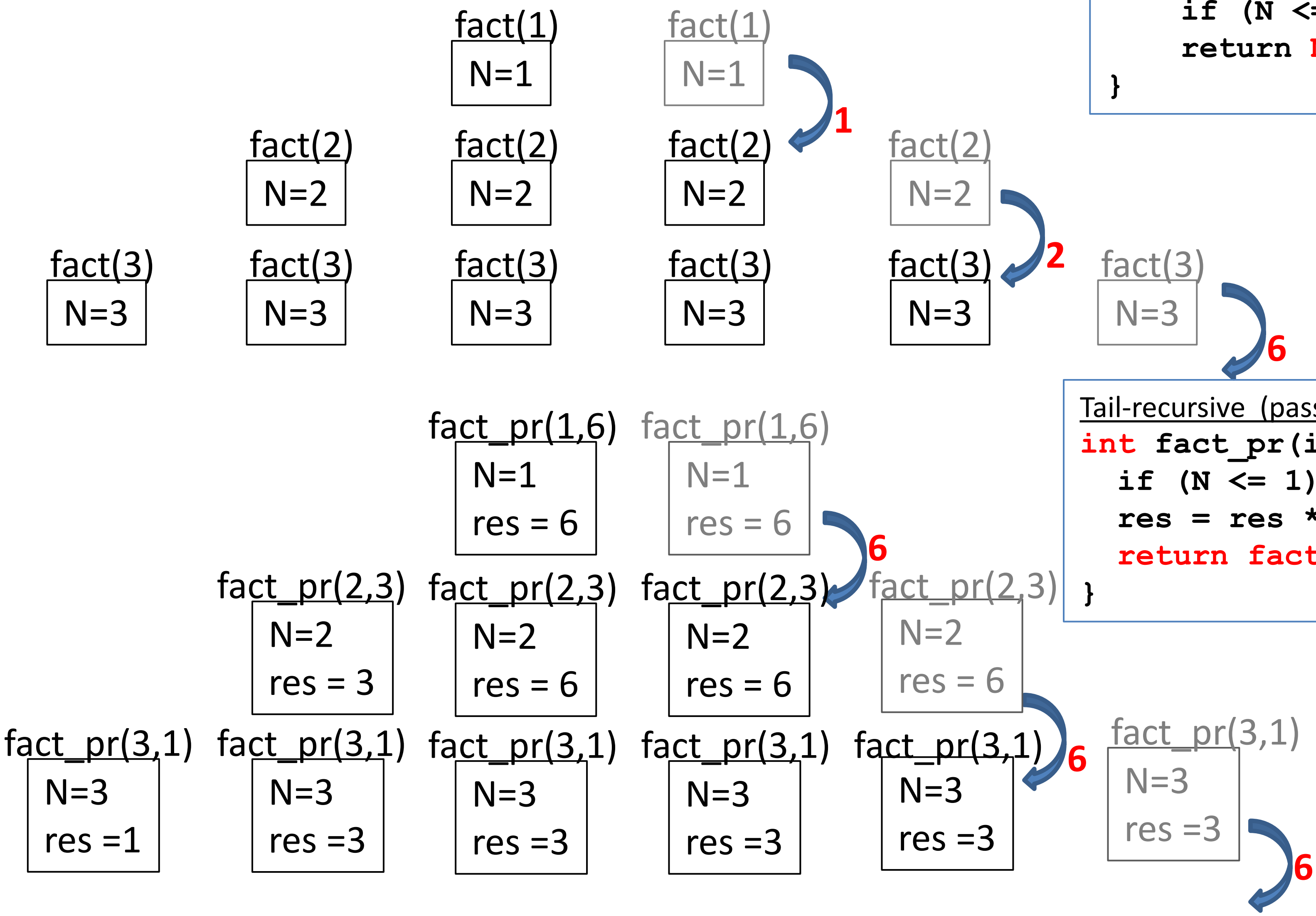
```
int fact_pr(int N,int res){
   if (N <= 1) return res;
   res = res * N;
   return fact_pr(N-1,res);
}
```



Behavior when compiler has optimization for tail-recursive functions.
Instead of building a stack, it replaces the caller stack frame with the callee stack frame.



Compare the two implementations:



NOT tail-recursive :

```
int fact(int N) {
    if (N <= 1) return 1;
    return N*fact(N-1);
}
```

Tail-recursive (pass & ret the answer):

```
int fact_pr(int N,int res){
  if (N <= 1) return res;
  res = res * N;
  return fact_pr(N-1,res);
}
```

Communication of computation

Recursive, not tail-recursive: **return answer**

```
int fact_ret(int N) {
   if (N <= 1) return 1;
   return N*fact_ret (N-1);
}
```

Tail-recursive: **pass and return answer**

```
int fact_pr(int N, int res) {
   if (N <= 1) return res;
   res = res * N;
   return fact_pr(N-1, res);
}
 // Wrapper  function (sets parameters).
int fact_pr_wrapper(int N) {
   int res = 1;
   return  fact_pr(N, res); // res =?
}
// Note that it combines passing data
with returning data.
```

Tail-recursive: answer in **updated argument** (pointer)

```
void fact_update(int N, int* res) {
   if (N <= 1) return;
   (*res) = (*res) * N;
   fact_update (N-1, res);
}
 // Wrapper  function (sets parameters).
int fact_update_wrapper (int N) {
   int res = 1;
   fact_update(N, &res);
  // note diff between N and  res when fct finishes
   return res;
}
```

What is the local work/computation? Is it done before or after the recursive call?

Difference (tail/non-tail recursive): Do the local computation before or after the recursive call.

Worksheet

- Show the stack frame for `fact_update(int N, int * res)`.
  - Pay attention to the fact that `res` is a pointer.

Parameters: Pass-by-Value or Pass-by-reference?

- Pass by reference (& in C++ or pointer in C) when a value computed in a recursive call must be available (is needed) after that call finished.
- Pass-by-value if
  - You only need to use the value in that recursive call.
  - If when a recursive call on a smaller problem finishes and returns, you need to come back to the values for that call
    - See the use of column indexes in the Queens problem (when you backtrack you get to a smaller column, corresponding to that call)
    - See the recursive implementation for generating permutations without repetitions. See how the backtracking takes you to the array before the changes done in the recursive call.

Recursive Vs. Non-Recursive Implementations

- Recursive functions can be easier to read.
  - They are simpler (less code, fewer loops, “smaller problem”), follow the math definition.
  - To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.
- Oftentimes recursive functions run slower. Why?
  - Recursive functions generate many function calls. The CPU has to pay a price (perform a certain number of operations) for each function call.
  - Possible solution: use tail-recursion when possible (some compilers have optimizations for it)
- Any recursive function can also be written in a non-recursive way.
  - Non-recursive implementations can be uglier (and more buggy, harder to debug) but more efficient.
    - See Ackerman recursive and non-recursive.
  - Compromise: make first version recursive, second non-recursive.
    - Use the recursive one to test the correctness of the non-recursive one.

Problem Solving: Recursive Solution

- Idea: Write the solution for the current problem using answers for smaller problems and some local computations.
- Steps:
  - Understand the problem and be able to solve it on paper.
  - Visualize the process and break-down components:
    - Think about the data that you use/generate and
    - What you do with that data.
  - Identify smaller sub-problems
    - Either do some processing an be left with a smaller problem or:
    - Given the answer to a smaller problem and local computations, solve the original problem
  - Assume you have the answer to the smaller problems. (your own recursive call will give that answer)
  - How do you combine or use those results to solve your original (big) problem?
    - How will you communicate the calculations?
  - How will this cycle stop?
    - Identify the smallest/trivial problem that you already know the answer for.

Implementations for N! Worksheet

- Iterative: `fact_iter`
- Recursive – however you want: `fact`
  - What would be a smaller problem s.t. if you know the answer for that, you have less work to do.
  - Problem decomposition: N! = 1*2*…*(N-1)*N
    - Must generate the values: 1,2,3,…,N
    - Must compute the cumulative result (product in this case)
    - Pass the cumulative result to or from recursive function calls
- Tail-Recursive – pass and return the answer:

```
int fact_pr(int N, int res)
```

- Tail-Recursive – no return. Updates pointer argument:

```
void fact_update(int N, int* res)
```

- Can you give a recursive function for N! that makes 2 recursive calls?
  - E.g. break the problem in 2 halves.

More N! Implementations:

- Write the recurrence formula for each of the functions below.
- Draw the trees & make the tables
- Derive time complexity

- N! - That uses an upper bound to stop.
- N! - That has two recursive calls (e.g. on ‘half’ the problem size)

Recursive Functions for Linked Lists Worksheet

- Example: **int count(link x)**
  - count how many links there are between x and the end of the list (x should be included in the count).
  - Recursive solution?
  - Base case?
  - Recursive function?

Recursive Functions for Linked Lists Answers

- Example: **int count(link x)**
  - count how many links there are between x and the end of the list (x should be included in the count).
  - Recursive solution? count(x) = 1 + count(x->next)
  - Base case: x = NULL, count(NULL) = 0.
  - Recursive function:

```
int count(link x)
{ if (x == NULL) return 0;
   return 1 + count(x->next);
}
```

Practice Recursive Implementations:

- Write the recurrence formula for each of the functions below.
- N! (That uses an upper bound to stop. That has two recursive calls (e.g. on 'half' the problem size))
- Binary search
- Find max in an array
- Sum of elements in an array
- Selection sort
  - Use recursion to do the work of the outer loop. (Extra: also for inner loop)
  - Remember the sorting process. What would be a smaller problem?
- Place N queens on an NxN checkerboard.
  - See online visualization (recursive fct calls, stored data, visual aid board)
    - https://www.cs.usfca.edu/~galles/visualization/RecQueens.html
- Generate permutations with repetitions
- Generate permutations without repetitions
- Merge sort ?

Recursive Array Sum

- Give a recursive method to add all the elements of an array A of size N.
- Guiding questions
  - What constitutes a smaller problem?
  - If you have the answer to the smaller problem, what do you have to do to get the answer to your current(original) problem?
  - What other problem is this similar to?
- Code
  - Any recursive solution
  - Tail-recursive solution
- Now that we solved it, what other problem is this similar to?

Binary Search - Recursive

```
/* Adapted from Sedgewick
*/
int search(int A[], int left, int right, int v)
  { int m = (left+right)/2;
    if (left > right) return -1;   // not found
    if (v == A[m]) return m;
    if (left == right) return -1;
    if (v < A[m])
         return search(A, left, m-1, v);  // recursive call
    else
         return search(A, m+1, right, v);  // recursive call

  }
```

- How many recursive calls?
- Any correspondence between the recursive and non-recursive implementations?

Self-Study

C Code Discussion

- The data computed by a recursive function can be 'passed back up' to the caller function in 2 ways:
  - Actually returned (left example)
  - By modifying a reference variable (through a pointer) (right example)
    - This is needed for tail-recursion

```c
int fact(int N) {
   if (N <= 0) return 1;
   return N*fact(N-1);
}

int main() {
   int N = 3;
   int res = fact(N);
   printf("N = %d , res = %d ", N, res);
}
```

```c
void fact_update(int N, int* res) {
   if (N <= 0) return;
   (*res) = (*res) * N;
   fact_update (N-1, res);
}
 // Wrapper function to set-up parameters.
int fact_update_wrapper(int N) {
   int res = 1;
   fact_update (N, &res);
  // note diff between N and  res when fct finishes
   return res;
}
```

TRAPS: Pointers to Local Variables in C

- Pointers to local variables.
  - OK to pass to the fct being called (e.g. fact_tail_helper) a reference/pointer to a local variable (e.g. &res).
  - BAD to return a pointer to a local variable.

```
void fact_tail_helper(int N, int* res) {
   if (N == 0) return;
   (*res) = (*res) * N;
   fact_tail_helper (N-1, res);
}

int main() {
   int N = 3;
   int res = 1;
   fact_tail_helper (N, &res);  // Ok. out->in
   printf("N = %d , res = %d ", N, &res);
}
```

```
// BAD. Incorrect.
// Wrong ‘direction’: in->out
int* test() {
   int local_res = 10;
   return &local_res;  // bad
}

int main() {
   int * res = test();
   printf("res = %d ", *res);
}
```

Variables: Local vs Static

- What will fact_v3(3) evaluate to?
- What will fact_v4(3) evaluate to?
- Is test_4 ok? (Yes, because res is static.)

```
int fact_v3(int N) {
    int res = 1;
    if (N == 0) return res;
    res = res * N;
    fact_v3(N-1);
    return res;
}
```

```
int fact_v4(int N) {
    static int res = 1;
    if (N == 0) return res;
    res = res * N;
    fact_v4(N-1);
    return res;
}
```

```
int*  test_4(int N) {
    static int res = 1;
    return &res;
}
```

```
// static variables have issues as well:
 int N = 5;
 printf("fact_v4(%d) = %d\n", N, fact_v4(N));
 N = 3;
 printf("fact_v4(%d) = %d\n", N, fact_v4(N));
```

```
fact_v4(5) = 120
fact_v4(3) = 720
```

Extra materials

Fun fact:
It is not known if this function always terminates.
(for any input)

```
int puzzle(int N)
{
  if (N == 1) return 1;
  if (N % 2 == 0)
       return puzzle(N/2);
  else return puzzle(3*N+1);
}
```

How is puzzle(3)
evaluated?

Factorial

Recursive Definition:

```
int factorial(int N)
{
   if (N <= 1) return 1;
   return N*factorial(N-1);
}
```

Same algorithm ↔ Version on the right, allows us to see `res` for different calls.

<u>**Practice execution of recursive fct calls**</u>

```
int fact (int N) {
   if (N <= 1) return 1;
   int res = N*fact (N-1);
   //T
   return res;
}
```

Practice:
For recursive calls generated from the original call `fact(6)`, what is `res` at time `T` when `N` is 3 and when `N` is 5?

My terminology:

- When talking about the algorithm or paper definition: base <u>case</u>, recursive case
- When talking about the implementation: base <u>step</u>, recursive step
- <u>I will probably end up mixing these terms.</u>

- The **recursive call** is the actual (self) function call. E.g. **`fact(N-1)`** above.

Passing Pointers in function calls in C

- Managing the memory yourself may be safer.

```
void fact_tail_helper(int N, int* res) {
    if (N == 0) return;
    (*res) = (*res) * N;
    fact_tail_helper (N-1, res);
}
```

```
//Referencing a local variable.
// OK if done in the correct 'direction'.
// Easier.
int fact_tail(int N) {
    int res = 1;
    fact_tail_helper (N, &res);
    return res;
}
```

```
//Managing the memory yourself.
// Make sure you do it right.
int fact_tail_2(int N) {
    int * res = (int*)malloc(sizeof(int));
    (*res) = 1;
    fact_tail_helper (N, res);
    int temp = (*res);
    free res;
    return temp;
}
```

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- Recursive algorithm
- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.
- It is based on the property that if T divides X and Y, then T also divides X mod Y.
- How is gcd(96, 36) evaluated?

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- How is gcd(96, 36) evaluated?
- gcd(96, 36) = gcd(36, 24) = gcd(24, 12) = gcd(12, 0) = 12.

Analyzing a Recursive Program – Factorial computes correct result

- Proof: by induction.
- Step 1: (the base case)
  - For N = 1, fact(1) returns 1, which is correct.
- Step 2: (using the inductive hypothesis)
  - Suppose that fact(N) returns the right result for N = K, where K is an integer >= 1. (fact(K) = K!)
  - Then, for N = K+1, fact(N) returns: N * fact(N-1) = (K+1) * fact(K) = (K+1) * K! = (K+1)! = N!
  - Thus, for N = K+1, fact(N) also returns the correct result.
- Thus, by induction, factorial(N) computes the correct result for all N.

Recursive Definition:

```
int fact(int N)
{
   if (N <= 1) return 1;
   return N*fact(N-1);
}
```

Where precisely was the inductive hypothesis used?

Analyzing a Recursive Program factorial computes correct result

- Proof: by induction.
- Step 1: (the base case)
  - For N = 1, fact (1) returns 1, which is correct.
- Step 2: (using the inductive hypothesis)
  - Suppose that fact(N) returns the right result for N = K, where K is an integer >= 1. (fact(K) = K!)
  - Then, for N = K+1, fact(N) returns:
    N * fact(N-1) = (K+1) * **fact(K) =** (K+1) * **K!** = (K+1)! = N!
  - Thus, for N = K+1, fact(N) also returns the correct result.
- Thus, by induction, fact(N) computes the correct result for all N.

Recursive Definition:

```
int fact(int N)
{
   if (N <= 1) return 1;
   return N*fact(N-1);
}
```

Where precisely was the inductive hypothesis used?

In substituting K! for fact(K).

Implementations for N!

Iterative :

```
int fact_iter(int N) {
   int i, r = 1;
  for (i = 2; i <= N; i++)   r *= i;
  return result;
}
```

Recursive, not tail-recursive (return answer):

```
int fact_ret(int N) {
    if (N <= 1) return 1;
    return N*fact_ret (N-1);
}
```

Tail-recursive (pass and return answer):

```
int fact_pr(int N, int res) {
    if (N <= 1) return res;
    res = res * N;
    return fact_pr(N-1, res);
}
  // Wrapper  function (sets parameters).
int fact_pr_wrapper(int N) {
    int res = 1;
    return  fact_pr(N, res); // res =?
}
// Note that it combines passing data with
returning data.
```

Tail-recursive (answer in updated argument):

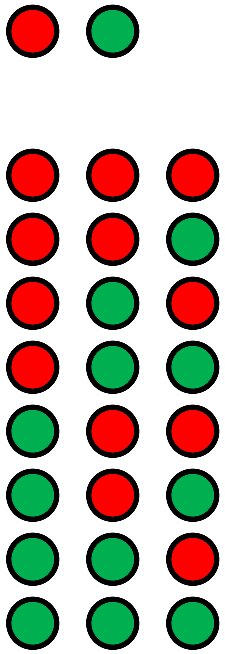
```
void fact_update(int N, int* res) {
    if (N <= 1) return;
    (*res) = (*res) * N;
    fact_update (N-1, res);
}
 // Wrapper  function (sets parameters).
int fact_update_wrapper (int N) {
    int res = 1;
    fact_update(N, &res);
   // note diff between N and  res when fct finishes
    return res;
}
```

What is the local work/computation? Is it done before or after the recursive call?

Difference (tail/non-tail recursive): Do the local computation before or after the recursive call.

N-tuples

- N positions, D types of items possible for each position, generate all N-tuples.
  - E.g.: lock combinations: 3 places, each place can have anyone of the 0-9 digits



N-tuples

- N positions, D types possible for each position, generate all N-tuples.
  - Lock combinations: 3 places, each place can have anyone of the 0-9 digits
- Method 1:
  - Iterate with x = 0 to D^N and convert x to an N-bit number in base D.
- Method 2: void perm(int* tuple_arr, int spots,…){ // if basecase: print tuple_arr
  - Recursive function that: will populate, update and print the tuple array.
  - Rough idea: For each position iterate over all digits:

```
for d = 0 -> D // assume D is not included: 0->(N-1)
    // set that position to have digit d.
```

  - Step 1 in understanding the problem and developing the solution: Assume N is not a variable, but it is always 3 (e.g. like a lock). Write the code.
    - As a preliminary test for your code, think about how many total permutations there are and what is the complexity of your code. They should match.
  - Step 2: Can you implement the above solution for cases where N is a variable (N is part of the input)?
  - Step 3: Can you use recursion?

Permutations

- Generate all permutations of N different elements.
- Write code.
- Write time complexity formula for the above code.