EECE 230 Introduction to Programming, Sections 3,4, and 12
Programming Assignment 8

Tue Nov 20, 2012

- This programming assignment consists of 6 problems.
- Related material: recursion (... Recursive Binary Search, Merge Sort, Tower of Hanoi).
- Due dates:
  * Problems 1,2,3 will be graded on Tue Nov 27 in the Lab.
    Essentially, we covered the background material of Problems 1, 2, and 3 on Tue Nov 20. We will elaborate on Tue Nov 27.
  * Problems 4,5,6 will be graded on Tue Dec 4 in the Lab (Correction: Dec not Jan)
- *Lab structure and regulations:*
  ⋆ The 3 hours Lab session is on Tuesdays in Lab rooms 1,2 and 5 from 2:00 pm to 5:00 pm. It consists of three parts:
    - *Occasional Solving Session (not graded but attendance mandatory)*
    - *Programming Assignment (graded)*
      Programming Assignments will be posted on Moodle on weekly basis. Typically, a Programming Assignment requires much more than the time allocated for this part in the Lab, so you are supposed to complete the major part of the assignment at home. The Lab instructor will grade your assignment and can help you with the problems you are facing.
    - *Occasional graded weekly quiz*
  ⋆ You are supposed to submit your own work. Cheating will not be tolerated and will be dealt with severely: zero grades on the programming assignments, disciplinary committee, Dean's warning.
  ⋆ Lab attendance is mandatory. Violating this rule can lead to a failing grade.

Problem 1 (Recurrences)

For each of the following recurrences, write a iterative function and a recursive function to compute it.

a)

$$a_n = \begin{cases} 1 & \text{if } n = 1 \\ 2a_{n-1} + 1 & \text{if } n \geq 2 \end{cases}$$

Thus $a_1 = 1, a_2 = 2 \times 1 + 1 = 3, a_3 = 2 \times 3 + 1 = 7, \ldots$

b)

$$b_n = \begin{cases} 1 & \text{if } n = 1 \\ b_{\lfloor n/2 \rfloor} + b_{\lceil n/2 \rceil} + n & \text{if } n \geq 2, \end{cases}$$

where, if x is a real number,

- $\lfloor x \rfloor$ means the floor of x, i.e., the largest integer less than or equal to x
- $\lceil x \rceil$ means the ceiling of x, i.e., the smallest integer greater than or equal to x

c)

$$c_n = \begin{cases} x & \text{if } n = 1 \\ y & \text{if } n = 2 \\ z & \text{if } n = 3 \\ c_{n-1} + c_{n-2} + c_{n-3} & \text{if } n \geq 4, \end{cases}$$

where x, y, and z are variables that you are supposed to pass to the functions as parameters.

Call the functions: *iterativeA*, *recursiveA*, *iterativeB*, *recursiveB*, *iterativeIC*, *iterativeIIC*, and *recursiveC*, where *iterativeIC* does not use arrays, and *iterativeIIC* is array based.

Is possible to implement *iterativeB* without using arrays?

Write a program to test your functions. Your test program should allow the user to enter the values of x, y, and z (for Part (c)).

d) Which of the above recursive functions is highly nonefficient compared its iterative counterpart?

Add a global variable to *recursiveC* and increment it each time the function is visited. How many time is it called when $n = 5, 10, 50, 100$?

Do the same for *recursiveA* and *recursiveB*.

Problem 2 (Power function: recursive versions)

a) **(Slow recursive power function)**

Do Programming Exercise 16.9 [Malik, page 977 in the third edition]:

Write a recursive function, *power*, that takes as parameters two integers x and y such that x is nonzero and returns x^y. Use the following recursive definition to calculate x^y. If $y \geq 0$,

$$power(x, y) = \begin{cases} 1 & \text{if } y = 0 \\ x & \text{if } y = 1 \\ x \times power(x, y - 1) & \text{if } y > 1 \end{cases}$$

If $y < 0$,

$$power(x, y) = \frac{1}{power(x, -y)}$$

b) **(Smart recursive power function)**

Write a more efficient recursive function.

Below is the idea illustrated via an example.

⋆ The naive method to compute x^8 is

x
$x^2 = xx$
$x^3 = x^2x$
$x^4 = x^3x$
$x^5 = x^4x$
$x^6 = x^5x$
$x^7 = x^6x$
$x^8 = x^7x$

A smarter and much more efficient method is

x
$x^2 = xx$
$x^4 = x^2x^2$
$x^8 = x^4x^4$

⋆ The naive method to compute x^9 is

x
$x^2 = xx$
$x^3 = x^2x$
$x^4 = x^3x$
$x^5 = x^4x$
$x^6 = x^5x$
$x^7 = x^6x$
$x^8 = x^7x$
$x^9 = x^8x$

A smarter and much more efficient method is

x
$x^2 = xx$
$x^4 = x^2x^2$
$x^8 = x^4x^4$
$x^9 = x^8x$

Assume that $y > 0$. Write a recursive definition to calculate x^y based on the idea in the above example. The recurrence is supposed to contain conditions based on whether y is even or odd.

Once you figure out the recurrence, the recursive function follows easily.

At the end take care of the cases: $y < 0$ and $y = 0$.

In addition to the C++ code, you are supposed submit the recurrence (commented).

Problem 3 (Selection Sort revisited: recursive version)

Write a recursive version *recursiveSelectionSort* of the Selection Sort algorithm. Follow the guidelines of Problem 2 of PA 6, i.e., use the *arraySmallest* and *swap* functions.

(*Hints:*

- The recursive function *recursiveSelectionSort* is very compact (around 4 lines!), and it does not contain for loops.
- *recursiveSelectionSort* takes 3 parameters: (a pointer to) the array A, the starting index *start*, and the final index *end*. The initial call is $recursiveSelectionSort(A, 0, n-1)$.)

Problem 4 (Finding the mode of a unimodal array)

An array $A[0 \ldots n-1]$ of size $n \geq 2$ is called *unimodal* if there exists an integer m, $0 \leq m \leq n-1$, such that $A[i-1] < A[i]$ for $i = 1, \ldots, m$ and $A[i-1] > A[i]$ for $i = m+1, \ldots, n-1$. We call m the *mode* of A.

For example the array

$$A[] = \{1, 2, 4, 7, 11, 10, 8, 4, -9\}$$

is unimodal. Its mode is $m = 4$ since:

i. $1 < 2 < 4 < 7 < 11$

ii. $A[4] = 11$

iii. $11 > 10 > 8 > 4 > -9$

Consider the following problem:

Find-Mode-Problem: Given a (pointer A) to a unimodal array and its size n, find the mode of A.

a) **(Naive Sequential Mode Finder)**

Write an iterative function

$$\text{int } sequentialModeFinder(\text{int* A,int n})$$

for the *Find-Mode-Problem*.

Mimic the idea of the Sequential Search algorithm (pass over the elements of A one by one and break the loop when you find the mode).

Note that you are not supposed to check if the array is unimodal. Your algorithm must assume that the array is unimodal, and it is not supposed to work correctly if the array is not unimodal.

b) **(Smart Recursive Mode Finder)**

Write an efficient recursive function

$$\text{int } recursiveModeFinder(\text{int* A,int start,int end})$$

for the *Find-Mode-Problem*.

Mimic the idea of the recursive Binary Search algorithm (consider the middle of the array and recur on the upper subarray or the lower subarray after suitable considerations).

Note that the initial call of the function is $recursiveModeFinder(A, 0, n-1)$.

Here again, you are not supposed to check if the array is unimodal. Your algorithm must assume that the array is unimodal, and it is not supposed to work correctly if the array is not unimodal.

Why is $recursiveModeFinder$ much more efficient than $sequentialModeFinder$?

c) **(Smart Sequential Mode Finder)**

Write an efficient iterative function

$$\text{int } fastInterativeModeFinder(\text{int* A,int n})$$

for the *Find-Mode-Problem*.

Mimic the idea of the iterative Binary Search algorithm (consider the middle of the array and ignore either the lower or the upper subarray after suitable considerations).

Write a program to test your functions.

Problem 5 (Quick Sort)

a) Quick Sort

The idea of the Quick Sort algorithm is the following. To sort an array $A[0 \dots n-1]$:

1. Let $x = A[n-1]$. Partition A around x using the partition algorithm [Programming Assignment 7], and let q be the new index of x in the reordered array (q is returned by the partition function). Thus now A is rearranged in such a way that:
   i. Each element of $A[0 \dots q-1]$ is less than or equal to x
   ii. $A[q] = x$
   iii. Each element of $A[q+1 \dots n-1]$ is greater than or equal to x
2. Recursively sort $A[0 \dots q-1]$ if the size of this subarray is greater than one
3. Recursively sort $A[q+1 \dots n-1]$ if the size of this subarray is greater than one

Try the idea on a small example, write a pseudocode, then implement the quickSort function

$$void\ quickSort(int\ A[], int\ start, int\ end).$$

As usual the initial call is $quickSort(A, 0, n-1)$.

Use the code of your partition function from PA 7.

Note that, unlike mergeSort, quickSort does not use temporary arrays, which makes it very practical

b) Randomized Quick Sort

Now, instead of partitioning the array A around the last element $A[n-1]$, consider partitioning A around a random element of A. We can do this by swapping the last element $A[n-1]$ with a (uniformly) random element element of A (i.e., an element $A[r]$ of A, where r is an element chosen (uniformly) at random from the set of indices $\{0, \ldots, n-1\}$), and then using the partition function. Do this, not only at the initial call, but each time the partition function is called.

Implement this randomized version of Quick Sort and call your function *randQuickSort*.

To generate (pseudo) random numbers, use the *rand*() function [Malik, page 1480]. In the main function use the command

```
srand((unsigned)time( NULL ));
```

to seed the pseudo-random-number generator with the current time so that the numbers will be different every time we run. Here is an example illustrating the usage of the *rand*() function: to generate a random element from the set $\{5, 6, 7, 8, 9, 10, 11\}$, we can use $5 + rand()\%7$.

This is the randomized Quick Sort algorithm, which is one of the fastest known sorting algorithms. The randomness introduced in randomized Quick Sort guarantees that the split generated by partition is well balanced with a high probability.

Write a program to test both functions.

Problem 6 (String Enumerator)

a) Print All

A length-n *abc*-string is a C-string $s[0 \ldots n]$ such that: for $i = 0, \ldots, n-1$, $s[i]$ is either the character 'a', or the character 'b', or the character 'c' (as usual $s[n]$ is the NULL character).

Write a recursive function *printAll*, which given an integer $n > 1$, prints all possible length-n *abc*-strings.

For example, if $n = 2$, *printAll* is supposed to print:

aa
ab
ac
ba
bb
bc
ca
cb
cc

Write a program to test your function.

(*Hints:*

- ⋆ *printAll* makes 3 recursive calls
- ⋆ You have to use a temporary *char*-array to keep track of the string under consideration. You can pass it (by reference) to the function, or declare it as a static variable in the function. You can also make it global but you are encouraged to avoid this.

⋆ The code is very compact. You can do it in around 10 statements.)

The following two parts ((b) and (c)) are both related to (a) but they are independent.

b) Store All

In this part, we are interested in length-4 *abc*-strings.

Modify *printAll* so that instead of printing all possible length-4 *abc*-strings, it stores them in a two-dimensional array $M[0 \dots 3^4 - 1][0 \dots 4]$.

Call this recursive function *storeAll*.

You are not supposed to use global variables. Compared to *printAll*, *storeAll* has two additional parameters: the matrix M (passed by reference) and a counter passed by reference also.

Write a program to test your function (the test program simply calls the function and prints the matrix).

c) Weight

This part is probably more difficult than the previous ones, so please give it enough time.

Now, we are interested in length-n binary strings.

An length-n *binary-string* is a C-string $x[0 \dots n]$ such that: for $i = 0, \dots, n-1$, $x[i]$ is either the character '0', or the character '1' (as usual $x[n]$ is the NULL character).

If $x[0 \dots n]$ is length-n binary-string, we define the *weight* w of x to be the number of entries of x which are equal to 1, i.e., w is equal to the number of indices i, $0 \leq i \leq n-1$, such that $x[i] = 1$.

Inspired by *printAll*, write a recursive function *printAllC*, which given a integer $n > 1$ and an integer $k \geq 0$, prints all length-n binary-strings of weigh less than or equal to k.

For example, if $n = 5$ and $k = 2$, *printAllC* is supposed to print (not necessarily in the shown order):

```
00000
10000
01000
00100
00010
00001
11000
10100
10010
10001
01100
01010
01001
00110
00101
00011
```

(*Hint: printAllC* makes either 2 recursive calls or 1 recursive call depending on some conditions. If you implement the function in such a way that it always makes two recursive calls (i.e., the straight forward modification of part (a)), it will take thousands of years (in principle) when $n = 1000$ and $k = 1$. If you smartly decide on whether you need one or two recursive calls, it will take seconds on $n = 1000$ and $k = 1$.)

d) (Optional) Permutation enumerator

A length-n permutation is a reordering of the integers $1, 2, \dots, n$.

Note that each integer from 1 to n appears in the reordering exactly once.

For instance 1432 and 4123 are length-4 permutations.

Write a recursive function *printAllPermutations* which given an integer n, prints all length-n permutations.

For example, if $n = 4$, *printAllPermutations* is supposed to print:

1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3214
3241
3124
3141
3421
3412
4231
4213
4324
4313
4123
4132

Write a program to test your function.