# Homework 3: Cache For Gold

Due: Oct. 2, 2018 at 6:00 PM (early)
Oct. 5, 2018 at 6:00 PM (on time)
Oct. 7, 2018 at 6:00 PM (late)

This is a partner homework as usual: work on all problems together; come to all office hours together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. **If you don't have a partner, please let us know.**

Problem 4 involves writing Java code, which must be handed in electronically using the hand-in script. Run `cs157_handin hw3` to copy everything from your current directory to our grading system. Starter code for these problems is in the directory `/course/cs157/pub/stencils/` Please **do not** rename the files/functions when you hand them in, or else your TAs will be sad. Make sure your functions return their results by assigning to the designated output variable and not by printing to the screen. Code will be graded automatically so there will be no style grades, and partial credit may not be given for code that does not work. Other parts of these problems involve writing explanations in complete sentences that must be handed in separately on paper to the hand-in bin as usual.

**1-credit track:** do problems 1, 3, and 5 *and indicate "***1-credit***" on each problem* so we can give 1-credit-specific feedback.

## Problem 1

> **Our Mission Statement: Got Gold? Need Cache? Best-in-Class Solutions!**
> We believe our distributed blockchain solution, which enables individuals to liquidate their hard-earned gold, will change the future of the cache-to-gold industry. By using cutting-edge ephemeral caching algorithms, we offer state-of-the-art service for everyone's cache needs.

(25 points total)

Recall the analysis from class comparing the "least recently used" (LRU) cache policy to the optimal offline policy. "Offline" in this sense means the algorithm can see all memory requests—past and future!—before deciding how to respond to each, and is thus unrealistic in a computational sense, though corresponds to our notion of what a "wizard" would do. By comparison, an *online* algorithm must respond to each request before the next request occurs. All realistic cache algorithms are online.

One can prove (using techniques you will see in the upcoming "greedy algorithms" section of this course) that an optimal offline policy is that every time a new entry is loaded to a full cache: evict from cache the entry that will first be used *farthest into the future*. We will refer to this policy as "OPT". For this assignment, you can assume this is an optimal policy.

1. (4 points) Consider three different cache policies: 1) OPT, as described above; 2) LRU, which always evicts from cache the memory location least recently used; and 3) FIFO, the "first in, first out" rule, which evicts the item that *entered* the cache earliest. Describe the sequence of

evictions for each of these three algorithms on a cache of size 4 for the following sequence of memory accesses: $1, 2, 3, 4, 5, 1, 3, 6, 2, 1, 4, 7, 7, 4, 5, 6, 3, 4, 1$, where the cache starts out empty.

2. (4 points) Show that on a cache of size $k$, the FIFO scheme can have $k$ times more cache misses than the LRU scheme. (**Hint:** Fill up the cache initially with $k$ elements 1 through $k$, and then find a sequence of memory accesses on $k+1$ elements where the LRU scheme has only 1 cache miss while the FIFO scheme has $k$.)

3. (4 points) Show the reverse of the above, that on a cache of size $k$, the LRU scheme can have $k$ times more cache misses than the FIFO scheme. (**Hint:** See above hint.)

4. (3 points) Show that the OPT scheme with a cache of size $k$ can be arbitrarily worse than the OPT scheme with a cache of size $k+1$. In particular, find an infinite sequence of memory accesses that requires 0 cache misses with size $k+1$ cache, but an arbitrarily large number of cache misses with size $k$ cache. (While this is a lot of cache misses, as you will show in the next problem, these cache misses are infrequent—at most one out of every $k$ requests will generate a cache miss, as it turns out.)

5. We now analyze something slightly more tricky, where you cannot just pick a single sequence of memory accesses to prove your result.

   (a) (4 points) Consider the OPT scheme, when the cache initially holds memory locations $1...k$, and is presented a sequence of $k$ memory accesses that are each between 1 and $k+1$. Show that OPT has *at most one cache miss*, total, in this situation.

   (b) (4 points) Show that for all non-random *online* cache algorithms $A$ (online means, in particular, that $A$ is not OPT), and arbitrary initial contents of a size $k$ cache, *there exists* a sequence of $k$ memory accesses to locations between 1 and $k+1$ such that $A$ has a cache miss *on every one of the $k$ accesses*. (**Hint:** Describe how to choose such a sequence of requests assuming you could *simulate* the behavior of algorithm $A$.)

   (c) (2 points) Conclude that, for caches of size $k$, the competitive ratio between *any* non-random online cache algorithm and OPT is at least $k$. (**Note:** The word "conclude" means that you have already done the hard part of this problem, and now you just need to very clearly write down the definition of "for caches of size $k$, the competitive ratio between *any* non-random online cache algorithm and OPT is at least $k$", and assemble pieces from the previous parts to prove that this definition is satisfied.)

**Comment:** As you have just shown, caches can perform drastically better or worse than other caches that are only slightly different. It might seem that we should just throw up our hands and give up on designing a reasonable cache. However, recall the very clean result from class that showed the simple LRU cache is 2-competitive with OPT (and hence with *any* other cache strategy!), as long as we make the cache twice as large. This simple change of perspective is what competitive analysis is very good at providing.

## Problem 2

**Who We Are: A Qualified Team**
We are a passionate group of gold investors, with a fundamental belief in the power of asset diversification via cache acquisition. We have decades of experience in obtaining gold, from candelabras to wedding rings, and we have leveraged that expertise to provide best-in-class rates to our customers. We have used our decades of combined experience to provide two optimized forms of cache, which offer solutions tuned to different performance problems.

(20 points total)
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 (check it out!), 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. (6 points) 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. (7 points) 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. (4 points) 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. (3 points) In light of the previous graph, describe in simple and general terms how, in practice, you might notice that your code is running into cache issues.

## Problem 3

> **Marketing Strategy: Arrays not big enough? Get More 4 Cheap!**
> Ever find yourself running out of space right before finishing that project? Don't skimp—array sizes available in the nearest power of two!

(30 points total)

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. You may ignore the run-time of memory allocation and freeing as operating systems generally do this very quickly. For parts that ask you to calculate a competitive ratio that is a constant, assume that writing an element to an array takes one unit of time, reading an element from an array takes one unit of time, so copying an element from an array to another array takes two units of time.

The following questions may sound oddly specific; however, they build components (including inequalities) that can be precisely used to solve later parts. Talk to TAs if you have any questions about how the parts fit together.

1. (4 points) Given an array $A$ of size $\leq n$ in the original Matlab implementation, what is the longest amount of time it might take to make the assignment `A(i)=17` for some $i \leq n$, expressed in big-$O$ notation in terms of $n$? You should give a specific example, analyze its cost, and then show that *every* other example has at most the same big-$O$ cost.

2. (4 points) Given an array of arbitrary initial size in the original Matlab implementation, what is the "worst-case" performance of making $n$ assignments to positions $\leq n$ in the array? Represent your answer in big-$O$ notation. As in part 1, both analyze a specific example, and also argue via inequalities why this is the worst case. (**Hint:** use your upper bound from part 1.)

3. (4 points) 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 the zeros array of that size before running the rest of your code. This involves foresight, and may or may not be possible, depending on how complicated your code is. Nonetheless, this is clearly the *optimal* (but possibly unattainable) strategy: allocate the memory once, and then perform each assignment in constant time. What is the competitive ratio of the original Matlab algorithm to this optimal (with foresight) algorithm, expressed in big-$O$ notation in terms of $n$? (**Hint:** For $k$ memory accesses made to locations $\leq n$, what is the cost of $OPT$? Using the previous parts, describe an example that performs much worse than $OPT$ in the

original Matlab implementation. And then slightly generalize previous parts to prove an upper bound, for any $k, n$, of the cost of the original Matlab implementation which lets you show that in all cases the ratio of Matlab to OPT is no worse than your example.)

4. (2 points) What is the competitive ratio of memory usage? (Remember, "memory usage" means the highest amount of memory used at any instant, including at any instant in the array–expanding procedure.) Your answer should be a specific constant. Justify your constant from both directions, as in previous parts.

5. How might we improve Matlab's performance? Stop and think here for a bit. Consider 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.

   (a) (5 points) What is the competitive ratio of memory usage between this strategy and the optimal strategy from part 3? (**Hint:** Consider running code that ends with an array of size $n$. The competitive ratio is the largest possible ratio between the memory usage of this power of 2 strategy versus the memory usage of the optimal strategy, which is just $n$. Be sure to consider cases when $n$ is *not* a power of 2!)

   (b) (8 points) What is the competitive ratio of computation time between this strategy and the optimal strategy from part 3? (**Hint:** Determine the maximum amount of time spent in the array–expanding code, given that the final array size is $n$. Then show that there is a short sequence of, e.g., $O(\log n)$ assignments that demands this much time. Compute the ratio of this time to the time for the optimal algorithm, namely $n + O(\log n)$—counting time for initialization and then each of the $O(\log n)$ assignments. For large $n$, this competitive ratio should converge to a constant.)

6. (3 points) 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 (referring to what you learned in parts 3, 4, 5a, and 5b), and informed by what you know about computers and programmers, why this might be a reasonable strategy.

## Problem 4

**Product Quality: Boost your memory with algorithms procured from a local source, only a minimal edit distance away!**
Our cage-free, organically fed, open source memory solutions are sure to fit all your allocation needs, in a jiffy! We have a strong commitment to sourcing our cache algorithms only from public GitHub repositories where the last commit was made at least five years ago, and have been certified by the Better JavaScript Consortium for our sustainable loop unrolling and promise chaining.

(21 points)

As a reminder, starter code for this problem is in the directory `/course/cs157/pub/stencils/`. **Be sure to use this stencil, or else your TA's will be sad**. Furthermore, please do not print anything to standard output in the body of any of the `editDistance()` methods.

A straight-forward implementation of the edit distance algorithm would use 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 traverse the array row-by-row from lower to higher indices. In this problem we will explore a few simple tweaks to this implementation. To make calculations easier, assume both strings have the same length $n$.

EDITDISTANCESIMPLE($s1, s2$)

```
1  Create 2D array table(0...length(s1), 0...length(s2))
2  Set table(i, 0) = i and table(0, j) = j for all valid i, j
3  for i = 1 to length(s1)
4      for j = 1 to length(s2)
5          if s1(i) == s2(j)
6              table(i, j) = table(i − 1, j − 1)
7          else table(i, j) = 1 + min(table(i − 1, j), table(i − 1, j − 1), table(i, j − 1))
```

1. (3 points) 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. Implement this strategy in the method EDITDISTANCE1($s1, s2$) using Java.

2. (2 points) Describe, in a few sentences, and using concepts from lecture or your knowledge of systems why and how this approach is an improvement.

3. (3 points) 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, of which the true case (line 6) leads to a memory copy from location $(i-1, j-1)$ to location $(i, j)$, and the false case (line 7) 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*, i.e., the *same* memory address is used for location $(i-1, j-1)$ and for location $(i, j)$. Write a method EDITDISTANCE3($s1, s2$) achieving this in Java. (**Note:** Avoid computing a division or a mod in your inner loop, as these instructions can take 20+ cycles.)

4. (3 points) Give a high level explanation of how your code works.

5. (2 points) 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?

   **Hint:** Try timing your own Java code **on Sunlab machines**; the speed difference seems to be about two clock cycles per 3-way minimum between the "good" order and the others,

in the test case where one string is all A's and the other string is all B's. (We recommend timing on desktop/Sunlab machines because battery optimizations on many laptops make code timing inconsistent.)

6. (3 points) 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 each $r$ from 1 to $n$, 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 a method EditDistance6($s1, s2$) in Java implementing this idea.

7. (3 points) 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 we work on the next diagonal. One advantage of this is that it lets us parallelize the code (think about what a parallel version of the previous part would look like!). 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.

   For this problem, consider the following very simple model: you have a total of $n$ tasks to do, and as many processors as you want. Each task takes a unit of time on a processor, which might lead us to expect that we can finish all the tasks in a unit of time if we choose to have $n$ processors; however, launching and synchronizing processors is expensive. Assume that launching $k$ new processors takes $1000k$ units of time. Plot the time of getting work done as a function of $k$, for a workload of $n = 1\,000\,000$ tasks. Plotting by hand is fine, as long as the plot is clear and labeled. What is the best number of processors to use?

8. (2 points) In the model of the previous part, if launching each new processor takes $p$ units of time, what is the best number of processors to use, in terms of $n$ and $p$? Feel free to ignore the issues of rounding. For the optimal number of processors, how does the cost of launching new processors compare to the cost of doing the work? This reflects an interesting general heuristic.

## Problem 5

**Get In Touch**
We are always looking to hear from you, our cache-loving, gold-hating customers. We seek to change the world, one cache line at a time, but we can only do that with your help, and so we ask that you help us help you. Let us know what cache types your portfolio is missing, what excess gold you have in your attic, and what we, your loyal cache-for-gold outlet, can do to earn more of your business.

(4 points)

Each of the problems in this homework is included for a reason, and here is your chance to reflect on these reasons and gain a deeper perspective on how the topics of this week fit into the course.

For each of the four previous problems (two for 1-credit track), write a few sentences describing why the problem is interesting—explain, perhaps, how the problem is an example of some general principles, and/or how you might imagine using the principles of the problem in later life. Try to describe the "punchline" of each problem, as it relates to you.