

Recursion

Sort Algorithms

CS 16: Solving Problems with Computers I
Lecture #17

Ziad Matni
Dept. of Computer Science, UCSB

FINAL EXAM IS COMING!

DEC 12_{th}!

- Material: **Everything** we've done
 - Homework, Labs, Lectures, Textbook
- **Tuesday, 12/12** in this classroom
- Starts at **4:00pm** ****SHARP**** (come early)
- Ends at **7:00pm** ****SHARP****
- **BRING YOUR STUDENT IDs WITH YOU!!!**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (double-sided ok) of written notes
 - Must be no bigger than 8.5" x 11"
 - You have to turn it in with the exam
- You will write your answers on the exam sheet itself.



**DSP Students: Put in
your requests TODAY!**

Final Exam Preparation

- Your TA office hours
- Your prof's office hours
- Exam prep questions (emailed them via Piazza)
- Exam review session with TAs next Thursday eve
 - **Thursday 12/7 at 5 PM- Phelps 3526**

Lecture Outline

- Recursion (Ch. 14)
- Sorting algorithms

Recursive Functions for Tasks

- **Recursive: (adj.) Repeating unto itself**
- **A recursive function contains a call to itself**
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
- For example: **Searching an array**
 - Could be divided into searching the 1st, then 2nd halves of array
 - Searching each half is a smaller version of searching the whole array

Example: The Factorial Function

Recall: $x! = 1 * 2 * 3 \dots * x$

You could code this out as either (the following is pseudocode):

- A for-loop:

```
(for k=1; k < x; k++) { factorial *= k; }
```

- Or a recursion/repetition:

```
factorial(x) = x * factorial(x-1)
             = x * (x-1) * factorial (x-2)
             = etc...
             until you get to factorial(1)
```


Example: Recursive Formulas

- Recall from Math, that you can create a recursive formula from a sequence

Example:

- Consider the arithmetic sequence:

5, 10, 15, 20, 25, 30, ...

- If I call $a_1 = 5$, then I can write the formula as:

$$\mathbf{a_n = a_{n-1} + 5}$$

Starting Point (aka Base Case)

- If we start with $n = 1$... (an arbitrary value)
- ... then we could devise an algorithm like this:

$$a_n = a_{n-1} + 5$$

1. If $n = 1$, then **return 5** to $a(n)$
 - This is called the base-case
 2. Otherwise, **return $a(n-1) + 5$**
 - This is the recursion (i.e. function calling itself)
- Example: $n = 3$
 - **According to [2]:** $a(n) = a(3) = a(2) + 5 = (a(1) + 5) + 5$
 - **According to [1]:** Since $a(1) = 5$, then $a(3) = (5 + 5) + 5 = \underline{15}$

Case Study: Vertical Numbers

- Problem Definition:
Write a recursive function that takes an integer number and prints it out one digit at a time vertically :

```
void write_vertical( int n );  
//Precondition:  n >= 0  
//Postcondition: n is written to the screen vertically  
//              with each digit on a separate line
```

```
write_vertical(3):  
3  
write_vertical(12):  
1  
2  
write_vertical(123):  
1  
2  
3
```

Case Study: Vertical Numbers

Analysis:

- Take a number, like 543.
- How do I separate the digits from each other?
 - So that I can print out **5**, then **4**, then **3**?
- Hint: Note that $543 = 500 + 40 + 3$

Case Study: Vertical Numbers

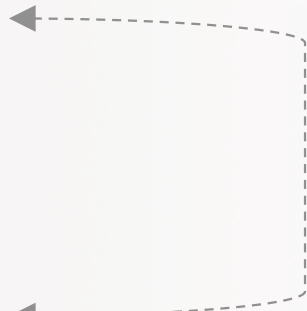
Algorithm design

- *Simplest case:*
If **n** is 1 digit long, just write the number
- *More typical case:*
 - 1) Output all but the last digit vertically (recursion!)
 - 2) Write the last digit (base case!)
 - *Step 1 is a smaller version of the original task - The recursive case*
 - *Step 2 is the simplest case - The base case*

Case Study: Vertical Numbers

The **write_vertical** algorithm (in pseudocode):

```
void write_vertical( int n )  
{  
    if (n < 10)  cout << n << endl;  
    // n < 10 means n is only one digit  
  
    else // n is two or more digits long  
    {  
        write_vertical(n with the last digit removed);  
        cout << the last digit of n << endl;  
    }  
}
```



Case Study: Vertical Numbers

- **Note that: $n / 10$ (integer division)**
returns n with *just the least-significant digit removed*
 - So, for example, $85 / 10 = 8$ or $124 / 10 = 12$
- **Whereas: $n \% 10$** returns the *least-significant digit of n*
 - In this example, $124 \% 10 = 4$
- **How might we combine these in the function?**

Case Study: Vertical Numbers

The ***write_vertical*** function in C++

```
void write_vertical( int n )
{
    if (n < 10)    cout << n << endl;
    // n < 10 means n is only one digit

    else // n is two or more digits long
    {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

See Display 14.1 in textbook

A Closer Look at Recursion

- The function **write_vertical** uses recursion
 - It simply calls itself with a different argument
- If you want to **track** a recursive call (i.e. to debug it):
 1. Temporarily stop the execution **at** the recursive call
 2. Show or save the result of the call before proceeding
 3. Evaluate the recursive call
 4. Resume the stopped execution

How Recursion Ends

- Recursive functions have to stop eventually
- One of the recursive calls must not depend on another recursive call
- Usually, that's the last recursive call
 - What ends recursion is the **base case**
 - Also called **stopping case**

“Infinite” Recursion

- A function that never reaches a base case, *in theory, will run forever*
- In practice, the computer will often run out of resources (i.e. memory usually) and the program will terminate abnormally
- So... design your recursive functions carefully!

Example: Infinite Recursion

- What if we wrote the function **write_vertical**, *without the base case*

```
void write_vertical(int n)
{
    write_vertical (n / 10);
    cout << n % 10 << endl;
}
```

- Will *eventually* call **write_vertical(0)**,
which will call **write_vertical(0)**,
which will call **write_vertical(0)**,
which will call **write_vertical(0)**, ...etc...

Stacks for Recursion



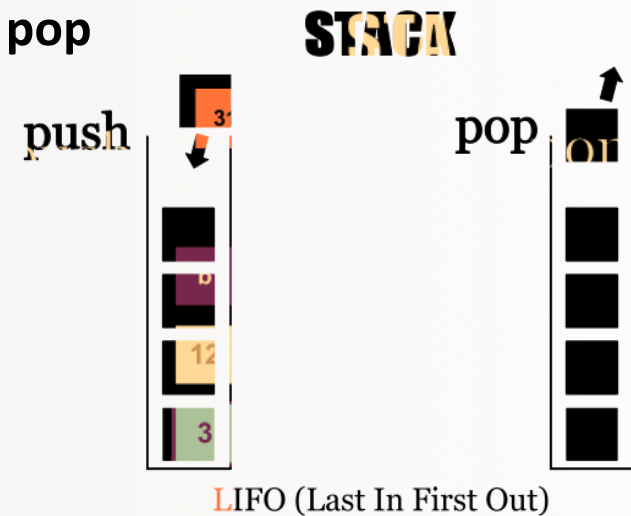
- Computers use a memory structure called a **stack** to keep track of recursion
- **Stack**: a computer memory structure analogous to a **stack of paper**
 - To place information on the stack, write it on a piece of paper and place it on **top** of the stack
 - To **insert more** information on the stack, use a clean sheet of paper, write the information, and place it on the **top** of the stack
 - To **retrieve** information, only the top sheet of paper can be read, and then thrown away when it is no longer needed

LIFO

- This scheme of handling sequential data in a stack is called:

Last In-First Out (LIFO)

- When we put data in a LIFO, we call it a **push**
- When we pull data out of a LIFO, we call it a **pop**
- The other common scheme in CS data organization is FIFO (First In-First Out)



Stacks & Making the Recursive Call

When execution of a function definition reaches a recursive call...

1. Execution is halted (paused)
2. Then, data is saved in a new place in the stack
 - It's part of **computer memory**, but think of it as a “clean sheet of paper”
3. The “sheet of paper” is placed *on top of the stack*
4. Then a *new* sheet is used for the recursive call
 - a) A new function definition is written, and arguments are plugged into parameters
 - b) Execution of the recursive call begins
5. And it goes on...

Stacks & Ending Recursive Calls

When a recursive function call is able to complete its computation with *no* recursive calls...

1. The computer retrieves the **top** “sheet of paper” from the stack
 - Resumes computation based on the information on the sheet
2. When that computation ends, that sheet of paper is “discarded”
3. The next sheet of paper on the stack is retrieved so that processing can resume
4. The process continues until no sheets remain in the stack

Activation Frames

- Instead of “paper”, think “memory” ...
- Portions of computer memory are used for the stack
 - The contents of these portions of memory is called an ***activation frame***
- Because each recursive call causes an **activation frame** to be placed on the stack
 - Infinite recursions can force the stack to grow **beyond** its limits

Stack Overflow

- Infinite recursions can force the stack to grow **beyond** its limits
- The result of this erroneous operation is called a **stack overflow**
 - This causes abnormal termination of the program



Recursion versus Iteration

Algorithmic Truism:

- Any task that can be accomplished using recursion *can also be done* without recursion (usually using loops)
- A **non-recursive** version of a repeating function is called an ***iterative-version***
- A **recursive** version of a function...
 - Usually runs a little slower
 - BUT it uses code that is *easier to write and understand*

Recursive Functions for *Values*

- Recursive functions don't have to be **void** types like the last example
 - They can also return values
- The technique to design a recursive function that returns a value is basically the same as what we described...

Program Example: A Powers Function

Example: Define a new **power** function (not the one in <cmath>)

- Let it return an integer, **2³**, when we call the function as: **int y = power(2,3);**
- Use the following definition: $x_n = x_{n-1} * x$ *i.e.* $2^3 = 2^2 * 2$
 - Note that this only works if n is a positive number
- Translating the right side of that equation into C++ gives: **power(x, n-1) * x**
 - What is the base/stopping case?
 - *It's when **n = 0**, then power() should return **1***

```
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Cannot use negative powers in this function!\n";
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1)*x );

    else // i.e. if n ==0
        return (1);
}
```

Stopping case

Tracing *power*(2, 3)

- **power(2, 3)** results in the following recursive calls:

– power(2, 3) is power(2, 2) * 2

– power(2, 2) is power(2, 1) * 2

– power(2, 1) is power(2, 0) * 2

– power (2, 0) is 1 (stopping case)

Therefore:

power(2,3)

= power(2,2) x 2

= (power(2,1) x 2) x 2

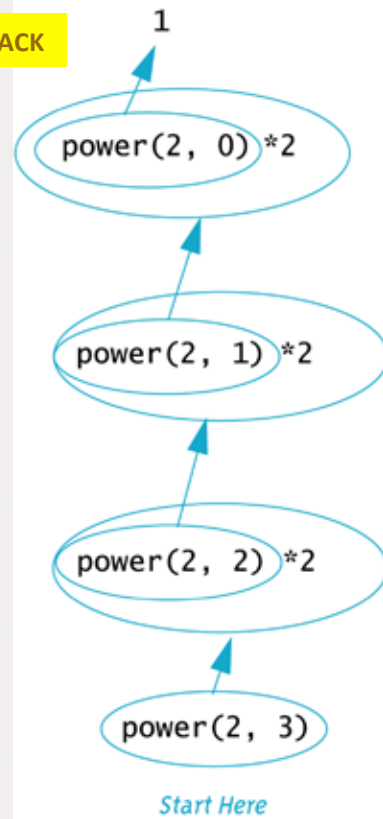
= ((power(2,0) x 2) x 2) x 2

= 1 x 2 x 2 x 2

= 8

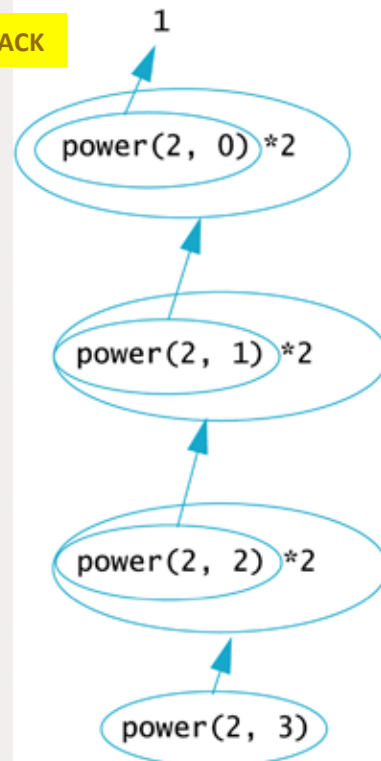
Sequence of recursive calls

PUSH INTO THE STACK



Sequence of recursive calls

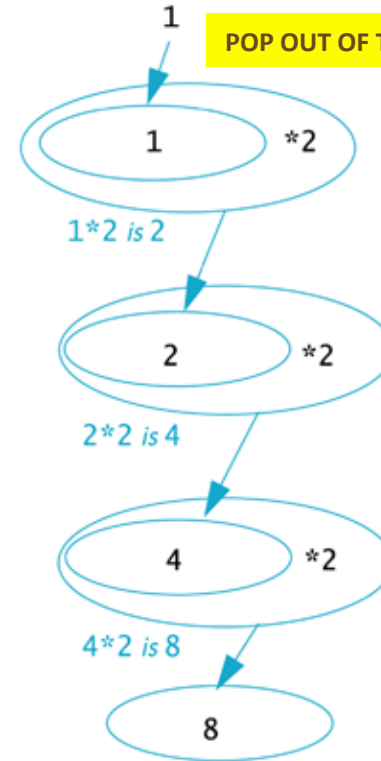
PUSH INTO THE STACK



Start Here

How the final value is computed

POP OUT OF THE STACK



power(2, 3) is 8

Thinking Recursively

- When designing a recursive function, you do not need to trace out the entire sequence of calls
- Instead just check the following:
 - That there is **no infinite recursion**,
i.e. that, eventually, a stopping case is reached
 - That each **stopping case** returns the correct value
 - That the **final value** returned is the correct value

Sorting

Sorting a Data Structure

- Sorting a list of values is another very common task
 - Create an alphabetical listing
 - Create a list of values in ascending order
 - Create a list of values in descending order
- Many sorting algorithms exist
 - Some are very efficient
 - Some are easier to understand

Some common sorting algorithms

Bucket sort
Bubble sort
Insertion sort
Selection sort
Heapsort
Mergesort

  Random	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Restart all								
 Nearly Sorted								
 Reversed								
 Few Unique								

The Selection Sort Algorithm

As used with an array

- When the sort is complete, the elements of the array are ordered in ascending order, such that:

$$a[0] < a[1] < \dots < a[\text{number_used} - 1]$$

- This leads to an outline of an algorithm:

*for (int index = 0; index < number_used; index++)
 place the indexth smallest element in a[index]*

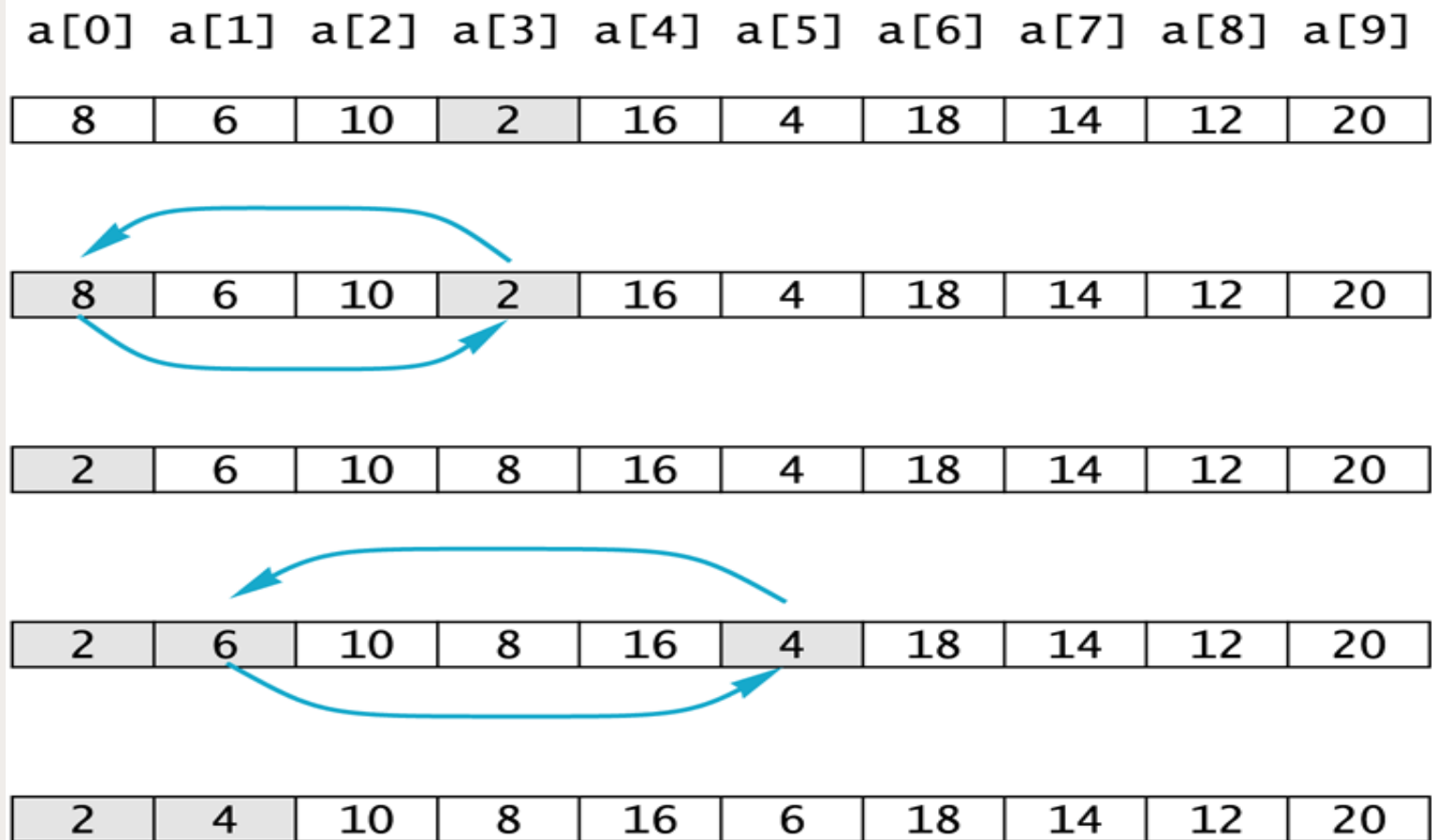
Sort Algorithm Development

- One array is sufficient to do our sorting *(See Display 7.11 in the textbook)*
- Start by searching for the *smallest* value in the array
- Place this value in $a[0]$, and place the value that was in $a[0]$ in the location where the smallest was found
 - i.e. swap them
- Then, starting at $a[1]$, find the smallest remaining value swap it with the value currently in $a[1]$
- Then, starting at $a[2]$, continue the process until the array is sorted

Selection Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	20

Selection Sort



```
void fill_array(a[], N, number_used);
void sort_array(a[], number_used);
void print_array(const a[], number_used);

void swap_values(int& v1, int& v2);
void index_of_smallest
    (const a[], start_index, number_used);

int main()
{
    int sample_arr[10], number_used;

    fill_array(sample_array, 10, number_used);
    sort_array(sample_array, number_used);
    print_array(sample_array, number_used);

    return 0;
}
```

```
void sort_array(a[], number_used)
{
    int index_of_next_smallest;

    for (int i = 0; i < number_used - 1; i++)
    {
        index_of_next_smallest = index_of_smallest(a, i, number_used);

        swap_values(a[i], a[index_of_next_smallest]);
    }
}
```

```
void swap_values(int& v1, int& v2)
{
    int temp = v1;
    v1 = v2;
    v2 = temp;
}
```

```
int index_of_smallest(const int a [], int start_index, int number_used)
{
    int min = a[start_index], index_of_min = start_index;

    for (int i = start_index + 1; i < number_used; i++)
        if (a[i] < min)
        {
            min = a[i];
            index_of_min = i;
        }

    return index_of_min;
}
```

DISPLAY 7.12 Sorting an Array (part 1 of 2)

```

1 //Tests the procedure sort.
2 #include <iostream>

3 void fill_array(int a[], int size, int& number_used);
4 //Precondition: size is the declared size of the array a.
5 //Postcondition: number_used is the number of values stored in a.
6 //a[0] through a[number_used - 1] have been filled with
7 //nonnegative integers read from the keyboard.

8 void sort(int a[], int number_used);
9 //Precondition: number_used <= declared size of the array a.
10 //The array elements a[0] through a[number_used - 1] have values.
11 //Postcondition: The values of a[0] through a[number_used - 1] have
12 //been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

13 void swap_values(int& v1, int& v2);
14 //Interchanges the values of v1 and v2.

15 int index_of_smallest(const int a[], int start_index, int number_used);
16 //Precondition: 0 <= start_index < number_used. Referenced array elements have
17 //values.
18 //Returns the index i such that a[i] is the smallest of the values
19 //a[start_index], a[start_index + 1], ..., a[number_used - 1].

20 int main()
21 {
22     using namespace std;
23     cout << "This program sorts numbers from lowest to highest.\n";

24     int sample_array[10], number_used;
25     fill_array(sample_array, 10, number_used);
26     sort(sample_array, number_used);

27     cout << "In sorted order the numbers are:\n";
28     for (int index = 0; index < number_used; index++)
29         cout << sample_array[index] << " ";
30     cout << endl;

31     return 0;
32 }

33 //Uses iostream:
34 void fill_array(int a[], int size, int& number_used)
35 void sort(int a[], int number_used)
36 {
37     int index_of_next_smallest;

```

<The rest of the definition of fill_array is given in Display 7.9.>

(continued)

DISPLAY 7.12 Sorting an Array (part 2 of 2)

```

38     for (int index = 0; index < number_used - 1; index++)
39         { //Place the correct value in a[index]:
40             index_of_next_smallest =
41                 index_of_smallest(a, index, number_used);
42             swap_values(a[index], a[index_of_next_smallest]);
43             //a[0] <= a[1] <= ... <= a[index] are the smallest of the original array
44             //elements. The rest of the elements are in the remaining positions.
45         }
46     }
47
48 void swap_values(int& v1, int& v2)
49 {
50     int temp;
51     temp = v1;
52     v1 = v2;
53     v2 = temp;
54 }
55
56 int index_of_smallest(const int a[], int start_index, int number_used)
57 {
58     int min = a[start_index],
59     index_of_min = start_index;
60     for (int index = start_index + 1; index < number_used; index++)
61         if (a[index] < min)
62         {
63             min = a[index];
64             index_of_min = index;
65             //min is the smallest of a[start_index] through a[index]
66         }
67     return index_of_min;
68 }
69 }

```

Sample Dialogue

This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90

C

YOUR TO-DOs

- ☐ HW 9 due Thu. 12/7
- ☐ Lab 9 due Wed. 12/6
- ☐ Visit Prof's and TAs' office hours if you need help!
- ☐ **STUDY FOR YOUR FINAL EXAM!!!**

</LECTURE>