DATA STRUCTURES

LECTURE 2: RECURSION

MADE BY: KEVIN HARVEY

Definition

Recursion is a way of decomposing a huge task into smaller tasks.

Right To Examples!

⚙️ Example: 2^5 (2 to the power of 5)

Goal: Calculate $2^5 = 2 \times 2 \times 2 \times 2 \times 2$

Right To Examples!

🌀 ***By recursion:***

```
int twoRaisedTo0(){ return 1; }
int twoRaisedTo1(){ return 2 * twoRaisedTo0(); }
int twoRaisedTo2(){ return 2 * twoRaisedTo1(); }
int twoRaisedTo3(){ return 2 * twoRaisedTo2(); }
int twoRaisedTo4(){ return 2 * twoRaisedTo3(); }
int twoRaisedTo5(){ return 2 * twoRaisedTo4(); }

int main() {
    cout << "2 to the 5th is " << twoRaisedTo5() << "\n";
}
```

🥴 That's... a lot of functions.

Every one does the same thing, except `twoRaisedTo0()` is special (base case).

Right To Examples!

✅ ***Simplified version (recursive method):***

```
int twoRaisedTo(int n) {
    if (n == 0)      // base case
        return 1;
    else
        return 2 * twoRaisedTo(n - 1);
}

int main() {
    cout << "2 to the 5th is " << twoRaisedTo(5);
}
```

Clarification

💡 Base case vs. General case

Type	Meaning	Example
Base case	stops recursion	n = 0 or n = 1
General (recursive) case	defines problem using smaller version of itself	$n! = n \times (n-1)!$

🧩 Every recursive function **must** have:

1. At least one *base case* (to stop recursion).
2. At least one *recursive case* (to break the problem down).

🧠 How Recursion Works Internally

When function `A()` calls another function `B()`:

1. `A` pauses its execution.
2. Local variables of `A` are stored (stack memory).
3. The return address is saved.
4. Control transfers to `B()`.
5. When `B` finishes, all of `A`'s info is restored.
6. `A` resumes from where it left off.

> Each function call creates an activation record (stack frame) containing local variables, parameters, and return addresses.

🧩 Recursive call example

```c
int f(int x) {
    int y;
    if (x == 0)
        return 1;
    else {
        y = 2 * f(x - 1);
        return y + 1;
    }
}
```

Calling `f(3)` creates:

```
f(3) → f(2) → f(1) → f(0)
```

- `f(0)` returns 1
- Then unwind:
    - `f(1)` → y = 2 * 1 = 2 → return 3
    - `f(2)` → y = 2 * 3 = 6 → return 7
    - `f(3)` → y = 2 * 7 = 14 → return 15 ✅

✨ Example: Factorial

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

✨ Example: Factorial

C++ implementation:

```cpp
int fac(int n) {
    if (n <= 1)
        return 1;
    else
        return n * fac(n - 1);
}
```

Trace for `fac(3)`:

```
fac(3) = 3 * fac(2)
fac(2) = 2 * fac(1)
fac(1) = 1 (base)
=> 3 * 2 * 1 = 6 ✅
```

⚔️ Recursion vs. Iteration

🔶Feature	🔁 Iteration	🔁 Recursion
Uses	Loops (`for`, `while`)	Selection (`if`, `switch`)
Stops when	Loop condition fails	Base case satisfied
Control	Counter variable	Simplified smaller problem
Efficiency	Faster, less memory	Slower, more memory
Elegance	More verbose	Often shorter and clearer

⚠️ Be Careful!

You can cause:

- Infinite **loops** in iteration, or
- Infinite **recursion** if you forget the base case!

```
int fac(int n){
    return n * fac(n - 1);   // ❌ no base case
}
```

```
int fac(int n){
    if (n <= 1)
        return 1;
    else
        return n * fac(n + 1);   // ❌ wrong direction
}
```

🔍 Recursive Searching

🟩 Recursive Linear Search

```
int linearSearch(int a[], int n, int target){
    if(n <= 0) return -1;             // base case
    if(a[n-1] == target) return n-1; // check last element
    return linearSearch(a, n-1, target);
}
```

Driver:

```
int main(){
    int arr[]={5,2,8,4,9};
    int result = linearSearch(arr,5,8);
}
```

☝️ This array goes **backwards not forwards**

🔍 Recursive Searching

🧩 Trace Example:

Array `{17, 6, 9, 21}`, target = 6

- Check last (21), nope
- Check 9, nope
- Check 6 ✅ found at index 1

> ☝️ This array goes **backwards not forwards**

🔍 Recursive Searching

2️⃣ Binary Search

```
int binarySearch(int a[], int first, int last, int target){
    if(first > last) return -1;
    int mid = (first + last) / 2;
    if(a[mid] == target) return mid;
    else if(target < a[mid])
        return binarySearch(a, first, mid - 1, target);
    else
        return binarySearch(a, mid + 1, last, target);
}
```

📘 **Example:**

Find the number **42** in the sorted list below 👇

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

🔍 Recursive Searching

🧼 Recursive Bubble Sort

```
void bubbleSortRecursive(int arr[], int n) {
    if (n == 1) return;
    for (int i=0; i<n-1; i++){
        if(arr[i] > arr[i+1])
            swap(arr[i], arr[i+1]);
    }
    bubbleSortRecursive(arr, n-1);
}
```

Driver:

```
int arr[]={64,34,25,12,22,11,90};
bubbleSortRecursive(arr,7);
```

🔍 Recursive Searching

⚡ Other Recursive Examples

🔷 Exponential Function

```
int exp(int num, int power){
    if(power == 0)
        return 1;
    return num * exp(num, power - 1);
}
```

🧩 📊 Comparison Table

🔢 Algorithm	🔍 Type	💡 Works On	⚙️ Method	🧠 Best Case	💀 Worst Case	✅ Advantages	⚠️ Disadvantages
Linear Search	Searching	🔶 Sorted or Unsorted	Compare each element sequentially	The Target Value is the first index of the array.	The Target Value is the Last Index of The Array or not found	✅ Easy to implement ✅ Works on any list	❌ Super slow for big data ❌ Linear performance
Binary Search	Searching	🔷 Sorted lists only	Divide list into halves repeatedly	The Target Value is in the middle of the array	The array is unsorted, therefore the array will misbehave.	✅ Very fast ✅ Fewer comparisons	❌ Requires sorted data ❌ Not for linked lists
Insertion Sort	Sorting	🔷 Any (works best on small or nearly sorted data)	Inserts each element into its correct place	The Array is already sorted	The array is unsorted	✅ Simple logic ✅ Efficient on small datasets	❌ Slow for large lists ❌ Quadratic time complexity
Bubble Sort	Sorting	🔷 Any	Repeatedly swaps adjacent elements	The array is sorted	The array is reversed.	✅ Easy to code ✅ Detects already sorted data	❌ Painfully slow ❌ Too many swaps

Thank you!