TND004: Data structures

- **Recursion** -- 1.1, 1.3, 10.2
  -- advised reading sec. [1.2.1-1.2.4]
  - Factorial
  - Binary search
  - Fibonnaci
  - How function calls are implemented by computers
    - Stack frames (activation records) – see TNG033/Fö2, slides 8-12
  - Tail recursion
- **Divide-and-conquer** -- sec. 10.2 (pags 485 and 486), *10.2.4*
  - Multiplying integers

1

Information

- Code in the lecture slides is simplified
  - e.g. drop `std::`, `const` from arguments, etc
  - focus is the algorithm (i.e. strategy) to solve the problem
- Feel free to take photo of stuff I hand-write on the board
  - I do not want to be in the photos
  - Just ask me to step aside

2

Recursion

- To solve a problem recursively
  - **Intuition**: *hey, to solve this problem, I'll just solve a smaller/simpler version of the same problem*
- Something that is defined in terms of itself -- math concept
  - Recursive functions -- factorial, fibonacci
  - Recursive structures -- trees

3

Recursion

Recurrence relation ⇔ recursive function

$$n! = \begin{cases} 1 & ,if\ n = 0 \\ n \times (n-1)! & ,if\ n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & ,if\ n = 0 \\ n \times (n-1) \times \cdots \times 1 & ,if\ n > 0 \end{cases}$$

Recursive function

```
long long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```


4

Recursive functions

```
void f( )
{
    …
    f( );
    …
}
```

Function **f** calls itself, directly

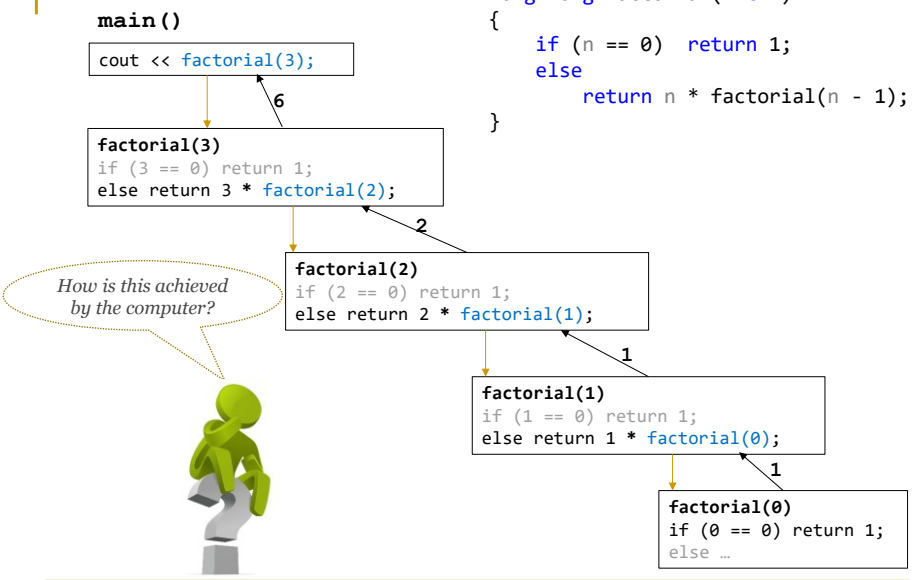
```
void f( )
{
    …
    g( );
    …
}
```

Function f calls itself, indirectly

```
void g( )
{
    …
    f( );
    …
}
```


5

```
long long factorial(int n)
{
    if (n == 0)  return 1;
    else
        return n * factorial(n - 1);
}
```

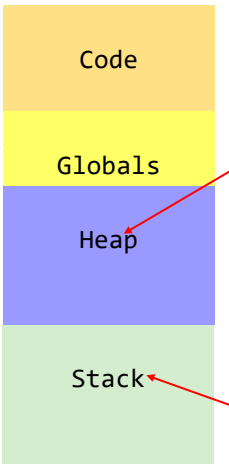


6

Which "*bookkeeping*" is done by the computer when a function is called?

-- review TNG033/lecture2, slides 8-12

Variables defined outside any function Persist in memory for the program's duration



Data structures which can **grow dynamically** are placed in the heap (`std::vector`)

See how does a vector work

Data structures **cannot grow** (size is set when they are created, e.g. `std::array`)

Managed by the programmer

- In C++, it's possible to reserve memory for variables while the program executes
- These variables exist until the programmer explicitly frees their memory
- Lifetime of these variables is not managed by scope rules

Managed automatically by the compiler

- Scope rules manage the lifetime of variables
- Local variables
- Functions' arguments
- Fast access

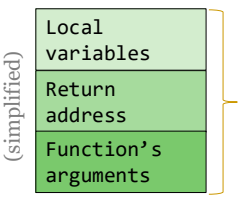
7

Calling a function

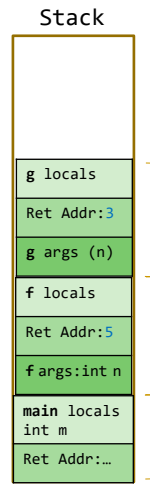
```
int f(int n) {
1.  …
2.  g(n);
3.  …
4.  return …;
}
int main() {
5.  int m = f(5);
6.    …
}
```

Function call: place a stack frame on top of the stack

Return: remove stack frame from the stack



Stack frame (activation record)



Stack frame for g

Stack frame for f

Stack frame for main

8

Recursion

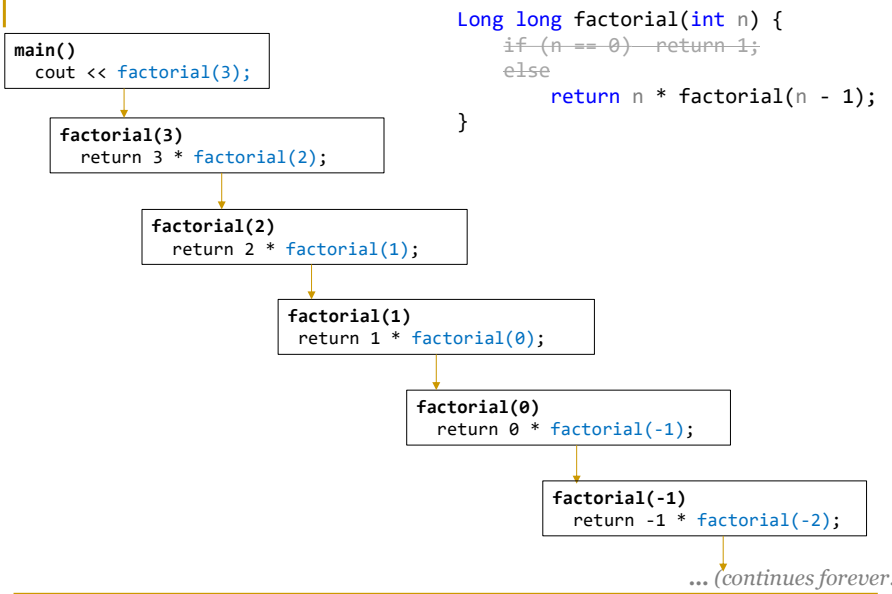
```
long long factorial(int n)
{
    if (n == 0)  return 1;
    else
        return n * factorial(n - 1);
}
```

Base case: solved without recursion

Recursive call

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop
- A different value is passed to the function each time it is called recursively
  - Make sure each recursive call progresses toward the base case
  - Otherwise, the program enters in an "*endless*" loop -- crashes!!

9




```
Long long factorial(int n) {
    if (n == 0)  return 1;
    else
        return n * factorial(n - 1);
}
```

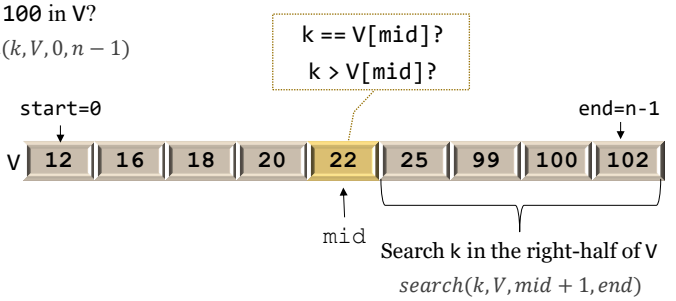

10

Recursion

Binary Search: **sorted** input sequence

Is k = 100 in V?

$search(k, V, 0, n-1)$



$$\textbf{search}(k,V,start,end) = \begin{cases} false & ,if\ end < start \\ true & ,if\ k = V[mid] \\ \textbf{search}(k,V,mid+1,end) & ,if\ k > V[mid] \\ \textbf{search}(k,V,start,mid-1) & ,if\ k < V[mid] \end{cases}$$

11

Binary search

$$\textbf{search}(k,V,start,end) = \begin{cases} false & ,if\ end < start \\ true & ,if\ k = V[mid] \\ \textbf{search}(k,V,mid+1,end) & ,if\ k > V[mid] \\ \textbf{search}(k,V,start,mid-1) & ,if\ k < V[mid] \end{cases}$$

```
bool search(int k, const vector& V, int start, int end) {
    if (end < start)  return false;

    int mid = start + (end-start) / 2;

    if (V[mid] == k) return true;

    else if (k > V[mid])
        return search(k, V, mid + 1, end);
    return search(k, V, start, mid - 1);
}
```

k not found!

k found!

Search right half

Search left half

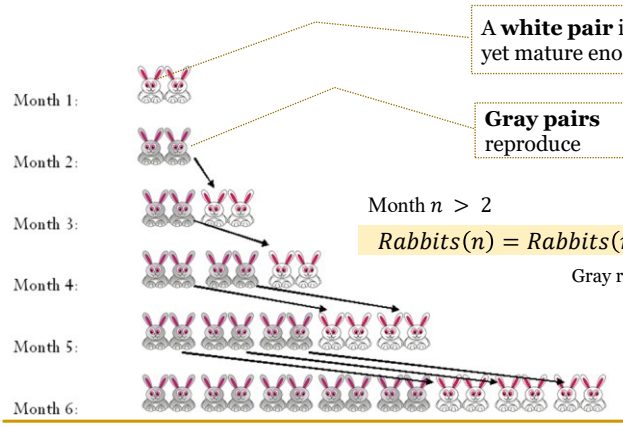
```
search(100, V, 0, size(V)-1); // search for 100 in V
```


12

The fibonnaci rabbits

Fibonacci rabbits come in pairs. Once a pair is two months old, it bears another pair and from then on bears one pair every month. Starting with a newborn pair at the beginning of a year, how many pairs of rabbits will there be at the end of the year?

Book *Liber Abaci* (1202) by Leonardo of Pisa



A **white pair** if rabbits is one that is not yet mature enough to produce another pair

Gray pairs reproduce

Month $n > 2$

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

Gray rabbits White rabbits

13

The fibonnaci rabbits

Recurrence relation

$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n \geq 2 \end{cases}$$

Try to come up with an equivalent non-recursive function

Recursive function

Base cases: solved without recursion

```
long long fibonacci(int n)
{
    if (n == 0 or n == 1)  return n;

    return fibonacci(n - 1) + fibonacci(n - 2);

}
```

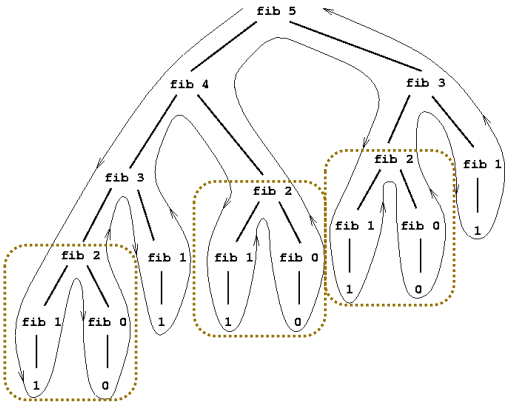
Recursive calls

14

```
long long fibonacci(int n)
{
    if (n == 0 || n == 1)  return n;

    return fibonacci(n - 1) + fibonacci(n - 2);

}
```



15

Tail recursion

```
void display(vector<int>& V, int i)
{
    if (i == size(V) {
        cout << "\n";
        return;
    }
    cout << V[i] << " ";
    display(V, ++i);
}
```

```
void main() {
    vector<int> A {-1, 4, …};
    display(A, 0);
}
```

Start index

Tail recursion

```
void display(vector<int>& V, int i)
{
    if (i < size(V)) {
        cout << V[i] << " ";
        display(V, ++i);
    }
    else
        cout << "\n";
}
```

Tail recursion

Tail recursion: function calls itself as last thing to be **executed** before returning

Tail recursion can be easily replaced by a `while`-loop

16

Tail recursion

```
long long factorial(int n)
{
    if (n == 0)  return 1;
    else
        return n * factorial(n - 1);
}
```

Not tail recursion

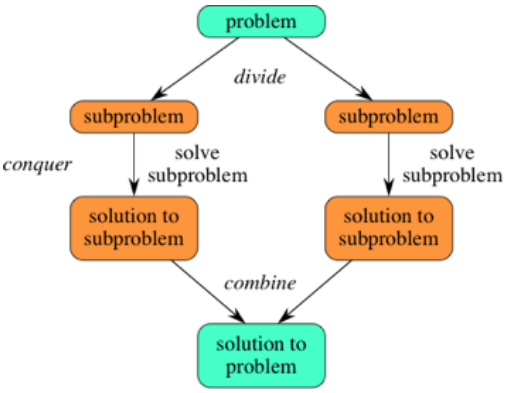
It's a bad programming practice to use recursion (e.g. tail recursion) that can be easily replaced by loops

- Use more memory space
- Can consume all memory in the computer's stack

What have we discussed so far

- Introduction to recursive functions
  - Factorial
  - Fibonnaci
  - Binary search
  - *other examples in the course*
- How function calls are implemented by computers
- Tail recursion

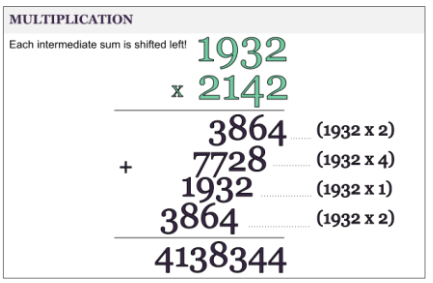
Divide-and-conquer



- The problems are divided in at least two subproblems
- The subproblems are disjoint
- **Recursion** is used to implement divide-and-conquer strategies

Multiplying integers

-- sec. 10.2 (pags 485 and 486), *10.2.4*



Algorithm 1: use nested loops

Algorithm 2: Use divide-and-conquer strategy

- Multiply two n-digits numbers, $x, y \geq 0$. Use only
  - Additions (+)
  - Multiplication by $10^k, k > 0$
  - Multiplication (*) of one-digit integers

Can be implemented efficiently

Multiplying integers

- **Algorithm 2**: Divide-and-conquer strategy
  - x =61.438.521 y = 94.736.407 n = 8 (number of digits)
  - $x \times y$ = 5.820.464.730.934.047
- Divide x and y in two halves
  - $x_L = 6143$ $x_R = 8521$ $x = x_L 10^4 + x_R$
  - $y_L = 9473$ $y_R = 6407$ $y = y_L 10^4 + y_R$
- Conquer
  - $x \times y = x_L y_L 10^n + (x_L y_R + x_R y_L) 10^{n/2} + x_R y_R$

Four multiplications where each number has half of the size of the original problem ($n/2$ digits)

Multiplying integers

Pseudocode for algorithm 2

```
// x and y are non-negative integers with at most n digits
// Assume n is even
Multiply(x, y, n) {
    if (n == 1) return x * y

    m = n / 2       // integer division
    p = 10^m

    xL = x / p      // integer division
    xR = x % p
    yL = y / p      // integer division
    yR = y % p

    a = Multiply(xL, yL, m)
    b = Multiply(xL, yR, m)
    c = Multiply(xR, yL, m)
    d = Multiply(xR, yR, m)

    return a * 10^n + (b + c) * 10^m + d      // x * y
}
```

Base case

Divide-and-conquer (recursion is used to solve "smaller" problems)

Combine

Multiplying integers -- in binary

Algorithm 2 works if numbers are represented in binary

If x and y are represented with n bits, split them into left and right halves:

$$x = x_L 2^{n/2} + x_R$$
$$y = y_L 2^{n/2} + y_R$$

Then

$$xy = x_L y_L 2^n + (x_L y_R + x_R y_L) 2^{n/2} + x_R y_R$$

Efficient operations in binary:

- $a + b$ -- can be implemented with bit operations XOR ($\oplus$) and AND ($\wedge$)
- $a \times 2^k$ -- shift bits of a k times to the left
- $a / 2^k$ -- shift bits of a k times to the right
- $a \,\%\, 2^k$ -- a ∧ (2^k – 1), 2^k – 1= 000…0111…1 (k ones)

Multiplying integers -- in binary

Pseudocode for algorithm 2 -- binary multiplication

```
// x and y are non-negative integers represented with at most n bits
// Assume n is even
Multiply(x, y, n) {
    if (n == 1) return x * y          // bit operation: x ∧ y

    m = n / 2       // integer division

    xL = x / 2^m    // m bit-shifts to the right
    xR = x % 2^m    // x ∧ (2^m – 1), 2^m – 1= 000…0111…1 (m ones)
    yL = y / 2^m    // m bit-shifts to the right
    yR = y % 2^m    // y ∧ (2^m – 1), 2^m – 1= 000…0111…1 (m ones)

    a = Multiply(xL, yL, m)
    b = Multiply(xL, yR, m)
    c = Multiply(xR, yL, m)
    d = Multiply(xR, yR, m)

    return a * 2^n + (b + c) * 2^m + d   // x * y

}
```

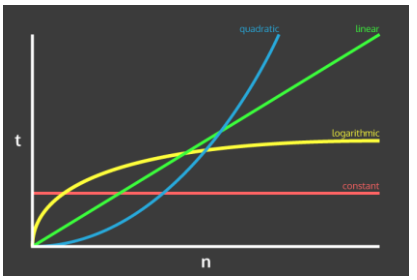
Multiplying integers

- Which algorithm is the best?
  - Algorithm 1?
  - Algorithm 2?

25

Algorithm analysis

- Just writing a working program is not good enough
  - Correctness -- e.g. software testing, formal verification
  - Termination
  - **Efficiency** -- use of *resources* as a function of input size
    - How to **estimate** the running **time** of an algorithm for **large inputs**?
    - How to estimate the memory **space** required for **large inputs**?
    - How to **compare** the running times/memory space of two algorithms?



t = time
n = input size (e.g. number of digits of x and y)

26

Next …

- **Lecture 2** -- read **sec. 2** of course book
  - Algorithm analysis
  - Big-O notation -- formal mathematical definitions
  - Examples
- **Lab 1** -- stable partition problem
  - Read the lab description
  - Do exercise 1: iterative algorithm -- before **HA** lab session
    - See how does a vector work
  - Understand the divide-and-conquer algorithm of exercise 2
  - Strict deadline to present the lab: **RE** on week 16
    - Otherwise, you cannot be awarded 3p for the labs