University of Nevada, Las Vegas Computer Science 477/677 Fall 2023

Assignment 6: Due Friday November 10, 2023, 11:59 PM

Name:____

You are permitted to work in groups, get help from others, read books, and use the internet. Turn the assignment in to Canvas, following the instructions given to you by Sabrina Wallace.

1. Solve these recurrences using the generalized Master theorem (Akra–Bazzi).

   (a) $F(n) = F(3n/5) + 4F(2n/5) + n^2$

   (b) $F(n) = F(n/5) + F(7n/10) + n$

   (c) $F(n) = F(n/2) + 2F(n/4) + n$

   (d) $F(n) = F(3n/5) + F(4n/5) + 1$

2. **Storing an Abstract Array as a 1-Dimensional Array:**

   Read this internet page:

   `https://www.prepbytes.com/blog/arrays/base-address-of-a-two-dimensional-array/`

   The discussion on that page presumes that the computer's random access memory (RAM) is a 1-dimensional array of *cells*, indexed by integers starting at 0, and that each cell consist of 4 bits of memory. That number could be different. Each addressable location could consist of 4, or 16, or 32, or whatever, bits. We write `RAM[i]` to be the i^{th} cell (actually, the $(i+1)^{\text{st}}$ because of the zero address.) An item to be stored in RAM could need any number of cells; that number is called *size* in that article. An array declared by your program would (normally) be stored as a contiguous block of cells starting at a *base adress*, which is chosen by the compiler. For example, if you declare the array int A[100], and each integer requires four cells, and the compiler chooses 1024 to be the base adress, 400 cells are allocated to store A. A[0] is stored at RAM[1024] ...RAM[1027], while A[i] has base address 1024+4i and is stored in RAM[1024+4i] ...RAM[1024+4i+3]. The number 4i is called the *offset* of A[i].

   Basic Rule: If a number of items are stored, the offset of any item is equal to the number of prdecessors of that item times the size of each item. To get the address in RAM, add the offset to the base address.

   Example: An array int A[5][3] is stored in row-major order, the base address is 2048, and once again, an integer uses 4 cells. The elements of A are stored in this order: A[0][0], A[0][1], A[0][2], A[1][0], ...A[4][1], A[4][2]. On the other hand, if they are in column-major order, their order in RAM is A[0][0], A[1][0], ...A[3][2], A[4][2].

   (a) In the example, if int A[5][3] is stored in RAM with offset 2048, what is the RAM address of A[3][1] if the storage is row-major? What if it is column-major?

   (b) If X[10][25][30] is stored with base address 8192, and each entry of the array requires 8 cells, what is the RAM address of X[8][11][15] if X is stored in row-major order? what if in column-major order?

3. **Sparse Arrays:**

   Crawley's Department Store hired a CS graduate to set up a system which could access any customer's complete record, containing all the information that Crawley's wants to save for that customer, by entering her social security number.

   The graduate (who slept late that day in CS477) started by defining a structured type called `record` and then declaring an array `record customer[1000000000]` because a social security number has nine digits and there are one billion strings of nine digits. But the number of customers that Crawley's has ever had is no more than twenty thousand.

   Instead, he should have stored the records in a *sparse array.* If Amanda Jones was a customer and had SSN x, then customer[x] will return her record, but if there is no customer with SSN y, then customer[y] will return a default value, such as zero, or perhaps the message "not found."

   The array `customer` is then a *sparse array.* There are a number of ways to implement sparse arrays, but my favorite is as a search structure of memos. A memo is an ordered pair (i,A[i]). In the department store example, where the memo consists of the social security number and the record of an actual customer. The structure is indexed by the SSN. Then the **fetch** command returns the customer's record if there is such a customer, otherwise a default. The **store** command either overwrites an existing record or creates a new memo.

4. **Memoization:**

Memos are stored as entries in a sparse array, which we can implement as a search structure, as described in Problem 3, except that, if there is no entry for a given index, that entry must be computed and then stored. I recommend that an ordinary (not balanced) binary search tree not be used, as it does not perform well in examples I have worked. Perhaps a treap is best, although I am not sure.

Consider the following recursive C++ function.

```
int F(int n)
 {
  if (n < 4) return 1;
  else return F((n+2)/2)+F((n+1)/2)+F(n/2)+F((n-1)/2)+n*n;
 }
```

(a) What is the asymptotic complexity of F(n)?

(b) What is the asymptotic time complexity of the calculation of F(n) using the recursive code above?

(c) What is the time complexity of the standard dynamic programming calculation of F(n)? (That means, compute and store F(1), F(2), F(3), F(4) ... F(n), in that order.)

(d) What is the time complexity of computation of F(n) using memoization? Assume that each arithmetic operation takes $O(1)$ time, and neglect the time it takes to insert or fetch a memo.

5. **A* Algorithm**

Walk throuth the A^* algorithm for the following weighted graph, finding the least cost path from S to T. The edge weights are in black and the heuristics are in red. The heuristics are both admissible and consistent. Your answer should label each fully processed vertex with both f and g values. Not all vertices will be processed.