מבוע לشفת C

Tutorial 13: recursion
Definition

• **Recursive definition** (or inductive definition) is used to define an object in terms of itself.

• A recursive function is a function which calls itself.
An Example From math

• An example for a well known recursive function:

• The Factorial function:
  – \( 0! = 1 \)
  – \( (n+1)! = (n+1) \times n! \)
Solving problems using 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 the stopping condition – think of the smallest problem that is needed for the computer to finish its calculation.

This is merely a theoretical step, but it will help us solving the problem.
Factorial in C

• C implementation of factorial \((n!)\)

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

- **Return value**
- **Recursive call**
- **Stopping condition.** Must be checked before recursive call
- **Otherwise, we’ll never reach it, and infinite recursion will occur**
Another Example

- C implementation for a power of 2 recursive function

```c
int power(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
What happens in the function call stack?

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

Every function call has its own copy of private parameters.
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.
Example: fibonacci

- Fib(0) = Fib(1) = 1
- Fib(n) = Fib(n-1) + Fib(n-2)
- 1, 1, 2, 3, 5, 8, 13, 21...

```c
int fibonacci(int n)
{
    if (n==0 || n == 1)
        return 1;
    int res1 = fibonacci(n-1);
    int res2 = fibonacci(n-2);
    return res1 + res2;
}
```
#include <stdio.h>

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

void main(void) {
    foo(10);
}
Another Riddle

#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.
• easy:

```c
void replace(char *str)
{
    int i = 0;
    while (str[i] != '\0') {
        if (str[i] == ' ') {
            str[i] = '*';
        }
        i++;
    }
}
```
Now, let's 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);
}
```
Searching Arrays
Recursive search in an array

The problem: find the smallest number 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 a given array of size n.

1. Like before, we'll reduce the problem size by reducing the size of the array.
2. We will assume that we already know how to find the maximum and minimum numbers in an array of size n-1.
3. **Step:** find the max and min numbers in an array without the first cell.
   
   Example (n = 5)
   - max = max(first cell in the array, max in the rest of the array)
   - min = min(first cell in the array, min in the rest of the array)

4. **Stopping condition:** for an array of size 1, return the only cell as max and min.
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 the index of a value in an array

• Lets say we want to search for 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 appearance of the number in the array.
• 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 that we know how to find the index of x in an array of size n-1.
3. If the number in 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
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;
    }
}
Finding a character inside a string

With a slight change, we can use this algorithm to find a character inside a 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.
Checking if a string is a palindrome

We want to check if a string is a palindrome – e.g. aabbaa or aba

```c
int is_palindrome(char* s, int len)
{
    if (len == 0 || len == 1)
        return 1;

    char left = *s;
    char right = *(s+len-1);

    if (left != right)
        return 0;

    return is_palindrome(s+1, len - 2);
}
```

This is good, but whoever uses the function has to know the length.
Checking if a string is a palindrome

```c
int is_palindrome(char* s, int len)
{
    if (len == 0 || len == 1)
        return 1;

    char left = *s;
    char right = *(s+len-1);

    if (left != right)
        return 0;

    return is_palindrome(s+1, len - 2);
}

int is_palindrome_wrapper(char* s)
{
    int len = strlen(s);
    return is_palindrome(s, len);
}
```

We can add a wrapper function which initializes the length parameter.