# Homework 2

Due: Feb 9, 2013 2:00 PM (early)
Feb 11, 2013 11:59 PM (on time)
Feb 13, 2013 11:59 PM (late)

The written portion of the homework must be handed in to the CS157 hand-in bin located on the CIT 2nd floor between the Fishbowl and the locker. The programming portion of homeworks (when applicable) must be handed in electronically via the hand-in script.

**Each problem must be handed in separately** with your name on the top and the time of hand-in. You may turn in different problems for different deadlines. For more information, please review the assignment and due date policy in the course missive on the course webpage.

Please ensure that your solutions are complete and communicated clearly: use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

You are allowed, and often encouraged, to use Wikipedia to help with your homework. See the collaboration policy on the course webpage for more details.

## Problem 1

Many algorithms (including many of the dynamic programming algorithms we have seen) involve reading through all the elements of an $n \times n$ array in order, and perhaps doing some local operations along the way. There are two common ways to order a 2-dimensional array: *row-major order*, and *column-major* order. In row-major order, first you go through the elements of the first row in order, then you go through the elements of the second row in order, etc., traversing each row after the previous one. Alternatively, in column-major order, you first go down the first column, and then go through each subsequent column in turn. These two orders are also the two main options for how to *store* arrays in memory: assuming we store an $n \times n$ array as a contiguous block of memory, do we store the array in row-major order or column-major order? Explicitly, element $(i, j)$ of the array, where $i$ is the row and $j$ is the column, indexed from 0, is stored in memory location $i + n \cdot j$ in column-major order, but in location $n \cdot i + j$ in row-major order. According to Wikipedia, C and Python store arrays in row-major order, while Matlab and Fortran store arrays in column-major order.

When an algorithm traverses an array, it is important to know if it is traversing the array in the *same* order that the programming language stores the array in, or the *other* order (that is, if your algorithm traverses the array in row-major order, your algorithm might have different performance depending on whether your programming language stores the array in row-major order, versus column-major order).

In this problem, we will compare the *cache* behavior of traversing in the same order, versus the other order. Model the cache as follows: the cache is of total size $c$ (perhaps $2^{16}$), consisting of *cache lines* each of size $\ell$ (perhaps $2^6$); each time your code requests access to a memory address, if it is not in cache, the aligned block of memory of size $\ell$ containing the requested location is copied to the cache in time $t$, and the *least recently used* cache line is evicted from cache. ("Aligned" here means that the start of each cache line will be a memory address that is a multiple of $\ell$; if $\ell = 64$

then locations $0 \ldots 63$ are a valid cache line, as are locations $64 \ldots 127$, but not $30 \ldots 93$.)

1. How much time does it take to read an $n \times n$ array in the *same* order as it is stored in, as a function of $n, c, \ell$, and $t$? Justify your answer, though feel free to assume that $\ell$ divides $n$.
2. How much time does it take to read an $n \times n$ array in the *other* order from which it is stored, as a function of $n, c, \ell$, and $t$? Justify your answer, though feel free to assume that $\ell$ divides $n$.
3. Plot the answers from the previous two parts as a function of $n$ (by hand is fine, as long as the plot is clear and labeled), taking $c = 2^{16}, \ell = 2^6$, and $t = 1$ (arbitrary time units). You should pick the range of $n$ to be most communicative.
4. In light of the previous graph, describe qualitatively how, in practice, you might notice that your code is running into cache issues.

## Problem 2

Matlab has the convenient feature of letting you write "off the end" of arrays. That is, if `myarray` is an array of size 10 (indexed in Matlab by indices 1 through 10), then you are allowed to set the 11th entry of this array of size 10, as in `myarray(11)=1234;` or the 300th entry of this array, as in `myarray(300)=111;`

How does Matlab manage this internally? Until recently, it had the following straightforward algorithm: each array corresponds to a contiguous block of memory of exactly the right size; every time the user wants to modify an element of the array, if the element is "inside" the array, do it; otherwise, allocate a new contiguous block of memory that is exactly the right size to hold the new expanded array, copy the old array to its new memory space, set the new parts of the array to be 0, and then implement the assignment the user requested; finally free the memory that is storing the old version of the array and update the Matlab variable to point to the new version of the array.

1. Given an array of size $n$, how many operations can a single assignment require, representing your answer in big-O notation? (You can assume that allocating and freeing memory can be done in logarithmic time.)
2. Given an algorithm that involves at most $n$ assignments to positions $\leq n$ in an array, how many operations might Matlab require to execute the code, representing your answer in big-O notation?
3. The *preallocation* strategy for writing Matlab code involves figuring out the final size of your array before you make any assignments, and then initializing your array to be that size. This involves foresight, and may or may not be possible, depending on how complicated your code making the assignments is. Nonetheless, this is clearly the *optimal* (but possibly unattainable) strategy, OPT: allocate once, and then perform each assignment in constant time. What is the competitive ratio of Matlab's algorithm to this optimal (with foresight) algorithm? What is the competitive ratio of memory usage?
4. How might we improve Matlab's performance? (Stop and think here for a bit.) Try the following strategy: every time we are forced to expand an array, instead of expanding it only as much as is necessary, we now round up to the nearest power of 2 size. Write pseudocode for this strategy.

5. Analyze your pseudocode: What is its competitive ratio with OPT? How does its memory use compare with OPT? How does its memory use compare with the original Matlab implementation?
6. Instead of rounding up to the nearest power of 2, repeat the calculations of the previous problem assuming we round up to the nearest power of $c$, for some constant $c > 1$. What value of $c$ would you pick?
7. Current versions of Matlab seem to use the following strategy when expanding arrays: round up to the nearest power of 2, or round up to the next multiple of 1 gigabyte, whichever is nearer. Explain, in the language of competitive analysis, and informed by what you know about computers+programmers, why this might be a reasonable strategy.

## Problem 3

On the previous problem set you wrote code in Java to compute the edit distance between two strings. Most of you used a two-dimensional array to store, for pairs $(i, j)$, the edit distance between the first $i$ characters of the first string and the first $j$ characters of the second, and then traversed the array row-by-row from top to bottom. In this problem we will explore a few simple tweaks to this code. In this problem, to make calculations easier, assume both strings have the same length $n$.

```
EditDistanceSimple($s1, s2, i, j$)
    Create 2D array $table$[LENGTH($s1$) + 1][LENGTH($s2 + 1$)]
    Set $table(i, 0) \leftarrow i$ and $table(0, j) \leftarrow j$ for all valid $i, j$
    for $i \leftarrow 1$ to LENGTH($s1$)
        do for $j \leftarrow 1$ to LENGTH($s2$)
            do if $s1(i) = s2(j)$
                then $table(i, j) \leftarrow table(i-1, j-1)$
                else $table(i, j) \leftarrow 1 + \min(table(i-1, j), table(i-1, j-1), table(i, j-1)$
```

1. The amount of memory used by the standard dynamic programming approach to edit distance is $O(n^2)$, which is the same order as the amount of time the algorithm takes; while we might be happy waiting for several billion CPU cycles, our code might crash on current hardware if it also demands several billion memory locations. Fortunately, we do not need to store the whole two-dimensional table: it is enough to store just the current row being computed, and the previous row that we have just finished computing. Write pseudocode for this strategy.
2. Describe, in a few sentences, and using concepts from lecture and/or your knowledge of systems why and how this approach is an improvement.
3. While you have now reduced the memory usage to just two arrays of size roughly $n$, processing a single row is essentially unchanged: there is an `if` statement comparing letters of two strings, which if false leads to a memory copy from location $(i-1, j-1)$ to location $(i, j)$, and if true leads to a minimum of 3 things being computed; processing a single row touches roughly $2n$ memory locations. There is a way to improve both of these aspects: your challenge in this problem is to find a way to lay out the entries of the table in memory so that when the two letters are the same, instead of doing a memory copy, the algorithm *does nothing.* To be more explicit: write pseudocode for the edit distance problem where the *same* memory address is used for location $(i-1, j-1)$ *and* for location $(i, j)$.

4. Describe in a few sentences how your algorithm works, and why it is correct.

5. In all of the algorithmic variants of edit distance we have seen so far, there is a computationally expensive step of computing a 3-way minimum of memory locations that represent $(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$. Programming languages typically have a built-in function to compute a 2-way minimum, instead of a 3-way minimum, so to compute a 3-way minimum, you must first compute the minimum of a pair of entries, and then take the minimum of that result and the third item. This leads to *three* different options for how to compute the 3-way minimum, depending on which pair of entries is considered first. The brainteaser is: which of these three ways is best, and why? (If you do not get this problem, skip it and move on: brainteasers have a very simple punchline that is hard to guess or derive. If you want a hint, you can try timing your own Java code; the speed difference seems to be about 2 clock cycles per 3-way minimum between the "good" order and the others.)

6. In all of the above variants of the edit distance algorithm, we are computing the same entries of the same table, in the same order—just storing them differently. However, there is a different order of computing these entries that may have some advantages: diagonal. We start at $(0, 0)$, and in round $r$, for $r \leq n$, compute the entries along the diagonal going from $(r, 0)$ to $(0, r)$; after $n$ rounds of this, we next compute the entries in the diagonal from $(n, 1)$ to $(1, n)$, and so on. Write pseudocode for this and explain why it works.

7. This order of traversing the 2-dimensional table gets rid of *data dependencies*: every time we compute something, the result does not need to be used until the next round. One advantage of this is that it lets us parallelize the code. In a diagonal of length $\ell$, each of the $\ell$ entries can be computed independently, perhaps even by $\ell$ different processors or cores. However, launching so many threads has a cost. In this problem, consider the following very simple model: you have $\ell$ units of work to do, and as many processors as you want. Each unit of work takes 1 time on 1 processor, which might lead us to expect that we can finish all the work (which corresponds to 1 diagonal) in 1 unit of time; however, launching and synchronizing processors is expensive. Assume that launching $k$ new threads/processors takes $1000k$ units of time. Plot the rate of getting work done as a function of $k$, for a workload of $1,000,000$ units—rate equals a million divided by the time it takes to finish all the work (plotting by hand is fine, as long as the plot is clear and labeled). What is the best number of processors to use?

8. In the model of the previous part, what is the general formula for the best number of processors to use, for a workload of $w$ units, if launching each new thread/processor takes $p$ times more time than doing a unit of work? (Feel free to ignore issues of rounding.)

## Problem 4

Consider the task of traversing a search tree to find a desired key. Most search trees you have seen are probably *binary* search trees, where every internal node has two children, but in this problem we will consider *k-ary search trees* for integers $k \geq 2$.

In a binary search tree each node has one value and two children, and each of the values in the left subtree of a node is less than the value in the node while each of the values in the right subtree is greater than the value in the node. You can think of each node as containing 3 things in order: a left subtree, a value, and a right subtree:

subtree | key | subtree

Each of the elements recursively contained in this node are in order: everything in the left subtree is smaller than the key which is smaller than everything in the right subtree. This pattern is generalized in $k$-ary search trees: each node has $k-1$ values stored in an array in the node, and $k$ children. For $k=4$ this could be depicted as:

subtree | key | subtree | key | subtree | key | subtree

The values in each of the subtrees respect the ordering here, in that, say, each value of the 3rd subtree of a node is between the values of the second and third keys in the node.

To search for a key in a $k$-ary search tree, we start at the root and iteratively descend the tree, where for each node we visit we find which subtree to descend down via a linear search over the keys in the node. (Recall from lecture why linear search is perhaps better than binary search here.)

In this problem, to simplify the analysis, we will assume that each level of the tree is completely filled: if the tree has height $h$, then the levels are indexed from 0 through $h$, and the $i$th level will have $k^i$ nodes, and $(k-1)\cdot k^i$ values stored in arrays at that level.

1. Write pseudocode for looking up the location of a key in a $k$-ary search tree. (Remember, the keys in each node are stored in an array of size $k-1$. The location of the subtrees does not need to be stored because you can put then in some standard location, so in your pseudocode feel free to write something like "descend to the $i$th subtree", and in your analysis in the following parts, you do not need to consider the time of this operation.)

2. Given a search tree with $n$ nodes, what is its depth? (Remember to justify your answer. Feel free to ignore issues of rounding.)

3. Next, we must figure out how long it takes to traverse each level of the tree, on average. In this problem, we will focus on caching: assume that to do an operation on a key in one of the arrays, the memory storing the key must first be loaded into cache; assume that cache is loaded in chunks (cache lines) of size $c$, and that each such load takes $t$ time. To simplify things, assume that the start of each node's $k-1$ element array is aligned with a cache block, and assume that the cache only stores 1 cache line at a time. In short, consider the memory as being divided into blocks of size $c$, and each time we touch a new block, we pay a cost of $t$. On average, how long does a linear search for a random element in a list of length $k-1$ take, under these rules? (Searching for a random element of a $k$-ary tree is not *exactly* the same as searching for a random entry among the $k-1$ at each level, but feel free to make simplifying assumptions along these lines.)

4. Taking the product of your results in the last two parts of this problem should give you an estimate for how long a single lookup will take in your $k$-ary tree. Now we must decide: what value of $k$ is best? Plot the time you have computed as a function of $k$, assuming $n=10^6, c=16$, and $t=20$ (by hand is fine, as long as the plot is clear and labeled).

5. Write a few sentences about what this tells you about how to write code to traverse trees; perhaps plot your answer to the previous problem with *different* parameters to explain a general idea.