Online Caches

1. Given the below sequence of memory accesses:

   ```
   {1,2,3,4,5,1,3,6,2,1,4,7,7,4,5,6,3,4,1}
   ```

   The caches, using different eviction policies, will vary accordingly. For a cache following a **FIFO** algorithm we'll have:

Last Incoming Item	Updated Cache	Evicted Item
1	[1]	None
2	[1, 2]	None
3	[1, 2, 3]	None
4	[1, 2, 3, 4]	None
5	[2, 3, 4, 5]	1
1	[3, 4, 5, 1]	2
3	[3, 4, 5, 1]	None
6	[4, 5, 1, 6]	3
2	[5, 1, 6, 2]	4
1	[5, 1, 6, 2]	None
4	[1, 6, 2, 4]	5
7	[6, 2, 4, 7]	1
7	[6, 2, 4, 7]	None
4	[6, 2, 4, 7]	None
5	[2, 4, 7, 5]	6
6	[4, 7, 5, 6]	2
3	[7, 5, 6, 3]	4
4	[5, 6, 3, 4]	7
1	[6, 3, 4, 1]	5

A cache utilizing a **LRU** algorithm, where we visualize the least recently used element as the one farthest to the left and the most recently used element farthest to the right, will look something like this:

Last Incoming Item	Updated Cache	Evicted Item
1	[1]	None
2	[1, 2]	None
3	[1, 2, 3]	None
4	[1, 2, 3, 4]	None
5	[2, 3, 4, 5]	1
1	[3, 4, 5, 1]	2
3	[4, 5, 1, 3]	None
6	[5, 1, 3, 6]	4
2	[1, 3, 6, 2]	5
1	[3, 6, 2, 1]	None
4	[6, 2, 1, 4]	3
7	[2, 1, 4, 7]	6
7	[2, 1, 4, 7]	None
4	[2, 1, 7, 4]	None
5	[1, 7, 4, 5]	2
6	[7, 4, 5, 6]	1
3	[4, 5, 6, 3]	7
4	[5, 6, 3, 4]	None
1	[6, 3, 4, 1]	5

A cache using an **OPT** algorithm, i.e. one able to predict future memory accesses will look something like this:

Last Incoming Item	Updated Cache	Evicted Item
1	[1]	None
2	[1, 2]	None
3	[1, 2, 3]	None
4	[1, 2, 3, 4]	None
5	[1, 2, 3, 5]	4
1	[1, 2, 3, 5]	None
3	[1, 2, 3, 5]	None
6	[1, 2, 6, 5]	3
2	[1, 2, 6, 5]	None
1	[1, 2, 6, 5]	None
4	[1, 4, 6, 5]	2
7	[7, 4, 6, 5]	1
7	[7, 4, 6, 5]	None
4	[7, 4, 6, 5]	None
5	[7, 4, 6, 5]	None
6	[7, 4, 6, 5]	None
3	[7, 4, 6, 3]	5
4	[7, 4, 6, 3]	None
1	[7, 4, 1, 3]	6

2. Given a cache of elements: [1, ..., k] let's first access elements in reverse order with a memory access sequence of:

$$\{k, ..., 1\}$$

Using a LRU eviction policy this'll produce a cache where, if we again visualize the least recently used element as the one farthest to the left and the most recently used element farthest to the right, will look like:

LRU: [k, ..., 1]

By contrast, the above sequence of memory accesses won't change a cache using FIFO since all the elements are already in the cache:

FIFO: [1, ..., k]

Now, let's say we then have a sequence of k memory accesses like:

$$\{k+1, 1, 2, ..., k-1\}$$

The element to be evicted in the LRU cache is k, while in the FIFO cache it's 1. Since we then proceed to access elements $\{1, ..., k-1\}$ and this is a subset of elements in the LRU cache, our LRU cache will only have a single miss. In our FIFO cache, however, on each access we're trying to reference the element we just evicted; every access in the FIFO cache will be a miss. Thus, we'll have a single cache miss in the LRU cache, but k misses in the FIFO cache.

3. Given a cache of elements: [1, ..., k] let's again first access elements in reverse order i.e. $\{k, ..., 1\}$. This gives us LRU and FIFO caches that might be visualized as below, where the leftmost element is the next element to be evicted:

LRU: [k, ..., 1]
FIFO: [1,..., k]

If we then have a sequence of k memory accesses like:

$$\{k+1, k, k-1, ..., 3, 2\}$$

This gives us a situation symmetric to part 2. The element to be evicted in the LRU cache is k, while in the FIFO cache it's 1. Since we then proceed to access elements $\{2, ..., k\}$ and this is a subset of elements in the FIFO cache, our FIFO cache will only have a single miss. In our LRU cache, however, on each access we're trying to reference the element we just evicted; every access in the LRU cache will be a miss. Thus, we'll have a single cache miss in the FIFO cache, but k misses in the LRU cache.

4. Let's say we have two OPT caches:

$$\textbf{OPT1}: [1, 2, ..., k]$$
$$\textbf{OPT2}: [1, 2, ..., k, k+1]$$

If we have an infinite sequence of memory accesses like:

$$\{k+1, k, ..., 2, 1, 2, ..., k, k+1, k, ..., 2, 1, ...\}$$

Since each memory access in the above sequence is already in OPT2, OPT2 will have zero cache misses. Since the size of OPT1 is less than the number of unique memory accesses we're making, then by the pigeonhole principle there must be a cache miss in OPT1. Every time we make k memory accesses there'll be a miss in OPT1. Thus, for an infinite sequence of memory accesses OPT1 can be arbitrarily worse than OPT2.

5. **Proposition.** For an optimal cache initialized with $[1, ..., k]$, for a sequence of k memory accesses at indices between 1 and $k+1$ inclusive, there is at most 1 cache miss.

Proof. For an optimal cache, note that all memory accesses before the first access at address $k+1$ must necessarily access the memory at addresses between 1 and k. Since the cache contains precisely the memory at those addresses, until such an access, there cannnot be a cache miss and the cache set remains the same. Then, at the first $k+1$ access, there *must* be a cache miss, since the cache has not seen that memory addresss before. (If there is no access at $k+1$, then all accesses are between 1 and k and clearly there are no misses.) In any case, afterwards, there are at most k memory accesses in the sequence. Since there are $k+1$ possible access locations, by the pigeonhole principle, at least one location is not accessed. This location is accessed farthest away in the future, and therefore the cache evicts the memory at this location. (If the address is $k+1$, then the cache simply does not store it at all.) Then, necessarily, the rest of memory accesses must form a subset of the cache and therefore there are no cache misses. In total, there is at most 1 miss.

Proposition. For any online cache of size k there exists a sequence of memory accesses of length k between 1 and $k+1$ such that the cache misses all of them.

Proof. Since our cache can only hold k elements and we're using $k+1$ possible memory accesses, there exists a value not within the cache by the pigeonhole principle. If we initially try to access this value we'll have to evict something, which, since we're not using an OPT algorithm, there's no guarantee will not show up again during our $k-1$ accesses. If we generate a sequence of memory accesses where on each successive access we try to access the element just evicted then we'll have k total misses.

Conclusion. For any non-random online cache, there exists a sequence of k memory accesses all of which the online cache will miss. However, by the first proposition, the optimal cache misses at most 1 of these accesses. Therefore, we conclude that the competitive ratio of online caches is at least $k/1 = k$.

Array Traversal

1. Given an $n \times n$ array, if we traverse the array in the same order as the memory is stored then we can picture simply traversing a large chunk of memory in sequential order. With a cache line l and transfer time into the cache t, we can model the total time T it would take to read an $n \times n$ array via:

   $$T = \frac{n^2}{l}(t)$$

   Here, we're assuming l divides n as the problem states. Since we're going to read each row sequentially we first need to know how many cache lines, l, will be transferred into our cache per row:

   $$\frac{n}{l}$$

   We'll read the entire contents of a row once it's loaded into cache, therefore we'll only need to load each cache line once. Since we have n rows we'll load in a total number of cache lines equal to:

   $$(\frac{n^2}{l})$$

   Once we know the total number of cache lines read into cache, we simply multiply by the time to transfer, t, and we have the total time T for our array.

2. Given an $n \times n$ array and a total cache size of c, if we instead traverse the array in a different order than it is stored in memory, for example traversing in column major order an array stored in memory in row major order, we model the total time T needed to read an array via:

   $$T = \begin{cases} \frac{n^2}{l}(t) & \text{if } nl \leq c \\ n^2(t) \end{cases}$$

   Let's say we're reading in an array, stored in memory in row major order, in column major order. Since l divides n we know that for each row of our array we'll need to load a new cache line. If the size of our cache, c, is greater than or equal to the memory required to read in every row nl, we'll only need to read in each row once. In this case T is equivalent to reading in an array in the same order as it's stored in memory.

   If, however, c is smaller than the amount of space we need to load in every row (nl), we'll have to continuously evict rows stored previously in our cache. When we try to

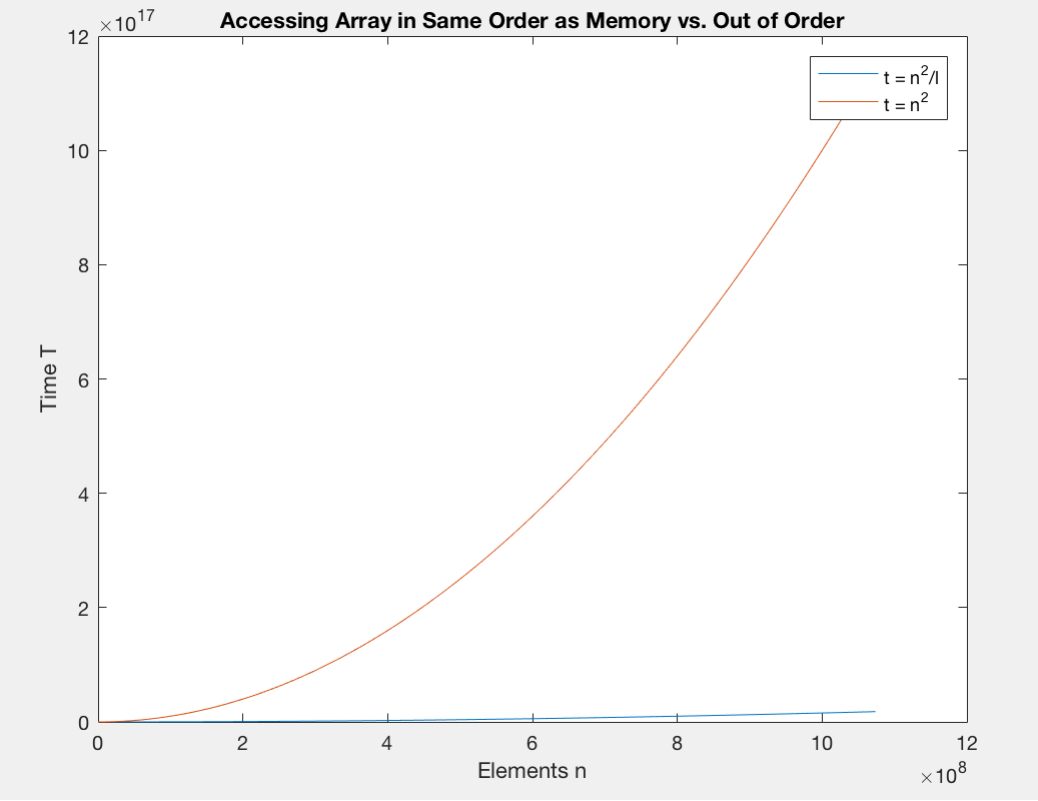
read in a value from an evicted row, then, we'll have to load into cache the same cache line. We'll do this l times.

Similar to above, we need to know how many cache lines there are per row. We therefore multiply by $\frac{n}{l}$ getting:

$$(nl)\frac{n}{l} = n^2$$

Analysis then runs similar to part 1: we multiply by t to get the total time T.

3. Taking $c = 2^{16}$, $l = 2^6$, and $t = 1$ let's plot time T as a function of n where n is the dimension of our array. Let's start our plot at $n = 2^{10}$. This is where the functions describing the number of cache loads diverges dependent on how we're traversing our array:



4. Examining the graph it's clear that if our code is accessing memory in the wrong order, it'll increase the number of cache misses and, thus, the overall running time of our code as cache lines have to be continually re-copied into the cache. If we notice our code running much slower than we expect, we might be running into cache issues.

Memory Re-allocation

1. Consider the assignment at index $i \leq n$ in a Matlab array A with $|A| \leq n$. Note that the cost of such an operation is 1 if $i < |A|$. Otherwise, the array of length $|A|$ has to be recopied and $i - |A|$ entries have to be initialized to zero so that the total cost is $|A| + i - |A| = i$ operations. This gives us that the cost is i if $i \geq |A|$ and 1 otherwise. In the worst case, since $i \leq n$, the cost is n and is therefore $O(n)$.

2. Let the array have size 0. Then, assign to its contents at indices $1, ..., n$ in order. Clearly, after step i the array has length i, therefore, it has to be recopied when the next assignment at $i + 1$ is made. We know from the previous question that each assignment i takes time i if $i > |A|$. Therefore, in total the time needed is

$$\sum_{i=1}^{n} i = n\frac{n+1}{2} = O(n^2).$$

   If we assign elements in any other order there will be cases when the array doesn't need to be reallocated and we can simply make the assignment in constant time. Thus, all other sequences take time at most $O(n^2)$ since each assignment takes time at most $O(n)$ and there are n assignments.

3. Let n be the final size of the array. Note that for the optimal algorithm, the time needed is the time for array A to be initialized to all zeros and then for all entries to be assigned. Therefore the total time is $n+n = 2n$. Hence, the optimal algorithm is $O(n)$.

   In the original Matlab algorithm, however, the array must be reallocated with each assignment. Part 2 generalizes here where we're making n assignments such that the time needed is $\sum_{i=1}^{n} +i = O(n^2)$. The competitive ratio is therefore $\mathrm{O}(\frac{n^2}{n}) = O(n)$.

4. Since our optimal algorithm knows ahead of time the max size of our array the total memory required is simply n.

   For any array of length k, an assignment that extends the array to length m will require the machine to maintain the old array and allocate memory for the new array. This requires $m + k$ memory. Therefore, the worst case memory usage given that a sequence of assignments that extends the array to length at most n is

$$\max_{k}(m_k + m_{k+1})$$

   where m_k is the length of the array after assignment k. Clearly, each assignment must extend the array and the final size of the array is n, therefore, $n \geq m_k > m_{k-1} > ... >$

m_0. Therefore, in the worst case, the memory usage is $n + m_{k-1}$. Since m_{k-1} is at most $n - 1$, the memory usage is bounded by $n + n - 1 = 2n - 1$.

This gives us a competitive ratio of the memory requirement of $\frac{2n}{n} = 2$.

5. (a) Suppose that Matlab now uses a new strategy where each time we enlarge our array the resultant size is a power of 2. Let's compare this to the OPT strategy.

Suppose we begin with some array A of length k and the final size of the array is n. We showed in the previous part that the memory required for extending the array of size k to size m is $k + m$. And as the array becomes larger, the memory requirement becomes strictly larger. Therefore, the maximum memory requirement at any given point is the memory required for the last assignment, at which point the array is at its largest. Clearly, this is bounded by $n + n - 1$ since the final size of the array is n and the size before the last array-extending assignment must be at most $n - 1$. Note that extending an array of size $2^k - 1$ to an array of size $n = 2^k$ maximizes the requirement and therefore, $2n - 1$ is a tight bound.

The competitive ratio between our new strategy and OPT is simply $O(\frac{2n}{n}) = 2$, the same as the ratio between the original Matlab algorithm and OPT. Thus, even though we're enlarging our array by more than we need to we aren't negatively affecting the asymptotic competitive ratio of our algorithm when compared to OPT.

(b) With this strategy, the first expansion makes the array have the size 2^m for some m. Then, subsequent expansions, since the array now has size 2^m, must make the array have size at least 2^{m+1} and must therefore at least double the size of the array. Suppose that there are k expansions in total, let $i_1, ..., i_k$ be the size of the array after each expansion and i_0 the initial size of the array. Note that for all $m > 0$, i_m is a power of two and $i_m \leq 2i_{m+1}$. Moreover, note that $i_k \geq n > i_{k-1}$ since the assignment must be greater than the length of the array to cause an expansion. Clearly, the total time cost is $i_0 + \sum_{0<m\leq k} i_m$ and, $\{i_1, ..., i_k\} \subseteq \{2^0, ..., 2^p\}$ where $2^p \geq n > 2^{p-1}$. This implies that $p = \lceil \log n \rceil$. Suppose that $i_1 = 2^t$ for some t, then since

$$\sum_{0<m\leq k} i_k < \sum_{t\leq q\leq p} 2^q$$

and

$$i_0 \leq i_1 - 1 = 2^t - 1 = \sum_{q<t} 2^q$$

we have that

$$i_0 + \sum_{m\leq k} i_k \leq \sum_{q<t} 2^q + \sum_{t\leq q\leq p} 2^q = \sum_{q\leq p} 2^p = 2^{p+1} - 1.$$

Therefore, the time requirement is at most $2^{p+1} - 1$ where $p = \lceil \log n \rceil$. However, the starting with an array of length 0 and assigning to indices 2^q for $0 \leq q \leq p$ requires

$$\sum_{q \leq p} 2^p = 2^{p+1} - 1$$

and is therefore, the worst case. In this case, the time complexity is at most

$$2^{p+1} - 1 = 2^{\lceil \log n \rceil + 1} \leq 2^{\log n + 2} = 4n.$$

which is $O(n)$.

Note that each expansion doubles the size of the array, therefore, there are at most $\log n$ array-extending assignments if the final size of the array is at n. Hence, the optimal assignment scheme takes at least n operations for reallocation and $\log n$ operations for reassignment, resulting in $O(n + \log n)$ operations total. Hence, the competitive ratio is at most

$$\frac{4n}{n \log n} = 4 + \frac{4}{\log n} \to 4.$$

and the competitive ratio is $O(1)$

6. This seems like a reasonable strategy since, according to what we've shown above, the algorithm from part 5 has a constant competitive ratio in terms of memory and performance when compared to OPT. Furthermore, there doesn't seem to be much downside, especially in terms of memory usage, from implementing this algorithm. Even though you're expanding the array more than it needs to, you're still competitive within the same constant factor of the OPT as the naive Matlab reallocation algorithm. Thus, you're no worse competitively in terms of memory usage and you're more competitive in terms of running time, so there's no downside.

Edit Distance

1. See *EditDistance.java*

2. This approach is an improvement over the original implementation because we're using substantially less memory. Instead of having to initialize an entire $(N+1) \times (N+1)$ table where N is the length of the string, we can instead get away with only using 2 $N+1$ arrays. Not only does this decrease our memory usage, but it also speeds up our code, greatly reducing the time needed initializing data structures.

3. See *EditDistance.java*

4. What we notice is that, given row r of elements where k is the base value down column 0, we can construct row $r+1$ simply by shifting diagonally the elements along the row:

r:	k	a	b	c	d	e

$r+1$:	$k+1$	k	a	b	c	d

Using the above idea, but only one array we can traverse the elements in our array by simply shifting.

k	...	2	1	0	a_1	b_1	c_1	d_1	e_1

k	...	2	1	0	a_2	b_2	c_2	d_2	e_2

k	...	2	1	0	a_3	b_3	c_3	d_3	e_3

...

Thus, if we shift the elements as shown above, in order to fill index i we only need to look at indices $[i-1], [i], [i+1]$. If the characters being compared are the same, instead of having to copy a value, because of our shift, that value is already at array position $[i]$, thus we don't have to do anything.

5. In order to test the running times, we took the average of 5 runs on strings of length 20000 ($time1$) and 10000 ($time2$), respectively. Our results are as follows:

ordering	time1	time2
([i-1][j-1], [i-1][j]), [i][j-1])	3676ms	943.8ms
([i-1][j-1], [i][j-1]), [i-1][j])	3554.2ms	913.6ms
([i-1][j], [i][j-1]), [i-1][j-1])	3631.8ms	986ms

From the above, it seems clear that the three way comparison in the order:

([i-1][j-1], [i][j-1]), [i-1][j])

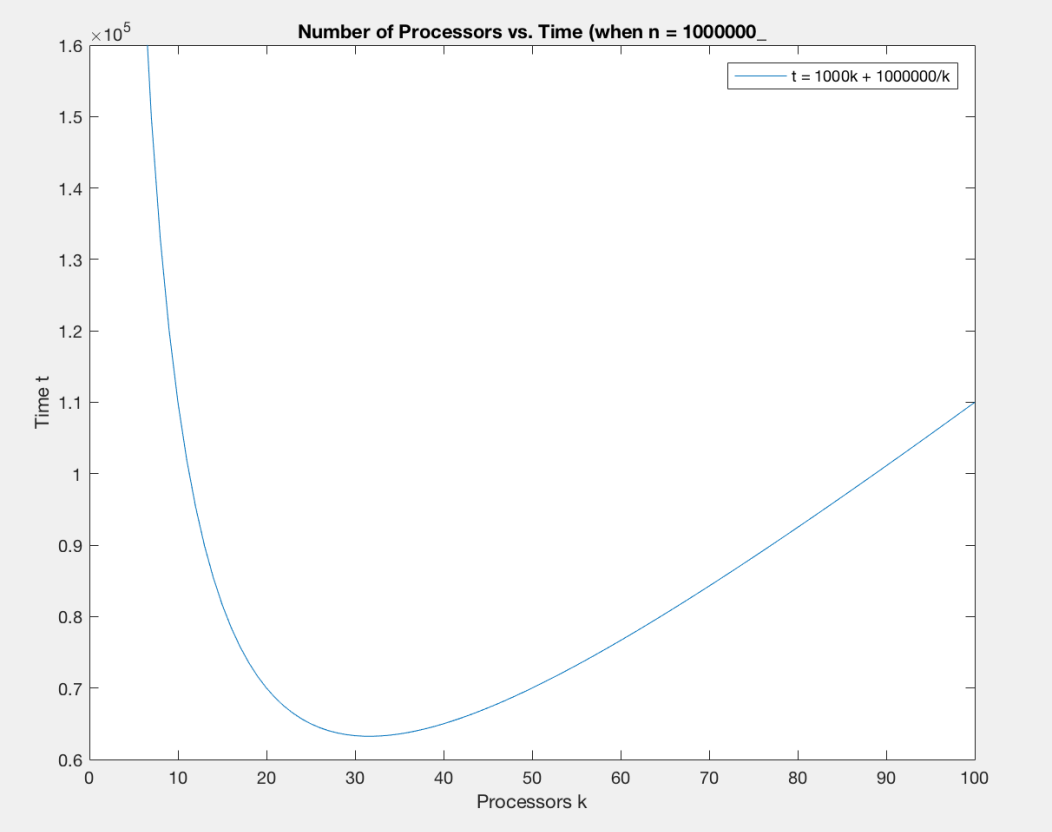
performs best. Before testing, I would've assumed ([i-1][j-1], [i-1][j]), [i][j-1]) would perform best. My intuition is that the first pair of entries is on the same row, and thus closer together in memory, making it less expensive to access these memory locations before jumping to a different row.

Perhaps we get the result we do due to effective pipe lining or perhaps the order we're actually accessing memory in our best case is optimal. Perhaps our processor is able to predict that the [i][j-1] is likely greater than [i-1][j-1], and therefore branch prediction is slightly speeding up our code. I would assume one of these would explain the differences in speed.

6. See *EditDistance.java*

7. If we let the time to start a processor be 1000 and the number of tasks $n = 1000000$, then the time to get work down can be modeled via:

$$t = 1000k + 1000000/k:$$



To find the optimal number of processors needed, simply take the derivative of the above function and find its local min:

$$0 = 1000 - 1000000/k^2$$
$$1000000 = 1000 * k^2$$
$$1000 = k^2$$
$$k = 10\sqrt{10} \approx 31.62$$

So, we'd want about 32 processors to do the above work in an optimal amount of time.

8. Given n tasks and p time to start a processor, the optimal number of processors to use is modeled by:

$$t = \sqrt{(n/p)}$$

Inspiring

Each of the problems above illustrated important points:

1. In problem 1 you're looking at different eviction policies in terms of the cache. What we found interesting was the fact that it seems no 1 online strategy is, in all cases, optimal. For example, given a cache using a LRU or FIFO eviction policy, we can come up with sequences of memory accesses to make the LRU perform better than the FIFO and vice versa. Thus, all we can do is try to implement an algorithm that is competitive against the OPT and then use this ratio to compare eviction policies.

2. In class we've talked about how we can think of expensive operations in terms of memory accesses and arithmetic. This problem illustrated just how expensive memory accesses can be when we don't take into account how memory is laid out in the system. By minimizing our cache misses by referencing memory in the same order that it's stored, we can dramatically decrease the amount of time certain algorithms run.

3. Assuming our calculations in this part are correct, I was surprised that there's an algorithm, the one from part 5, which is seemingly better in every way than the naive reallocation algorithm Matlab used to implement. Not only is the algorithm from part 5 more competitive in terms of running time, being only a constant factor slower than OPT, but when compared to the naive algorithm's competitive memory ratio, it's the same. Although at first glance it might appear that always increasing an array by a factor of 2 would waste substantially more memory than expanding it only as much as it needs, when we look at these algorithms and how they compete with OPT, they're identical.

4. Lastly, it was exciting seeing how using the underlying architecture of the system could substantially increase performance. Coding in Java, we got to see just how expensive memory accesses can be, and how minimizing them speeds up your code. Similar to question 2, we also got to see how memory can be laid out in a certain order to increase performance. This time, we weren't necessarily worried about the cache, but rather freeing up dependencies to allow for parallel computation. It was fun taking a single algorithm and optimizing it, in sequence, to see just how much performance could be gleaned through slight tweaks.