**CS-251, Fall 2017**

**Written Homework 1**

**Due Monday Jan 29, by 8:00am**

**Submission will be done using gradescope (you will scan and upload your written homework). Details of the gradescope submission process will be posted to Piazza.**

* **Your writeup must be neat and clear**
* **There are 6 problems, some with multiple parts; clearly label your answers.**

**Each Problem will be scored out of 20 points (for a total of 120 points).**

|  |  |
| --- | --- |
| **PROBLEM 1:** The function **has\_dups** to the right determines if a given array of n elements has any duplicate elements: if at least one value appears two or more times, **true** is returned (it "has duplicates"); otherwise it returns **false** (all elements are distinct: it does not "have duplicates").  Take a few minutes to understand the logic of the function and why it works. | **bool has\_dups(int a[], int n){**  **int i, j;**  **for(i=0; i<n; i++) {**  **for(j=i+1; j<n; j++) {**  **if(a[i] == a[j])**  **return true;**  **}**  **}**  **return false;**  **}** |
| **Your job:** write a linked-list version of ***exactly***the same algorithm. A linked list is a sequence of elements just like an array after all -- i.e., a given linked list either has duplicates or it does not.  Use the struct and function prototype below. | |

|  |
| --- |
| struct NODE {  int val;  NODE \*next;  };  bool has\_dups(NODE \*lst) {  NODE \*pi, \*pj;  for(pi=lst; pi != NULL; pi = pi->next) {  for(pj=pi->next; pj != NULL; pj = pj->next) {  if(pi->val == pj->val)  return true;  }  }  return false;  } |
| // while loop version  bool has\_dups(NODE \*lst) {  NODE \*pi, \*pj;  pi=lst;  while(pi != NULL) {  pj=pi->next;  while(pj != NULL) {  if(pi->val == pj->val)  return true;  pj = pj->next  }  pi = pi->next;  }  return false;  } |

You will submit a scanned hardcopy (hand-written or printed). Of course, you are free to try out your solution in a real program.

**PROBLEM 2:** Below is a (trivial) C function which returns the square of its parameter (a non-negative integer):

|  |
| --- |
| unsigned int square(unsigned int n) {  return n\*n;  } |

Your job: write a function which also returns but with the following constraints:

* You cannot use the multiplication operator ‘\*’
* You cannot use the division operator ‘/’
* You cannot have any loops
* You cannot add any additional parameters to the function
* Your function must be self-contained: no helper functions!
* You cannot use any globals
* You cannot use any static variables
* You cannot use any "bit twiddling" operations -- no shifts, etc.

However, …

* You *can* use recursion
* You *can* use the ‘+’ and ‘-’ operators.

You will submit a scanned hardcopy (hand-written or printed) or pdf via gradescope. Of course, you are free to try out your solution in a real program.

|  |
| --- |
| Addendum: derivation required!  You must explain the logic of your solution! (Explain how you derived it).  Just giving a correct C++ function is not sufficient and will not receive many points (possibly zero!) |

|  |
| --- |
| **SOLUTION** |
| Ideas:  If we are using recursion, we’d better have at least one base case.  That should be easy:  if(n==0) return 0;  Otherwise (n>0), suppose we recursively compute  How can we used to determine ?? Well…  **^**    there’s that pesky we want. Let’s solve for it:  There’s still a pesky multiplication in there (the term). To fix that:  [translated into C++ below and also in demo program square.c] |
| unsigned int sq1(unsigned int n) {  if(n==0)  return 0;  n\_1\_sq = square(n-1); // computes (n-1)^2 and stores it.  **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **\* our job: return n^2**  **\* we know: (n-1)^2**  **\* = n^2 - 2n + 1**  **\***  **\* our answer is in there!**  **\***  **\* n^2 = [n^2 - 2n + 1] + 2n -1**  **\* = n\_1\_sq + n + n - 1**  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**  return n\_1\_sq + n + n - 1;  } |
| // alternative implementation of the same idea  // -- gets rid of the temporary variable  unsigned int sq2(unsigned int n) {  if(n==0) return 0;  return sq2(n-1) + n + n - 1;  } |
| // Just for fun… an interesting recursive solution that  // breaks some of the given rules.  // does our recursive call have to be square(n-1)?  // maybe not…  // can you figure out what it is doing and why it works?  unsigned int sq3(unsigned int n) {  unsigned int tmp;  if(n==0)  return 0;  tmp = sq3(n >> 1); // is this cheating? yes...  /\*\*  \* note, this is equivalent to:  \*  \* tmp = sq3(n/2);  \*/  tmp += tmp;  tmp += tmp;  if(n % 2 == 1) // cheating probably...  tmp += n + n - 1;  return tmp;  } |

**PROBLEM 3:** Below is a C++ function which is supposed to take an integer array a[] of length and create a “clone” of a (an array of the same length with the same contents) and return the clone.

This attempt is faulty!!!

|  |
| --- |
| int clone\_array(int a[], int n) {  int b[n];  int i;  for(i=0; i<n; i++) {  b[i] = a[i];  }  return b;  } |

**3.A:** identify and describe the errors in this attempt to the best of your ability. Hint: one of the issues relates to the return type (but this is not the only issue). Describe a scenario in which things might go haywire even if the return type issues is corrected.

|  |
| --- |
| **ANSWER** |
| **ISSUE 1:**  The return type is int meaning that a single integer is returned. But we want to return an array.  In C++, this means we want to return an int pointer (which is the base address of the array); return type should be **int \*** |
| **ISSUE 2:** The array b[] is local to the function and is allocated on the call stack. Then (even if we fix the return type), we return the base address of this stack-allocated array.  Bad news! g++ should give you a warning when you do this sort of thing. |

**3.B:** if this was an exam question worth 10 points, how much partial credit would you give if you were the grader?

|  |
| --- |
| **Anything reasonable is ok (certainly not 10!)**  **MY ANSWER:**  something in the 3-5 range? |

**3.C:**  Give a correct version!

|  |
| --- |
| **ANSWER** |
| // CORRECTED VERSION  int \* clone\_array(int a[], int n) {  int \*b;  int i;  b = new int[n]; // from the heap!  for(i=0; i<n; i++) {  b[i] = a[i];  }  return b;  } |
| // ANOTHER ***WRONG*** VERSION JUST FOR FUN…  // the author of this version is trying to be fancy and  // to use pointers to access array elements directly  // and pointer arithmetic to advance through the array  //  **// Do you see the goof?**  int \* clone\_array(int a[], int n) {  int \*b;  b = new int[n]; // from the heap!  while(n) {  \*b = \*a;  b++;  a++;  n--;  }  return b;  } |

**PROBLEM 4:** Consider the C++ function below:

|  |
| --- |
| void fubar(unsigned int n) {  int i, j;  for(i=0; i<n; i++){  cout <<"tick" << endl;  }  for(i=0; i<n; i++) {  for(j=0; j<n; j++) {  cout <<"tick" << endl;  }  }  } |

**4.A:** Complete the following table indicating how many “ticks” are printed for various parameters n.

Unenforceable rule: derive your answers “by hand” -- not simply by writing a program calling the function.

|  |  |
| --- | --- |
| **ANSWER** | |
| **n** | **number of ticks printed when fubar(n) is called** |
| 0 | 0 |
| 1 | 2 |
| 2 | 6 |
| 3 | 12 |
| 4 | 20 |

**4.B:** Derive a closed-form expressing the number of ticks as a function of n -- i.e., complete the following:

*“For all , calling fubar(n) results in \_\_\_\_\_\_\_\_\_\_\_\_\_ ticks being printed”*

Give a brief justification of your answer; you do not need a formal proof.

|  |
| --- |
| **ANSWER** |
| *“For all , calling fubar(n) results in ticks being printed”* |
| Why? Examining the code, the first loop executes exactly times producing one tick each iteration.  Then we have a nested loop; the outer loop executes times and each time through, the inner loop starts anew; each time the inner loop runs to completion, it produces ticks. So, the nested loops produce ticks.  This gives us the total of |

**PROBLEM 5:** Consider the recursive C function below:

|  |
| --- |
| void foo(unsigned int n) {  cout << "tick" << endl;  if(n > 0) {  foo(n-1);  foo(n-1);  }  } |

**5.A:** Complete the following table indicating how many “ticks” are printed for various parameters n.

Unenforceable rule: derive your answers “by hand” -- not simply by writing a program calling the function.

|  |  |  |
| --- | --- | --- |
| **ANSWER** | | |
| **n** | **number of ticks printed when foo(n) is called** | **comments** |
| 0 | 1 | if fails; no recursive calls |
| 1 | 1 + 1 + 1=3 | one initial tick followed by two calls of foo(0) each of which produces 1 tick (from row above) |
| 2 | 1+3+3=7 |  |
| 3 | 1+7+7=15 |  |
| 4 | 15+15+1=31 |  |

**5.B:** Derive a conjecture expressing the number of ticks as a function of n -- i.e., complete the following:

*“Conjecture: for all , calling foo(n) results in \_\_\_\_\_\_\_\_\_\_\_\_\_ ticks being printed”*

|  |
| --- |
| **ANSWER** |
| *Conjecture: for all , calling foo(n) results in 2n+1-1 ticks being printed* |

**5.C:** Prove your conjecture from part B (hint: Induction!)

|  |
| --- |
| **ANSWER** |
| Proof by induction.  BASIS:  [We must show that foo(0) results in ticks being printed]  Consider the behavior of foo(0). First, the print statement executes resulting in one tick printed.  Then the if-statement fails since n==0 and so no recursive calls are made and the function returns.  Thus: a total of 1 tick is printed.  Noting that , we conclude that the claim holds for n=0. |
| INDUCTIVE HYPOTHESIS:  Assume for ***some*** integer that foo(k) results in exactly ticks being printed. |

(continued on next page)

|  |
| --- |
| INDUCTIVE STEP:  We must prove that foo(k+1) results in exactly total ticks being printed. |
| PROOF OF INDUCTIVE STEP:  Let’s trace what happens when foo(k+1) is invoked (the parameter n=k+1).   1. First, the print statement is executed resulting in one tick 2. Since we know , we also know that . Thus, the if-statement succeeds and... 3. the first recursive call foo(n-1) is made. Recall that n=k+1 and so n-1 = k. So, by the inductive hypothesis, this first call results in exactly ticks being printed. 4. After the first call returns, the code immediately calls foo(n-1) again, resulting in *additional* ticks being printed (again by the inductive hypothesis).   Thus, in total we have:  tick from A  ticks from C by I.H.  ticks from D. by I.H.  or, total ticks printed must be:    (algebraic manipulation from here on)        Thus, we have completed the inductive step, completing the proof. |

**PROBLEM 6:** In the puzzle game sudoku we have a 9x9 grid which must be populated with integers in {1..9}. In a correct solution each row, column must contain each value in {1..9} exactly once (there are also 9 3x3 sub-grids that must obey the same rule).

We want a function which takes an integer array of length 9 representing a sudoku row and determines if it is "ok" or not according to the rule above; it should return true or false accordingly.

Below is an *attempt* at solving this problem.

|  |
| --- |
| // array row[] is assumed to be of length at least 9  bool sudoku\_row\_ok(int row[]) {  int sum=0;  int i;  for(i=0; i<9; i++) {  if(row[i] < 1 || row[i] > 9)  return false; // out of range  sum += row[i];  }  if(sum == 45) // notice: 1+2+3+4+5+6+7+8+9 = 45  return true;  else  return false;  } |

**2.A:** The above attempt is faulty! Give and briefly explain in your own words a counter-example showing that it is faulty.

|  |
| --- |
| **ANSWER:** While the following is indeed true  "If a sudoku row is legal then its sum is 45"  it's converse is not:  "if a sudoku row has a sum of 45, then it is legal" (NOT TRUE!!)  Equivalently, we can say that having a sum of 45 is a ***necessary*** condition for a row being legal, but it is not a ***sufficient*** condition.  Counter example: consider an array of nine 5's:  int test[] = {5,5,5,5,5,5,5,5};  sudoku\_row\_ok(test) will return true when clearly it should return false (you might call this a "false positive"). |

**2.B:**  Write a correct version of the function. You may not rearrange the elements in the given array. Your solution just has to be correct -- if it seems inefficient, don't worry about it (at least for the purposes of this homework).

|  |
| --- |
| There are many correct solutions to this problem.  Below are three... |
| // APPROACH 1  bool sudoku\_row\_ok(int row[]) {  int sum=0;  int i;  for(i=0; i<9; i++) {  if(row[i] < 1 || row[i] > 9)  return false; // out of range  }  // note: !has\_dups() is eq to "no duplicates"  return !has\_dups(row, 9); // from problem 1  } |
| // APPROACH 2  // first a simple helper function  // returns number of occurrences of x in array a[]  // n is array length  int num\_occ(int a[], int n, int x){  int num=0;  int i;  for(i=0; i<n; i++) {  if(a[i]==x) num++;  }  return num;  }  // idea: every integer in {1,2,...,9} must appear exactly once  // (and this is a sufficient condition).  bool sudoku\_row\_ok(int row[]) {  int x;  for(x=1; x<=9; x++) {  if(num\_occ(row, 9, x) != 1)  return false;  }  return true;  } |

|  |
| --- |
| // APPROACH 3  // idea: use a local array count[] where count[x] keeps track of  // how many x's we have seen so far. If count[x] ever becomes 2  // we have a failure  // Note: count[0] is not used.  bool sudoku\_row\_ok(int row[]) {  int i, x;  int count[10] = {0};  for(i=0; i<9; i++) {  x = row[i];  if(x<1 || x>9) // first make sure entry is in-range  return false;  count[x]++;  if(count[x] == 1) // check if this is 2nd occurrence of x  return false;  }  // claim: if we get here count[] must be {0,1,1,1,1,1,1,1,1,1}  return true;  } |

|  |
| --- |
| **DISCUSSION (not part of assigned homework)**: This problem is a special case of the "is-permutation" problem:  GIVEN: an array a[] of n integers.  TASK: determine if a[] is a permutation of {1,2,3,...,n}  (return true/false accordingly) |
| All three of the approaches above can be adapted to this more general formulation of the problem.  Can you do these adaptations (below is a template of a function for this more general formulation)?  bool is\_permutation(int a[], int n) {  }  NOW: perform best and worst-case runtime analysis for all three of these approaches to the is\_permutation problem (runtime expressed as a function of n). |