Recursion

Recursion is defining something in terms of itself.
- We define many data structures recursively
  - A linked list node contains a pointer to a node
  - A binary tree node contains two pointers to nodes
- Many functions can be defined recursively:
  - Factorial: \( n! = n(n-1)! \)
  - Differentiation (chain rule): \( \frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx} \)
  - The binomial coefficient: \( \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \)
  - Euclid’s algorithm for GCD is recursive!

Recursive Functions in C++

- Most modern programming languages allow recursion in functions;
- In C++, you simply call a function from within itself, e.g.:
  ```cpp
  unsigned int factorial(unsigned int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
  }
  ```

The Base Case

Note the first line of the `factorial` function:
```cpp
unsigned int factorial(unsigned int n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}
```
What would happen without that line?
When the input \( n \) is 0 we call it the **base case**.
The test for the base case **must come before** the recursive call!

Example: Palindrome

- A palindrome is a recursive object; it is:
  - Empty, or
  - A single character, or
  - A palindrome between two of the same character
- Here’s a recursive test function:
  ```cpp
  bool is_palindrome(const string &s, int start, int end) {    
    if (end <= start) return true;
    return (s[start] == s[end]) && is_palindrome(s, start + 1, end - 1);
  }
  ```
  ```cpp
  bool is_palindrome(const string &s) {    
    return is_palindrome(s, 0, s.length() - 1);
  }
  ```
Example: Binomial Coefficient

```c
unsigned int nchoosek(unsigned int n, unsigned int k) {
    assert(n >= k);
    if (k == 0 || k == n) return 1;
    return nchoosek(n-1,k) + nchoosek(n-1,k-1);
}
```

Note - more than one base case!
Note - two recursive calls!

Common Mistakes

- No base case:
  ```c
  void infinite(int n) {
    cout << n << endl;
    infinite(n-1);
  }
  ```

- Recursion step doesn’t reduce problem:
  ```c
  void infinite2(int n) {
    if (n < 0) return;
    cout << n << endl;
    infinite2(n);
  }
  ```

Recursion vs. Iteration

Recursion is often the simplest approach.

However, recursion can usually be replaced by iteration plus some storage for intermediate results.

```c
unsigned int factorial(unsigned int n) {
    unsigned int ans = 1;
    for (int j = n; j > 1; j--) ans = ans * j;
    return ans;
}
```

Recursive Decomposition

- Recursion works well when:
  - Problem can be rewritten as smaller sub-problems
  - Sub-problems have the same structure as original
  - Solving all sub-problems solves original problem
  - Examples (from previous slides)
    - Palindrome rewritten as: “check outer two characters, then test for smaller palindrome”
    - Binomial coefficient rewritten as sum of “easier” binomial coefficient problems

Recursion as Induction

The basic form of recursion follows that of induction:

- Recursive base case(s) == inductive base case(s)
  - If we apply our function to problem of size 1, then we get the correct answer
  - E.g., if a string is size 1 or 0, then it is a palindrome
- Recursive step == inductive step
  - If we are correct on problem of size n, then we are correct on a problem of size n + 1
  - Palindromes are a bit tricky here, because we actually prove 2 cases, one for odd numbers and one for evens:
    - If our program works for strings of n letters, then prove it works for strings of n + 2 letters
Example: Permutations

- Problem: find all permutations of an ordered set
  - E.g., what are all permutations of \((a, b, c)\)?
    - Answer: \((a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)\)
  - What about \((a,b,c,d,e,f,g,h,...)\)?
    - Ugh. Let the computer do it.
    - OK... how?

You Try: Permutations

- What is the recursive substructure?
  - E.g., what is a smaller problem then \((a,b,c)\)?

- Given the above, what is the base case?

Maze Solving

Consider solving a maze:

- Assume potential loops, so right-hand rule fails
- Instead, have string and a marker
  - Mark where you’ve been, so you don’t loop
  - Unroll string behind you so you can back up
  - Pick a passage, follow as far as you can until dead-ending or repeating yourself
  - Back-up to the last branching and try one you haven’t tried (or back up further if no choices left)

Backtracking

- The maze solving algorithm above is an example of backtracking
- Essentially, try every possibility in a branching problem, avoiding repeats
- This sort of has the recursive sub-structure:
  - The problem is only made smaller by a little bit
  - We have to remember choices (or do we?)

Maze Solving Pseudocode

\[
\text{solve}_2d\_\text{maze}(\text{maze}, x, y):
\]

\[
\text{if at exit, yay!}
\]

\[
\text{else:}
\]

\[
\begin{align*}
\text{mark maze[x][y] as visited} \\
\text{if can go right:} \\
\text{solve}_2d\_\text{maze}(\text{maze}, x+1, y) \\
\text{if can go down:} \\
\text{solve}_2d\_\text{maze}(\text{maze}, x, y+1) \\
\end{align*}
\]

etc.
Winning!

**MINIMAX**

---

**Backtracking for Games**

- For 2-player perfect information games
- Like trying every possibility, but:
  - Assume each player is trying to win 😊
  - Each player has a different goal, so have to switch
- Classic algorithm is called *minimax*

---

**Example: Nim**

- The game:
  - Put \( n \) tokens on the table
  - Each player gets to take 1, 2, or 3 tokens each turn
  - Player who takes the last token loses
- Work backwards from base case:
  - If 1 coin left for other player, you win 😊
  - Thus, if 2-4 coins left for you, you can force win 😊
  - However, if 5 coins left for you, you lose, because any move you make leaves a good move for opponent...

---

**Solving Nim Recursively**

```python
find_good_move(ncoins):
    for i = 1 to min(3, ncoins):
        if ncoins - i == 1:    // base case: WIN 😊
            return i
        if find_good_move(ncoins - i) == NO_GOOD:
            return i
    return NO_GOOD    // base case: LOSE 😞
```

---

**Up Next**

- Friday, Nov. 3
  - Lab 9, continued
  - Project 4 assigned
- Monday, Nov. 6
  - Analysis of Algorithms 1
  - Read 16.1, 16.4, and 16.7
  - Lab 9 due
- Wednesday, Nov. 8
  - Analysis of Algorithms 2