UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 6

Yichao (Joey)

May. 8, 2020

Credit: Carey Nachenberg, Junheng Hao

Announcements

UCLA Samueli Computer Science

- Midterm 2 is scheduled May 19.
- Project 3 is due on Wednesday, May 20

Outline Today

UCLA Samueli Computer Science

- Recursion

Recursion

Basics

UCLA Samueli Computer Science

- Function-writing technique where the functions refers to itself.
- Let's talk about the factorial example again!
  - Similar to mathematical induction → Prove `k=1` is valid and prove `k=n` is valid when `k=n-1` is valid.
  - Base cases are important and need to be carefully considered.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}
```

Without explicit loops!

Recursion

Basics

UCLA Samueli Computer Science

- Remember that recursive functions are just functions that call themselves.
- You first call yourself recursively **on a slightly smaller version of the argument**, before doing anything else.
- Then the key is that you get to **assume that the recursive call does the right thing**, and now your job is to **figure out how to use that result to produce the overall result** that is desired.

Recursion

Pattern: How to write a recursive function

UCLA Samueli Computer Science

- Step 1: Find the base case(s).
  - What are the trivial cases? Eg. empty string, empty array, single-item subarray.
  - When should the recursion stop?
- Step 2: Decompose the problem.
  - Take tail recursion as example.

> ➔ Take the first (or last) of the ***n*** items of information
> ➔ Make a recursive call to the rest of **(*n*-1)** items. The recursive call will give you the correct results.
> ➔ Given this result and the information you have on the first (or last item) conclude about current ***n*** items.

- Step 3: Just solve it!

Recursion

Pattern: How a recursive function works

UCLA Samueli Computer Science

```
void printFun(int test)
{
     if (test < 1)
          return;
     else {
          cout << test << " ";
          printFun(test - 1);
          cout << test << " ";
          return;
     }
}

int main()
{
     int test = 3;
     printFun(test);
}
```

- When any function is called from main(), the memory is **allocated to it on the stack**.
- A recursive function calls itself, the memory for a called function is **allocated on top of memory** allocated to calling function and **different copy of local variables** is created for each function call.
- When the base case is reached, the function returns its value to the function by whom it is called and **memory is de-allocated** and the process continues.

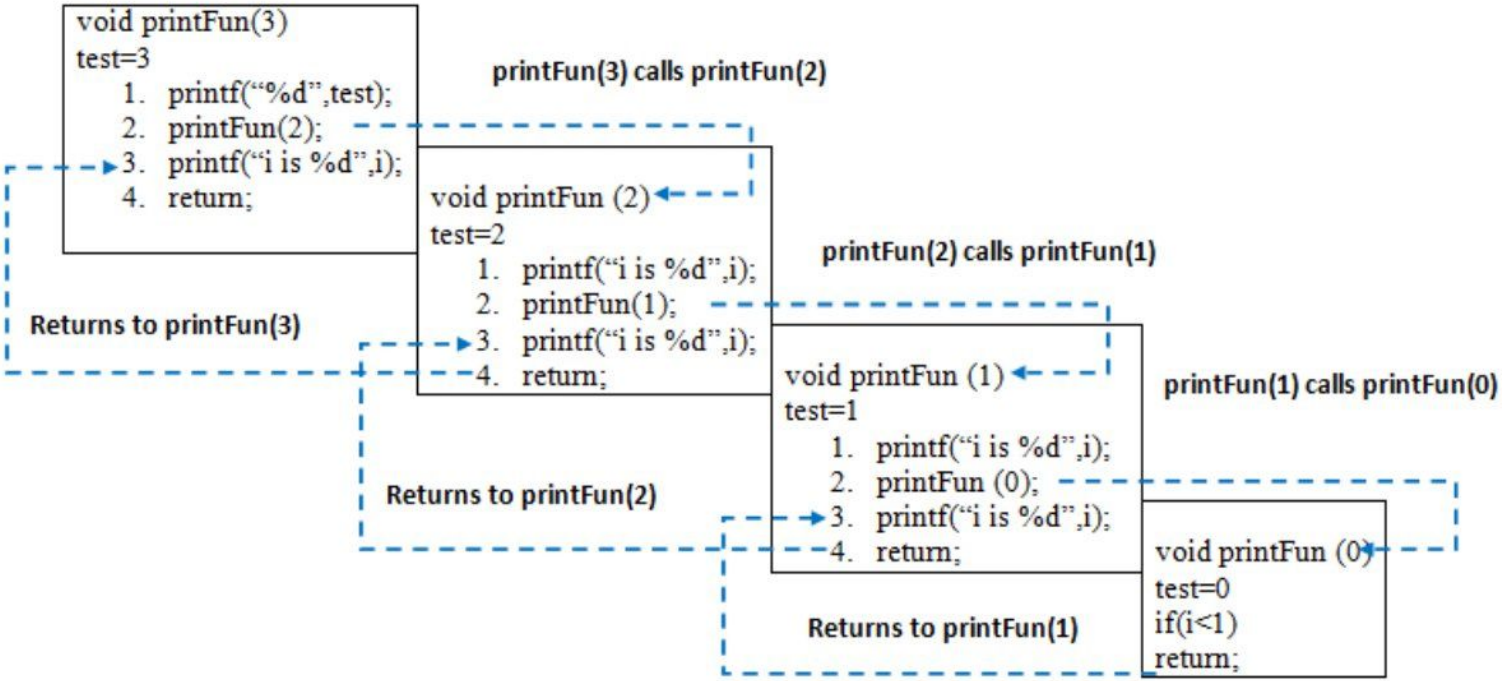
Output :

```
3 2 1 1 2 3
```

Recursion

Pattern: How a recursive function works

UCLA Samueli Computer Science



Recursion

Examples

UCLA Samueli Computer Science

- **C++ Program for Binary Search**
    - Compare x with the middle element.
    - If x matches with middle element, we return the mid index.
    - Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
    - Else (x is smaller) recur for the left half.

We basically ignore half of the elements just after one comparison.

Recursion

Examples

UCLA Samueli Computer Science

```
int binarySearch(int arr[], int l, int r, int x)
{


}
```

Recursion

Examples

UCLA Samueli Computer Science

```
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
   }

   // We reach here when element is not present in array
   return -1;
}
```

Recursion

Examples

UCLA Samueli Computer Science

- Generating subarrays using recursion
    - Given an array, generate all the possible subarrays of the given array using recursion.

Examples:

```
Input : [1, 2, 3]
Output : [1], [1, 2], [2], [1, 2, 3], [2, 3], [3]

Input : [1, 2]
Output : [1], [1, 2], [2]
```

Recursion

Examples

UCLA Samueli Computer Science

```
void printSubArrays (vector<int> arr, int start, int end)
{


}
```

Recursion

Examples

UCLA Samueli Computer Science

```
void printSubArrays (vector<int> arr, int start, int end)
{
    // Stop if we have reached the end of the array
    if (end == arr.size())
        return;

    // Increment the end point and start from 0
    else if (start > end)
        printSubArrays(arr, 0, end + 1);

    // Print the subarray and increment the starting point
    else{
        cout << "[";
        for (int i = start; i < end; i++){
            cout << arr[i] << ", ";}

        cout << arr[end] << "]" << endl;
        printSubArrays(arr, start + 1, end);
    }
    return;
}
```

Recursion

Examples

UCLA Samueli Computer Science

```
void printSubArrays (vector<int> arr, int start, int end)
{
    // Stop if we have reached the end of the array
    if (end == arr.size())
        return;

    // Increment the end point and start from 0
    else if (start > end)
        printSubArrays(arr, 0, end + 1);

    // Print the subarray and increment the starting point
    else{
        cout << "[";
        for (int i = start; i < end; i++){
            cout << arr[i] << ", ";}

        cout << arr[end] << "]" << endl;
        printSubArrays(arr, start + 1, end);
    }
    return;
}
```

Output:

```
[1]
[1, 2]
[2]
[1, 2, 3]
[2, 3]
[3]
```

Recursion

Smallberg's notes

UCLA Samueli Computer Science

What I've emphasized all week is how to understand a recursive solution to a problem and know it's correct:

1. **Identify the base cases** (paths through the function that make no recursive calls) and recursive cases.
2. Come up with measure of the size of the problem for which the base case(s) provide a bottom, typically 0 or 1.
3. Verify that if the function is called with a problem of some size, **any recursive call it makes is to solve a problem of a strictly smaller size**. (Problem sizes should be nonnegative integers.) This proves termination, since a decreasing sequence of nonnegative integers must eventually hit bottom.
4. Now that we've proved termination, **verify that the base cases are handled correctly**.

Recursion

Practice Examples: Merge sort and Quick sort

UCLA Samueli Computer Science

Merge sort

1. Find the middle point to divide the array into two halves:

```
middle m = (l+r)/2
```

2. Call mergeSort for first half:

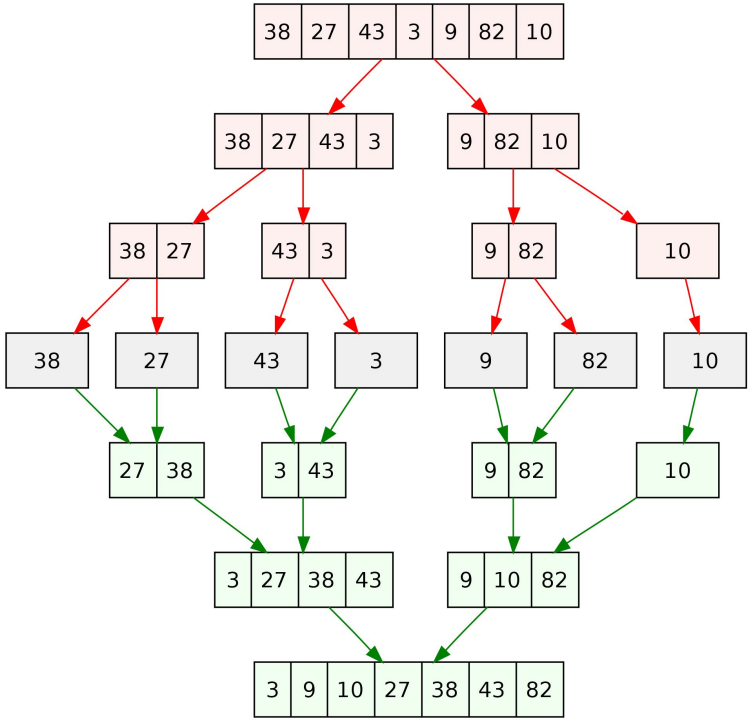
```
mergeSort(arr, l, m)
```

3. Call mergeSort for second half:

```
mergeSort(arr, m+1, r)
```

4 .Merge the two halves sorted in step 2 and 3:

```
merge(arr, l, m, r)
```



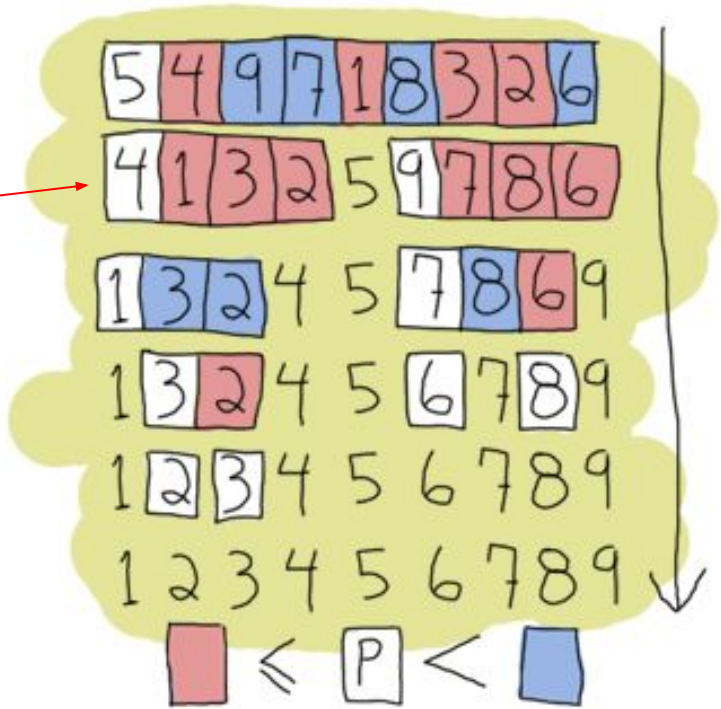
Recursion

Practice Examples: Merge sort and Quick sort

UCLA Samueli Computer Science

Quick Sort

Two recursion calls!



Group Exercises: Worksheet

UCLA Samueli Computer Science

- Exercise problems from **Worksheet #6** (see “LA worksheet” tab in CS32 website). Answers will be posted next week.
- Questions for today:
  - Recursion Code Tracing
  - Delete Linked List (Recursively)
  - getMax
  - sumOverThreshold
  - endX

Group Exercises: Worksheet Prob. #1

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;

int LA_power(int a, int b)
{
    if (b == 0) return 0;
    if (b % 2 == 0) return LA_power(a+a, b/2);
    return LA_power(a+a, b/2) + a;
}

int main()
{
    cout << LA_power(3, 4) << endl;
}
```

What does this code print?

What does the function do?

Group Exercises: Worksheet Prob. #1

UCLA Samueli Computer Science

```
#include <iostream>
using namespace std;

int LA_power(int a, int b)
{
    if (b == 0) return 0; // 1
    if (b % 2 == 0) return LA_power(a+a, b/2); // 2
    return LA_power(a+a, b/2) + a; // 3
}

int main()
{
    cout << LA_power(3, 4) << endl;
}
```

1st recursive call:
    Enter 2: LA_power(6, 2)

2nd recursive call
    Enter 2: LA_power(12, 1)

3rd recursive call
    Enter 3: LA_power(24, 0) + 12

4th recursive call:
    Enter 1: 0

Trace back up the chain:
    LA_power(24, 0) = 0
    LA_power(12, 1) = 12
    LA_power(6, 2) = 12
    return 12

Group Exercises: Worksheet Prob. #2

UCLA Samueli Computer Science

Given a **singly-linked list class LL** with a member variable *head* that points to the first *Node* struct in the list, write a function to **recursively delete the whole list**. Assume each Node object has a next pointer.

```
void LL::deleteList()
```

Group Exercises: Worksheet Prob. #2

UCLA Samueli Computer Science

Base Cases:

- Empty list - return

Subproblem:

- Delete a smaller version of the original linked list

Algorithm

- If empty (head == nullptr) -> return
- Delete rest of the list (head->next)
- Delete current node

```
void LL::deleteList(){
  // base case
  if(!head) return;
  // need to store the current node
  Node* curr = head;
  head = head->next;

  // delete rest of list
  deleteList()
  // delete current node
  delete curr;
}
```

Group Exercises: Worksheet Prob. #3

UCLA Samueli Computer Science

Implement the function getMax recursively. The function **returns the maximum** value in a, an integer array of size n. You may assume that n will be at least 1.

```
int getMax(int a[], int n);
```

Group Exercises: Worksheet Prob. #3

UCLA Samueli Computer Science

Base Cases:

- One element

Subproblem:

- Get the maximum value of the rest of the elements in the array

Algorithm

- If there is one element -> return that element
- Get the max of the rest of array (arr + 1)
- Return the greater value between the current value and the maximum of the rest of the array

```
int getMax(int a[], int n){
  if(n == 1) return a[0];
  int max = getMax(a + 1, n - 1);
  return a[0] > max ? a[0] : max;
}
```

Group Exercises: Worksheet Prob. #4

UCLA Samueli Computer Science

Rewrite the following function **recursively**. You can add new parameters and completely change the function implementation, but you can't use loops.

This function **sums the numbers** of an array **from left to right** until the sum exceeds s**ome threshold**. At that point, the function returns the running sum. Returns -1 if the threshold is not exceeded before the end of the array is reached.

```
int sumOverThreshold(int x[], int length, int threshold);
```

Group Exercises: Worksheet Prob. #4

UCLA Samueli Computer Science

Base Cases:

- One element:
  - less than threshold -> return -1
  - more than threshold -> return element

Subproblem:

- Whether or not the rest of the array can sum over the threshold or not

Algorithm

- Base Case
- Run function on the rest of the array
- If the return value is -1 -> return -1
- Else -> return sum of current value and returned value

```
int sumOverThreshold(int x[], int length, int threshold) {
  if(length == 1){
    return x[0] < threshold ? -1 : x[0];
  }
  int sub = sumOverThreshold(x + 1, length - 1, threshold - x[0])
  return sub == -1 ? -1 : x[0] + sub;

}
```

Group Exercises: Worksheet Prob. #5

UCLA Samueli Computer Science

Given a string *str*, recursively compute a new string such that **all the 'x' chars have been moved to the end**.

```
string endX(string str);
```

Group Exercises: Worksheet Prob. #5

UCLA Samueli Computer Science

Base Cases:

- Empty string -> return empty string

Subproblem:

- First character is ‘x’
  - return function on str + 1 + ‘x’
- First character is not ‘x’
  - return str[0] + function on str + 1

Algorithm

- Base Case
- If str[0] == ‘x’
  - return endX(str + 1) + ‘x’
- Else
  - return str[0] + endX(str + 1)

```
string endX(string str) {
  if (str.length() <= 1)
    return str;
  if (str[0] == 'x')
    return endX(str.substr(1)) + 'x';
  else
    return str[0] + endX(str.substr(1));
}
```