C מבוא לشفט

Tutorial 13: recursion
Last week

• sorting
This Week

• Recursion
• Loops using recursion
• Searching an array using recursion
• examples
Recursion
Definition

• **recursive definition** (or inductive definition) is used to define an object in terms of itself.
• A recursive definition of a function defines values of the functions for some inputs in terms of the values of the same function for other inputs.
• Recursive functions is a functions that **calls itself**.
An Example From math

• An example for a well known recursive function:
  • The Factorial function:
    – 0! = 1
    – (n+1)! = (n+1) * n!
Recursion in C

- Implementation in C for power of 2 recursive function

```c
int recursive(int i)
{
    int x;
    if (i == 0)
        return 1;
    x = recursive(i-1);
    x = x * 2;
    return x;
}
```

- Stopping condition. Must be checked before recursive call.
- Otherwise, we'll never reach it, and infinite recursion will occur.
- Return value
- Recursive call
What happens in the function call stack?

Every function call has its own copy of private parameters

```c
int recursive(int i)
{
    int x;
    if (i == 0)
        return 1;
    x = recursive(i-1);
    x = x * 2;
    return x;
}
```

Calling from main:
```
recursive(4);
```

```
main
recursive(0) i←0
recursive(1) x←2
recursive(2) x←4
recursive(3) x←8
recursive(4) x←16
```

int recursive(int i)
{
    int x;
    if (i == 0)
        return 1;
    x = recursive(i-1);
    x = x * 2;
    return x;
}
```
What are the required stopping conditions?

\[ a_n = a_{n-1} + 1 \]

\[ a_n = a_{n-2} + 3 \cdot a_{n-1} \]

\[ a_n = a_{n-2} + 1 \]

\[ b_n = \sum_{i=1}^{k} b_{n-i} \]
Motivation

• Every recursion can be written using a loop (and vice versa).
• So why do we need recursion?
• Some tasks are much easier to solve using recursive functions.
• Lets write an algorithm which prints the names of all the files inside a given directory, and inside it’s inner directories, and then their inner directories ...
• Lets assume we have the following functions to help us:
  – get_files(directory)- find all the files inside the directory
  – print_file_names(files)- prints files names
  – get_sub_directories(directory)- returns all the subdirectories of given directory
Print files names using a loop

pseudo code

• First, lets try using a loop:

  print_files(directory):
  1. files = get_files(directory)
  2. print_file_names(files)
  3. sub_directories = get_sub_directories(directory)
  4. for each sub_directory in sub_directories
     4.1 files = get_files(sub_directory)
     4.2 print_file_names(files)
     4.3 sub_sub_directories = get_sub_directories(sub_directory)
     4.4 for each sub_sub_directory in sub_sub_directories
        4.4.1 files = get_files(sub_sub_directory)
        ...

  It doesn’t end...
Print files names using a Recursion

• So much simpler using recursion:
  
  \textbf{print\_files}(\textit{directory}):
  
  1. \textit{files} = \textbf{get\_files}(\textit{directory})
  2. \textbf{print\_file\_names}(\textit{files})
  3. \textit{sub\_directories} = \textbf{get\_sub\_directories}(\textit{directory})
  4. for each \textit{sub\_directory} in \textit{sub\_directories}
     4.1 \textbf{print\_files}(\textit{sub\_directory})
Loops using Recursion
Loops using Recursion

• Every loop can be expressed using recursion.
• Instead of writing a loop containing some action, we could write the action in a function
• And the function will call itself until some condition is met.
Loops using Recursion

- Those two examples are equal:

```java
while (cond) {
    statement;
}

void while_func() {
    if (!cond) {
        return;
    }
    statement;
    while_func();
}
```

- Also, every recursive call as it’s own private parameters, which might hurt the space complexity.
Loops using Recursion

Simple example:

```c
int i = 0;
while (i < 3) {
    printf("%d\n", i);
    i++;
}
```

```c
void count(int i) {
    if (i >= 3) return;
    printf("%d\n", i);
    i++;
    count(i);
}
```

```c
count(0); :how to use
```
Riddle

• What will be printed?

```c
#include <stdio.h>

void foo(int i)
{
    if (i < 0) {
        return;
    }
    printf("%d\n", i);
    foo(i - 1);
}

void main(void) { 
    foo(10);
}
```
Another Ridlle

```c
#include <stdio.h>

void bar(int i) {
    if (i < 0) {
        return;
    }
    bar(i - 1);
    printf("%d\n", i);
}

void main(void) {
    bar(10);
}
```

Same program, small change

Switched the lines order

What will be printed now? Can you explain it?
Exercise

- Write a function which receives a string, and replaces every space in it with a star.

```c
void replace(char *str)
{
    int i = 0;
    while (str[i] != 0) {
        if (str[i] == ' ')
        {
            str[i] = '*';
        }
        i++;
    }
}
```
Now, lets do it recursively

- Do the same, without a loop:

```c
void replace(char *str, int i)
{
    if (str[i] == 0) {
        return;
    }
    if (str[i] == ' ') {
        str[i] = '*';
    }
    replace(str, i+1);
}
```

- What is the first i we should deliver?
Little bit harder

- could you do it with a single parameter? (no i)

```c
void replace(char *str)
{
    if (str[0] == 0) {
        return;
    }
    if (str[0] == ' ') {
        str[0] = '*';
    }
    replace(str + 1);
    printf("%c",str[0]);
}
```

Replace("h ell o");
What is the difference?

- Are they doing the same?
- What happens differently?

```c
void replace(char *str) {
    if (str[0] == 0) {
        return;
    }
    if (str[0] == ' ') {
        str[0] = '*';
    }
    replace(str + 1);
}
```

```c
void replace(char *str) {
    if (str[0] == 0) {
        return;
    }
    replace(str + 1);
    if (str[0] == ' ') {
        str[0] = '*';
    }
}
```
Searching arrays
General approach to solving recursion function

• When Given a problem:
  1. Find a way to make the problem “smaller”
  2. Assume the function knows how to solve the smaller problem
  3. Find a way to solve the big problem using the small problem – this is the recursive step
  4. Find what is the small problem for which we know the solution – this is the stopping condition.

This is merely a theoretical step, but it will help us solve the problem.
Recursive Search in an array

**The problem:** find the smallest cell in an array of size n.

1. The natural way to reduce the size of the problem is to reduce the size of the array.
2. We’ll assume we know how to find the Minimum in an array of size n-1.
3. **Step:** we’ll find the minimum cell in an array without the first cell.

4. Stopping condition: if the array is of size 1, we can return the only cell as the minimum.
int min(int a[], int size) {
    int minRest;

    if (size == 1) {
        return a[0];
    }

    minRest = min(a+1, size-1);
    if (a[0] < minRest) {
        return a[0];
    }
    else {
        return minRest;
    }
}
Re-cap

• It’s easy to change the last program so it will return the max value instead of the min value.

• Can you write a recursive function which will find both Min and Max in a single iteration over the array?

• Remainder: functions can return more than one value using pointers.
Algorithm for finding Maximum and Minimum Recursively

**Goal:** find both Maximum and Minimum numbers in an array of size n.

1. Like before, we’ll reduce the problem size by reducing the size of the array
2. We’ll assume that we already know how to find the Maximum and Minimum numbers in an array of size n-1.

3. **Step:** find the Maximum and Minimum numbers in an array without the first cell
   
   Example ($n = 5$):
   
   
   \[
   \begin{array}{cccc}
   1 & 5 & 9 & 2 & 4 \\
   \end{array}
   \]
   
   $min=2, max=9$

   $max=\text{Max(first cell in the array, max in array without the first cell)}$
   
   $min=\text{Min(first cell in the array, min in array without the first cell)}$

   \[
   \begin{array}{cccc}
   1 & 4 & 9 & 2 & 4 \\
   \end{array}
   \]

   $min=\text{MIN(1,2)}=1$
   
   $max=\text{MAX(1,9)}=9$

4. Stopping condition:
   
   for array of size 1: the only cell is both maximum and minimum

   \[
   \begin{array}{c}
   \end{array}
   \]

   $min=4, max=4$
Solution

```c
void min_and_max(int a[], int size, int *min, int *max) {
    if (size == 1) {
        *min = a[0];
        *max = a[0];
        return;
    }

    min_and_max(a+1, size-1, min, max);
    if (a[0] < *min) {
        *min = a[0];
    }
    if (a[0] > *max) {
        *max = a[0];
    }
}
```

Pay attention: min and max are already int pointers. This is why we don’t add &
Finding index of a value in an array

- Lets say we want to search the number “7” in an unsorted array.
- If the number doesn’t exits, return -1.
- If it does exist- return the index of the first cell in the array containing the wanted number.
- How will we do it recursively?
Algorithm for recursive search

**Goal:** find the index of \( x \) in an array of size \( n \).

1. The natural way to reduce the size of the problem is to reduce the size of the array.
2. We'll assume we know how to find the index of \( x \) in an array of size \( n-1 \).
3. If the first index is \( x \) then the returned index should be 0.
4. otherwise, we should distinguish between two possibilities:
   - We haven't found \( x \) in the array without the first cell. return -1
   - We have found it- in the index \( i \). return \( i+1 \)

5. Stopping condition: for an array of size 0: return -1
Solution

```c
int find(int a[], int size, int x) {
    int i;
    if (size == 0) {
        return -1; /* The array doesn’t contain x */
    }
    if (a[0] == x) {
        return 0; /* x is the first element in a[] */
    }
    i = find(a+1, size-1, x);
    if (i == -1) {
        return -1;
    } else {
        return i+1;
    }
}
```

Because when we are inside a function we think that the first cell in the array given to us is really the first one. But this isn’t the case.

Why do we add 1 to the value returned from the recursive call?
Finding a character inside a string

With a slight change, we can use this algorithm to find a character inside string:

```c
int find_in_string(char* str, char x)
{
    int i;
    if (*str == '\0') {
        return -1; /* The string doesn’t contain x */
    }
    if (*str == x) {
        return 0;
    }
    i = find(str+1, x);
    if (i == -1) {
        return -1;
    }
    else {
        return i+1;
    }
}
```

We don’t we need a “size” variable in this function.