Admin: In-class quiz on 10/19
PG #3 out today
Be careful on collaboration

Today: Amortized Analysis of Algorithms

- Cycle-finding (Nivasch)
- Move-to-front in a list (Sleator/Torjan)

Both algorithms extremely simple; analysis is the interesting part. Amortized analysis is appropriate for both.
Amortized Analysis: Overview

- We have some data structure that supports some base operations. (E.g. insert, search)
- So far, we have studied (w.c. typically) time per basic op.
- But we are often interested in w.c. time for a sequence of basic operations.
- E.g. Let \( S = B_1, B_2, \ldots, B_m \) be a sequence of \( m \) basic ops.
  
  Let \( t_1, t_2, \ldots, t_m \) be actual cost (running time) for each operation.

  Then \( \mathcal{T} = \sum_{i=1}^{m} t_i \) is total cost for sequence \( S \); this is what we are interested in bounding.

  Let \( a_1, a_2, \ldots, a_m \) be a set of "amortized" costs; to be valid, these should satisfy

  \[
  \sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i \quad (\ast)
  \]

  (There is no uniquely defined "amortized cost"; they can be defined arbitrarily as long as (\ast) holds.)
- My favorite algorithm that uses a stack!
- Like "multipop" analysis (see Chap. 17)

We want to determine if a system "is in a loop", and if so, determine size of loop. System is deterministic.

Initial state $x_0$

Update rule $x_i \rightarrow x_{i+1}$ (deterministic)

Loops if $x_i = x_j$ for $i \neq j$

(With $j$ minimal, then $\lambda = j - i$ is cycle length)

**E.g.**

\[ x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_6 \rightarrow x_7 \]

- $|\text{tail}| = \mu = 3$
- $x_8 = x_3$, cycles
- $x_\mu = 1^{st}$ repeated element
- $\lambda = 5$
- $\mu = 3$

Crypto, game of Life, ...
* Helpful to think of values $x_i$ as "random"; if not, we could hash them. (If they are "random", then stack size $\propto \log n$)
* We assume we can compare two values, not only to see if they are $=$, but to see which is larger.

**Algorithm**
* Given $x_0$ & update fn $x_i \rightarrow x_{i+1}$
* Returns $\lambda$ (cycle length) (can extended to get $x_\mu$, etc.)
* Stack initially empty
* Stack will hold pairs $(x_i, i)$. On stack, both $x_i$'s and $i$'s are increasing from bottom to top.
* To process $x_j$:
  * Pop all entries off stack $(x_i, i)$ where $x_i > x_j$
  * If an $x_i = x_j$ is found in stack, stop & return cycle length $\lambda = j - i$
  * Else push $(x_j, j)$ on stack & continue

**Correctness:**
* Each new value is pushed on stack (possibly after popping others)
* Let $x_k$ be minimum value on cycle be $x_k$.
* Once it is pushed onto stack, it will not be popped until it is seen again next time around, when cycle will be detected.
Analysis:
- Each push or pop costs 1 (actual) \( \text{"multiply \# \text{\# pushes is } k \ (\text{for } x_0, \ldots, x_{k-1})" } \)
- \# pushes is \( k \) (for \( x_0, \ldots, x_{k-1} \))
- \# pops is \( \ldots \) ?
- \( t_j = 1 + p_j \) (1 for pushing \( x_j \), \( p_j \) for multi-pop)
- \( t = \sum_{i=1}^{k} (1+p_j) \) terms are highly variable!

Solution: "prepay" for pops, so "amortized cost" is 0.
Let \( \Phi = \"potential\" = \"amt of pre-payment\"
Let amortized cost of push = \( \Delta \)
\( \Delta \) to pay for push
\( \Delta \) to go into bank (increase \( \Phi \)) to pay for its later removal.
\( \Phi \) is not real or implemented, just an accounting fiction.

Each push costs \( \Delta \)
Each pop costs 0 (actual cost is paid for by reducing \( \Phi \))

\[
q_j > t_j + (\Phi_j - \Phi_{j-1}) \text{ covers actual + } \Delta \text{ bank}
\]

\[
\sum_{j=1}^{k} q_j > \sum_{j=1}^{k} t_j + (\Phi_k - \Phi_0)
\]
In our case \( \Phi = \text{size of stack}, \Phi_0 = 0, \Phi_k \geq 0 \) so
\[
\sum_{j=1}^{k} t_j \leq \sum_{j=1}^{k} q_j = 2k
\]
Move T Front:

- Uses unordered list to implement dictionary
  - insert (x)
  - delete (x)
  - search (key)

- Could be implemented in an array or a doubly-linked list.
- Accessing i th element costs i
- Inserting costs n+1 (current size of list is n)
- After insert or access, elt may be moved for free closer to head of list by arbitrary amt. ("free exchanges")
- Deleting i th element costs i
- Algorithm may exchange (transpose) any two adjacent elements at any time (cost=1 "paid exchange")

Possible strategies: MTF (move to front)

  - move accessed elt to front
  - Transpose (move accessed elt up 1 position)
  - God's algorithm (uses knowledge of future to keep list in optimal order)
Beautiful result: uses amortized analysis to show MTF is within constant factor of any algorithm (including God's).

Let (for some algorithm A & sequence S of opns)

\[ C_A(S) = \text{total cost of all opns except paid exchanges} \]
\[ X_A(S) = \text{total cost of all paid exchanges} \]
\[ F_A(S) = \text{total # of free exchanges} \]

\[ m = |S| = \# \text{ of operations} \]

Assume list starts out empty.

**Example:**

\[
\begin{align*}
\{ & 1 \text{ insert 11} \rightarrow \begin{array}{c}11\end{array} \\
& 2 \text{ insert 20} \rightarrow \begin{array}{c}11 \rightarrow \end{array} \begin{array}{c}20\end{array} \\
& 3 \text{ insert 6} \rightarrow \begin{array}{c}11 \rightarrow \end{array} \begin{array}{c}20 \rightarrow \end{array} \begin{array}{c}6\end{array} \\
& 4 \text{ search 20} \rightarrow \begin{array}{c}20 \rightarrow \end{array} \begin{array}{c}11 \rightarrow \end{array} \begin{array}{c}6\end{array} \\
& 5 \text{ search 6} \rightarrow \begin{array}{c}6 \rightarrow \end{array} \begin{array}{c}20 \rightarrow \end{array} \begin{array}{c}11\end{array}
\end{align*}
\]

\[ m = 5 \]
\[ C_{MTF}(S) = 1 + 2 + 3 + 2 + 3 = 11 \]
\[ X_{MTF}(S) = 0 \text{ (always true for MTF, only uses free exchanges)} \]
\[ F_{MTF}(S) = 0 + 0 + 0 + 1 + 2 = 3 \]
Amazing theorem:

**Theorem:** For any algorithm A (even God's) and any sequence of operations $s$ (starting with empty list):

$$C_{MTF}(s) \leq 2C_A(s) + X_A(s) - F_A(s) - m \leq 2(C_A(s) + X_A(s))$$

"Competitive" analysis: MTF not worse than $2 \times$ alg A

MTF is very useful in practice, too!

- Fix algorithm A
- Define potential $\Xi$ as measure of how different list orders are for A & MTF (unusual analysis since $\Xi$ defined w.r.t. A)
- Give amortized analysis w.r.t. $\Xi$
- This illustrates how amortized analysis can vary, even for same alg.
  - We have different potential fn $\Xi$, and thus different amortized costs, for each other algorithm A!
- Amortized cost of MTF $\leq 2 \times$ actual cost of A
Proof:

Minor lemma: For any $A$: $F_A(s) \leq C_A(s) - m$

\[ \text{PF: After accessing or inserting an element, can do at most } m-1 \text{ free exchanges until elt is at front.} \]

Define

\[ \Phi = \# \text{ inversions} = \# \text{ pairs } i, j \text{ s.t. } A[i] > A[j] \]

\[ A's \text{ list } \quad \begin{array}{c} i \quad j \end{array} \]

\[ \text{MTF's list } \quad \begin{array}{c} j \quad i \end{array} \]

\[ \Phi_0 = 0 \quad (\text{empty list initially}) \]

\[ \Phi > 0 \quad \text{always (but } \leq \binom{n}{2}) \]

If actual MTF cost of opn is $t$, define

\[ \text{amortized cost} = t + \Phi' - \Phi \]

\[ (t \text{ plus increase in } \Phi) \]

Then

\[ \sum_{i=1}^{m} t_i = \sum a_i + (\Phi_0 - \Phi_m) \leq 0 \]

\[ \geq \sum a_i \]
Consider search of \( x \) at some step

Suppose \( x \) is \( i \)-th place on A's list, \( k \)-th place on MTF's

Amortized cost of finding \( x \) in MTF

\[
= \text{actual cost} + \frac{\text{increase in # inversions by moving } x \text{ to front}}{k}
\]

\[
= \frac{2k - 2v - 1}{v}
\]

\[
= 2(k - v) - 1
\]

(But \( k - v \leq i \) since there \( k - v \) els are before \( x \) in A's list)

\[
\leq 2i - 1
\]
Thus

$$\sum t_i \leq \sum a_i \leq 2C_A(s) - m + X_A(s) - F_A(s)$$

(a) (actual) MTF Cost (b) MTF Amortized Cost
For binary search trees, similar idea ("move to root via splaying after each search") gives $\Theta(\log n)$ amortized time per operation. (Sleator/Tarjan 1985)