Homework One: Complexity and ADTs

Duncan Parke Data Structures, SP25

Question One

> Show how non-negative numbers can be represented in an imaginary ternary computer using trits(1,2,3) instead of bits(0,1). Why don't we do things this way?

If we had a computer which uses trits, we would need to use a base-3 system of numerical representation. The value of a trits' place would be 3^{nth} power, where n is its position relative to the rightmost trit.

We don't do this because the computer hardware we run on operates by representing data as either the presence or lack of voltage. Thus, having to read a third state in between on and off would be difficult, making a trit system impractical.

Question Two

> If each element of an array RM, with 10 rows and 20 columns, stored in row major order, takes four bytes of space, where the first element of RM starts at 100, what is the address of RM[5][3] and RM[9][9]?

Assuming row major order, we'll have have the data structure stored such that rows are consecutive in memory. Thus, each row will contribute 4bytes * 20 columns, and then the column index will be an extra 4 bytes per offset.

Of note, I am assuming that the address we are given is in decimal, since I'm accustomed to seeing hex addresses appended with 0x notation. I'm also assuming that we are operating in a system which is byte addressable

So, the element RM[5][3] will be stored at location 100 + (5*80) + (4*3) = address 512

and the element RM[9][9] will be stored at location 100 + (9*80) + (9*4) = address 856

Question Three

> A lower triangular matrix is an nxn array in which has a[i][j] == 0 if i < j. What is the maximum number of non zero elements? How can they be stored in memory sequentially? Find a formula k = f(i,j) to store location a[i][j] in k (you only want to store the nonzero elements). Do not write code. Code would work if you were manually converting the matrix from one form to the other all at once, but you need the formula to convert otherwise

1. The maximum number of non-zero elements is the sum of n - i, for all i >= 0 and i < n. This can be more efficiently written as n(n+1)/2
2. They can be stored sequentially by flattening out the array so that we have a header which points to each base for a given row, but that we only store the non-zero values at that location. Thus, we can index through it using pointer logic but still maintain the structure of the array 3.To store something at element a[i][j], you'll need a formula which returns the location based on the maximum number of

elements that are able to be stored in the array at that location plus the offset. Thus, we'll have a k = c + sizeof(arr_element) ((i-1)i)/2 + sizeof(arr_element) * j, where c is the starting location in memory (assumed const) of the array. This will give you the location in memory by leveraging the number of total memory locations used by the array up to index i.

Question Four

> A tridiagonal matrix is an nxn array in which has a[i][j] = = 0 if |i-j| > 1. What is the maximum number of non zero elements? How can they be stored in memory sequentially? Find a formula k= f(i,j) to store location a[i][j] in k, when |i-j| <=1 (you only want to store the nonzero elements). Do not write code. Code would work if you were manually converting the matrix from one form to the other all at once, but you need the formula to convert otherwise

1. The maximum number of non-zero elements in the array are those which are on the diagonal, on the diagonal starting at (0,1) and the diagonal starting at (1,0). Thus, we'll have an diagonal of size n, a diagonal of size n-1, and another diagonal of size n-1, meaning the total number of non-zero elements will be n + (n-1) + (n-1), or 3n - 2.
2. They can be stored in memory by storing each diagonal separately, so the main diagonal, then the super, then the sub.
3. To access the memory from k, you'll need to access the offset by determining which diagonal you are trying to access. If i == j, then you'll be accessing the main diagonal. If j = i - 1, you're accessing the lower diagonal. If j = i + 1, the super. So, for i = j, then the access will simply be base + i. For i = j + 1, we'll need n + i. And finally, for i = j - 1, we'll need n + n - 1 + i, or 2n + i - 1. To collapse this into one function k, we'll need to use indicator functions which resolve to one given a case. Ergo, for any case where j != i, we'll need to add in n.

so, we'll always need the base indexer of i. Then, we can contribute n if i != j, and then contribute an additional n - 1 if i = j -1. So, we have:

k = i + |i - j|n + (n-1)(j - i). I think we could also resolve the abs by making j-i contribute further, but given that this isn't code I'll leave it with abs.

Question Five

> Consider two functions f(n)=an2, and g(n)=bnlgn. For what value of n do they intersect, and at which value(s) of the constants a and b? Pick some values of a and b and then find n. Experiment with different sets of values for a and b. What trends do you observe? an2 and bnlgn are terms that might come from an expression representing the amount of work done by a piece of code

1. To find the points of intersection between the two functions, it is necessary to set the functions equal to each other and then solve for the points of equality. There will be an obvious case at n = 0, since both are dependent on n being multiplied. But, excluding that case, we can divide both sides by n
2. an^2 = bnlgn // simplfy by n
3. an = blgn

Thus, we have a solution where the pieces will intersect at any point where a * n is equal to b * lg(n), assuming n is non-zero and positive.

Playing with this on a graphing calculator, we can see that they will only intersect at 0 for cases where the ratio between a and b is not somewhere in the neighborhood of 51/8, b/a.

Question Six

> What is the maximum size of a problem that can be solved in one hour if the algorithm takes lg n microseconds

To answer this question, we'll need to find some n for which (assuming lg == log base 10) the $\log_{10} n$ = 3,600,000,000us, where 3,600,000,000us is the number of us in an hour.

To find this, we'll need to apply the log identity: $\log_b n = m : b^m = n$. Thus, $n = 10^{3.6billion}$

Question Seven

> What is the maximum size of a problem that can be solved in one hour if the algorithm takes n^3 microseconds?

To answer this question, we will need to solve the equation for n, given $n = 3600^{1/3} s/s = 3{,}600{,}000{,}000^{1/3}$ us/us = 1532.61, which will round down to 1532 as a maximum size.

Thus, the maximum size of a problem is 1532.