ECE15: Introduction to Computer Programming Using the C Language

Lecture 8: Recursive Functions
Lecture Outline

❖ Recursion and recursive functions
  ▸ Introduction to recursive functions
  ▸ Solving simple problems using recursion

❖ Example: Reverse Printing
❖ Example: Binary Search
What are Recursive Functions?

Functions that call themselves!

When is recursion useful?

When a problem with parameter \( n \) can be solved using a solution to a problem with a smaller parameter, e.g. \( n - 1 \), or \( n/2 \)

\[
gcd(m, n) = gcd(n, m \% n); \]

\[
factorial(n) = n \times factorial(n - 1); \]

Stopping condition or base case

To stop, a recursive function must have one or more base cases: inputs for which it returns immediately, without recurring
A Simple Recursion

```c
void printx()
{
    printx();
    printf("x");
}

int main()
{
    printx();
    return 0;
}
```

How many x’s will get printed?

Infinite loop! x will never get printed. Program will crash once repeated printx() calls overflow the stack.

[first call] void printx()
{
    printx();
    printf("x");
}

[second call] void printx()
{
    printx();
    printf("x");
}

[third call] void printx()
{
    printx();
    printf("x");
}

...
`void printx()`
{
    printf("x");
    printx();
}

`int main()`
{
    printx();
    return 0;
}

---

Still an infinite loop!
But now, `x` gets printed repeatedly until the program crashes when `printx()` calls overflow the stack.

---

What If We Change the Order?

---

How many x’s will get printed?

---

Lecture 10
ECE15: Introduction to Computer Programming Using the C Language
Adding a Stopping Condition

Lecture 10
ECE15: Introduction to Computer Programming Using the C Language

```c
void printi(int i) {
    if (i < 1) return;
    printf("%d\n",i);
    printi(i-1);
}

int main() {
    printi(4);
    return 0;
}
```

printi1.c
void printi(int i) {
    if (i < 1) return;
    printi(i-1);
    printf("%d\n",i);
}

void printi(int i) {
    if (i < 1) return;
    printi(i-1);
    printf("%d\n",i);
}

int main() {
    printi(4);
    return 0;
}

What If We Change the Order?

int main() {
    printi2(4);
    return 0;
}
Recursion and recursive functions

- Introduction to recursive functions
- Solving simple problems using recursion

Example: Reverse Printing

Example: Binary Search
Factorial: Recursive Computation

**Function definition:** \( \text{factorial}(n) = 1 \cdot 2 \cdot 3 \cdots n \)

**Recursive formula:** \( \text{factorial}(n) = n \cdot \text{factorial}(n-1) \)

**Stopping condition:** \( \text{factorial}(1) = 1 \)

```c
#include <stdio.h>

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

int main() {
    int n;
    printf("n: ");
    scanf("%d", &n);
    printf("%d! = %d\n", n, factorial(n));
    return 0;
}
```

factorial.c
Function definition:  \( \text{power}(x, n) = x^n \)

Recursive formula:  \( \text{power}(x, n) = x \cdot \text{power}(x, n-1) \)

Stopping condition:  \( \text{power}(x, 0) = 1 \)

```c
#include <stdio.h>

double power(double x, int n);

int main()
{
    double x;
    int n;
    scanf("%lf^%d", &x, &n);
    printf("%f^%d = %f\n", x, n, power(x, n));
    return 0;
}

double power(double x, int n)
{
    if (n == 0)
        return 1;
    return x * power(x, n - 1);
}
```

power.c
Fibonacci Rabbits

- Leonardo Fibonacci, ~1170-1250
- Introduced decimal system to Europe
- Asked following question
- Pair of newborn rabbits put in a field
- After one month, they mate
- After another month, give birth to new pair (male & female)
- Repeat every month, thereafter
- Never die
- How many rabbit pairs after one year

11
Fibonacci Numbers in Nature

Lecture 7

ECE15: Introduction to Computer Programming Using the C Language
Fibonacci Program (recursive)

- **Represents:** Number of rabbits alive after n months
- **Defined:**
  - F(0)=F(1)=1
  - F(n)=F(n-1)+F(n-2) for n ≥ 2

```c
#include <stdio.h>

int fib(int n) {
    return n<=1 ? 1 : fib(n-1)+fib(n-2);
}

int main() {
    int n;
    printf("n: ");
    scanf("%d", &n);
    printf("Fibonacci(%d)=%d\n", n, fib(n));
    return 0;
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>F(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
</tr>
</tbody>
</table>

- Inefficient
Efficient Fibonacci (Non Recursive)

```c
#include <stdio.h>

int main() {
    int n, i;
    printf("n: ");
    scanf("%d", &n);
    int fib[n+1];
    fib[0]=fib[1]=1;
    for (i=2; i<=n; i++)
        fib[i]=fib[i-1]+fib[i-2];
    printf("Fibonacci(%d)=%d\n", n, fib[n]);
}
```

```c
#include <stdio.h>

int main() {
    int n, i, fib[2]={1,1};
    printf("n: ");
    scanf("%d", &n);
    for (i=2; i<=n; i++)
        fib[i]=fib[(i-1)%2]+fib[(i-2)%2];
    printf("Fibonacci(%d)=%d\n", n, fib[n]);
}
```

```c
#include <stdio.h>

int main() {
    int n, i, fib[2]={1,1};
    printf("n: ");
    scanf("%d", &n);
    for (i=2; i<=n; i++)
        fib[i]=fib[0]+fib[1];
    printf("Fibonacci(%d)=%d\n", n, fib[n]);
}
```

```c
int j=0;
for (i=2; i<=n; i++, j=1-j)
    fib[j]=fib[0]+fib[1];
```

Fibonacci2.c

Fibonacci3.c

Fibonacci4.c
GCD: Recursive Computation

Function definition: \( \text{gcd}(m, n) = \max_d \{ d \mid m \ \&\& \ d \mid n \} \)

Recursive formula: \( \text{gcd}(m, n) = \text{gcd}(n, m \mod n) \)

Stopping condition: \( \text{gcd}(m, 0) = 1 \)

```c
#include <stdio.h>

int gcd(int m, int n);

int main()
{
    int m, n;
    scanf("%d%d", &m, &n);
    printf("gcd(%d, %d) = %d\n", m, n, gcd(m, n));
    return 0;
}

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

gcd\((m, n)\)
Recursion and recursive functions
- Introduction to recursive functions
- Solving simple problems using recursion

Example: Reverse Printing

Example: Binary Search
**Task:** Read characters until '\n', then print the them in reverse.

**Solution?** Store the characters in an array, then print them out from the end:

```c
for (i=n-1; i>=0; i--) printf("%c", a[i])
```

Need a bound on the number of characters to store in the array `a[]`. Dynamic memory allocation does not help.

**Recursive Solution:**

```c
reverse("abcdef\n") ➜ [reverse("bcdef\n"),"a"]
```

- Read the first character and store it somewhere
- Reverse print the remaining characters (via a recursive call to self)
- Now print the first character
What is the stopping condition?

reverse("\n") → return

Use getchar and putchar to read and print one character at a time

```c
#include <stdio.h>

void print_reverse(void);

int main()
{
    print_reverse();
    printf("\n");
    return 0;
}

void print_reverse(void)
{
    char c;
    c = getchar();
    if (c == '\n') return;
    print_reverse();
    putchar(c);
}
```
Here is how the function call stack evolves when `print_reverse` is called with the user input: `go! \n`
Lecture Outline

❖ Recursion and recursive functions
  - Introduction to recursive functions
  - Solving simple problems using recursion

❖ Example: Reverse Printing

❖ Example: Binary Search
**Binary Search**

**Task:** Given an integer array \( a[\] \) of size \( n \), sorted in increasing order

\[
a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_{n-2} \leq a_{n-1}
\]

and a value \( x \), determine whether \( x \) appears in the array, and if so, where

**Solution?** Compare \( x \) to every element in the array:

```c
for (i = 0; i < n; i++) if (a[i] == x) return i;
```

Inefficient: requires \( n \) comparisons and does not use the fact that the array is sorted.

**Binary search:** only \( \log_2(n) \) comparisons!

- Compare \( x \) with the middle element in the array \( a_{n/2} \)
- If \( x = a_{n/2} \), we have found \( x \)
- If \( x < a_{n/2} \), search for \( x \) in the first half of the array
- If \( x > a_{n/2} \), search for \( x \) in the second half of the array
Recursively search array \( a[] \) for value \( x \).

If \( x \) is **not** found, return \(-1\)

If \( x \) found, return index \( i \) such that \( a[i] \) contains \( x \)

```c
int search(int a[], int imin, int imax, int x) {
    if (imin > imax) return -1; // array is empty
    int m = (imin+imax)/2; // midpoint index
    if (a[m] == x)
        return m;
    if (a[m] > x)
        return search(a, imin, m-1, x);
    return search(a, m+1, imax, x);
}
```

Write iteratively without recursion
extend N to 10M
Explain more what happens when size is 3, 2, 1
Recursion

Elegant solution to some problems.
In some cases, recursion is the only solution.

Advantages

❖ Handle potentially infinitely many cases using few simple functions
❖ Easy to write and understand: all "bookkeeping" done automatically

Disadvantages

❖ Usually takes more time and memory than iterative solutions
❖ If too deep may overflow stack
❖ May be difficult to debug