1 Overview

In the last lecture we discussed amortized analysis and looked at the 3 methods: aggregate, accounting, and potential. We used these 3 methods to show that the amortized complexity of the binary counter problem is $O(n)$. Finally, we applied amortized analysis to the problem of inserting data into dynamic arrays, showing that inserting $n$ objects has an amortized cost of $O(n)$, despite the worst-case result of $O(n^2)$.

In this lecture we start looking at self-adjusting data structures, focusing on the move-to-front (MTF) algorithm. Move-to-front was introduced in [ST85] and the details can be found therein. Using competitive analysis, we see that it is $4$-competitive with the optimal solution, when performing a sequence of queries.

2 Searching a list

In this lecture, we focus on the problem of searching an array, without random access. Logically, we can think of the structure as a doubly-linked list. Therefore, the various operations we can perform, and their relative work, are:

- Insert(x): $O(n)$ time - append to end,
- Find(x): $O(i)$ time - step through list until $x$ is found, $i$ is the index of $x$,
- Delete(x): $O(i)$ time - find $x$ and delete it.

2.1 Static Structure

We first consider the static version of the problem that involves only searching the list of $n$ elements (i.e., only perform Find(x)). Furthermore, we consider the case where the list of $m$ queries, and their respective frequencies, $f_x$ are known in advance. With this known, we can easily define the optimal solution by ordering the list of $n$ elements in non-decreasing order by the frequency, such that, for some $x_i$ and $x_j$ $i < j \iff f_{x_i} \geq f_{x_j}$. Informally, elements with higher query frequencies appear first in the list, reducing the total cost of performing all $m$ queries.

To prove this is optimal, we simply consider swapping 2 elements in the layout. We then compare the total time of performing all $m$ queries in the original layout, $T(m)$, with that of the new layout $T'(m)$. We simply need to show that $T(m) \leq T'(m)$, to show that the original layout, $T(m)$ is optimal.
2.2 Dynamic Problem

We now consider the problem of performing \(m\) queries on \(n\) elements, but without previous knowledge of query frequencies. However, we add the use of the swap operation. The swap operation swaps the position of two neighboring list elements.

Using the swap operation, we introduce the **move-to-front** method that modifies \(\text{Find}(x)\) to the following:

\(\text{Find}(x)\): Searches for \(x\) and moves it to the front. This operation takes \(i\) work to find \(x\), and \(i - 1\) work to move the \(x\) element to the beginning of the list (using swap). Thus, \(\text{Find}(x)\) for move-to-front takes \(2i - 1\) total work.

### 2.3 Competitive Analysis

**Definition 1.** Algorithm \(A\) is \(c\)-competitive if, for any input of size \(n\):

\[
T_A(n) \leq c \cdot T_{OPT}(n) + O(1)
\]

Applying competitive analysis to the move-to-front algorithm, we assume \(A^*\) is the optimal solution when performing a given sequence of \(n\) queries, and \(T_{A^*}(n)\) is the time for \(A^*\) to perform all \(n\) queries. If \(A\) is our move-to-front algorithm, we can determine if \(A\) is \(c\)-competitive if:

\[
\frac{T_A(n)}{T_{A^*}(n)} = c + o(1)
\]

The strategy to show that \(A\) is \(c\)-competitive, therefore, is to compute \(T_A(n)\) as a function of \(T_{A^*}(n)\). We can then use amortized analysis to prove that the amortized cost to search \(A\) is within a constant factor, \(c\), of \(A^*\). For this problem, we use the potential method of amortized analysis. We first define the potential function as:

\[
\Phi(D_i) = 2 \cdot \text{Inv}(A_i, A^*_i)
\]

Where \(A_i\) and \(A^*_i\) are the states of the arrays of \(A\) and \(A^*\) after the \(i\)th query, using the move-to-front and optimal algorithms, respectively. The \(\text{Inv}(X, Y)\) function is defined as the number of pairs of elements that differ in ordering between the \(X\) and \(Y\) arrays. Formally, we define it as:

\[
\text{Inv}(A, B) = \# \text{ of pairs } (x, y) \text{ s.t. } x = A[i] \& y = A[j], x = B[i'] \& y = B[j'] \text{ and } \[i < j \& i' > j'] \text{ or } [i > j \& i' < j']
\]

Informally, this is the metric of the number of elements that are in a different order between two arrays.
**Example:** Given the lists $A = [3, 7, 5, 2]$ and $B = [7, 2, 3, 5]$, what is $\text{Inv}(A, B)$?

To determine the number of inversions, we look at each pair of elements in $A$ and $B$ and see if their relative positions are the same. In this example, we see that, in $A$, 3 comes before 7, but in $B$, 7 comes before 3, therefore this is an inversion. Applying this test to every pair of elements, we see that $\text{Inv}(A, B) = 3$ because the pairs $(3, 7)$, $(3, 2)$, and $(2, 5)$ are inverted between $A$ and $B$.

Using the potential method, the amortized cost for the $i$th operation is defined as $\hat{c}_i = c_i + \Delta \Phi_i$, where $\Delta \Phi_i = \Phi(D_i) - \Phi(D_{i-1})$. Therefore, to determine the amortized cost, we need to determine the change in the number of inversions between the optimal $A^*$ and move-to-front $A$, when performing a query.

Assume query $i$ is searching for target element $x$. We define $j$ and $k$ as the index of $x$ in $A^*$ and $A$, respectively. When performing a query, element $x$ is swapped $k - 1$ times in $A$, until it is the first element in $A$. Figure 1 illustrates this process when performing the $i$th query on both $A^*$ and $A$.

We now need to determine the maximum number of *increases* and *decreases* in inversions when querying. Note, that moving $x$ to the front of $A$ affects only the order of $x$ and the elements that come before it. To determine the increase in inversions, we consider two cases: $j > k$ and $j \leq k$.

When $j > k$, every swap, in the worst case, can create a new inversion (because $A^*$ has $j$ elements *before* $x$). Therefore, the number of new inversions is at most $k - 1$. When $j \leq k$, however, we know that, the number of elements that precede $x$ in $A^*$ is exactly $j - 1$. After moving $x$ to the front of $A$, all these elements will be placed after $x$ in $A$. So the number of new inversions is also $j - 1$. This gives us the total *increase* in inversions is at most $\min(k - 1, j - 1)$.

Conversely, we always perform $(k - 1)$ swaps, of which at most $\min(k - 1, j - 1)$ create new inversions. Since every swap either creates or removes an inversion, the total decrease in inversions is at least $(k - 1) - \min(k - 1, j - 1)$. The total change in potential is then:
\[ \Delta \Phi_i = 2(\text{increase} - \text{decrease}) \]
\[ \leq 2(\min(k - 1, j - 1) - ((k - 1) - \min(k - 1, j - 1))) \]
\[ = 4 \cdot \min(k - 1, j - 1) - 2(k - 1) \]

Then the amortized cost of searching for \( x \) and moving it to front is:

\[ \hat{c}_i = c_i + \Delta \Phi_i \]
\[ \leq (k + (k - 1)) + 4 \cdot \min(k - 1, j - 1) - 2(k - 1) \]
\[ = 2k - 1 + 4 \cdot \min(k - 1, j - 1) - 2k + 2 \]
\[ = 1 + 4 \cdot \min(k - 1, j - 1) \]
\[ \leq 1 + 4(j - 1) \]
\[ \leq 4j \]

In the optimal solution, to find \( x \) we need to perform at least \( j \) work (since \( x \) is at index \( j \)), so \( c_i^* \geq j \), therefore:

\[ \hat{c}_i \leq 4j \leq 4c_i^* \]

We calculate the total amortized cost of performing \( n \) queries using the MOVE-TO-FRONT algorithm as:

\[ T(n) = \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq \sum_{i=1}^{n} 4c_i^* \]
\[ \leq 4 \cdot T_{OPT}(n) \]

Note that the above analysis assumed that the optimal solution did not perform any swaps. Let us see what happens if the optimal algorithm \( A^* \) also employs swaps.

The swaps that \( A^* \) performs does not affect the complexity of the MOVE-TO-FRONT algorithm. However, each swap may cause the # of inversions to increase by 1, which in turn causes \( \Delta \Phi_i \) to increase by at most 2. At the same time, the swaps also increase the cost \( \tilde{c}_i^* \) the optimal solution by the number of swaps that \( A^* \) performs Therefore:

\[ \tilde{c}_i = 4c_i^* + 2(\text{# of swaps}) \leq 4(c_i^* + \text{# of swaps}) = 4\tilde{c}_i^* \]

Therefore, MOVE-TO-FRONT is still 4-competitive with the optimal solution that uses arbitrary number of swaps.
References