



Algorithms and Data Structures

Amortized Analysis

Ulf Leser

-
- Two Examples
 - Two Analysis Methods
 - Dynamic Tables
 - SQL - Analysis

 - This lecture is not covered in [OW93] but, for instance, in [Cor03]

Setting

- SOL: Sequences of **operations influencing each other**
 - We have a sequence Q of operations on a data structure
 - Searching SOL and rearranging a SOL
 - Operations are not independent – by changing the data structure, costs of subsequent operations are influenced
- Conventional WC-analysis produces misleading results
 - Assumes all operations to be independent
 - Changing search order in a workload does not influence WC result
- **Amortized analysis** analyzes the complexity of any sequence of dependent operations
 - In other terms: We seek the **worst average cost** of each operation in any sequence

„Amortizing“

- Economics: How long does it take until a (high) **initial investment** pays off because it leads to **continuous business improvements** (less costs, more revenue)?
- Example:
 - Investment of 6000€ leads to daily rev. increase from 500 to 560€
 - Investment is amortized after 100 days
- WC: Look at **all days independently**
 - Look at ratio cost / revenue
 - Compare 560-6000 to 500-0
 - Don't invest!



Algorithmic Example 1: Multi-Pop

- Assume a **stack S** with a special operation: **mpop(k)**
 - mpop(k) pops $\min(k, |S|)$ elements from S
 - Implementation: mpop calls pop k times
- Assume **any sequence Q** of operations push, pop, mpop
 - E.g. $Q = \{\text{push}, \text{push}, \text{mpop}(k), \text{push}, \text{pop}, \text{push}, \text{mpop}(k), \dots\}$
- Assume costs $c(\text{push}) = 1$, $c(\text{pop}) = 1$, $c(\text{mpop}(k)) = k$
- With $|Q| = n$: What cost do we expect for a given Q?
 - Every op in Q costs 1 (push) or 1 (pop) or k (mpop)
 - In the worst case, k can be $\sim n$ (n times push, then one mpop(n))
 - Worst case of a single operation is $O(n)$
 - **Total worst-case** cost: $O(n^2)$



Note: Costs only $\sim 2 * n$

Problem

- Clearly, the cost of Q is in $O(n^2)$, but this is **not tight**
- A simple thought shows: The cost of Q is in $O(n)$
 - Every element can be **popped only once**
 - No matter if this happens through a pop or a mpop
 - Pushing an element costs 1, popping it costs 1
 - Q can at most push n elements and, hence, only pop n elements
 - Thus, the total cost is in $O(n)$
 - It is maximally $2^{*(n-1)}$
- We want to derive such a result in a systematic manner (analyzing SOLs is not that easy)

Example 2: Bit-Counter

- We want to generate **bitstrings** by iteratively adding 1
 - Starting from 0
 - Assume bitstrings of length k
 - Roll-over counter if we exceed 2^k-1
- Q is a sequence of „+1“
- We count as cost of an operation the number of **bits we have to flip**
- Classical WC analysis
 - A single operation can flip up to k bits
 - “1111111” +1
 - Worst case cost for Q : $O(k*n)$

00000000		
00000001	1	1
00000010	2	3
00000011	1	4
00000100	3	7
00000101	1	8
00000110	2	10
00000111	1	11
00001000	4	15
00001001	1	16
00001010	2	18
...		

Problem

- Again, this complexity is overly pessimistic / not tight
- Cost actually is in $O(n)$
 - The right-most bit is flipped in every operation: cost= n
 - The second-rightmost bit is flipped every second time: $n/2$
 - The third ...: $n/4$
 - ...
 - Together

$$\sum_{i=0}^{k-1} \frac{n}{2^i} < n * \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 * n$$

-
- Two Examples
 - Two Analysis Methods
 - Accounting Method
 - Potential Method
 - Dynamic Tables
 - SOL - Analysis

Accounting Analysis

- Idea: We **create an account** for Q
- Operations put / withdraw some amounts of “money”
- We choose these amounts such that the current state of the account is always (throughout Q) an **upper bound of the actual cost incurred by Q**
 - Let c_i be the true cost of operation i , d_i its effect on the account
 - We require
$$\forall 1 \leq k \leq n: \sum_{i=1}^k c_i \leq \sum_{i=1}^k d_i$$
 - Especially, the account must never become negative
 - “ \leq ” gives us more freedom in analysis than “ $=$ ”
- It follows: An **upper bound for the account** (d) after Q is also an upper bound for the true cost (c) of Q

Application to mpop

- Assume $d_{\text{push}}=2$, $d_{\text{pop}}=0$, $d_{\text{mpop}}=0$
- Upper bounds?
 - Clearly, d_{push} is an upper bound on c_{push} (which is 1)
 - But neither d_{pop} nor d_{mpop} are upper bounds for c_{pop} / c_{mpop}
- Let's try: $d_{\text{push}}=2$, $d_{\text{pop}}=1$, $d_{\text{mpop}}=n$
 - Now all individual d 's are upper bounds for their c 's
 - But this doesn't help (worst-case analysis)

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n d_i \leq n * n \in O(n^2)$$

- But: We only have to show that the **sum of d 's** for any prefix of Q is higher than the sum of c 's

Application to mpop

- Assume $d_{\text{push}}=2$, $d_{\text{pop}}=0$, $d_{\text{mpop}}=0$
- Summing these up **yields an upper bound** on the real cost
 - Idea: Whenever we push an element, we pay 1 for the push and 1 for the operation that will (at same later time) **pop exactly this element**
 - It doesn't matter whether this will be through a pop or a mpop
 - Note: For every pop, there must have been a **corresponding push** before
 - Thus, when it comes to a pop or mpop, there is **always enough money** on the account
 - Deposited by previous push's
 - "enough": Enough such that the sum remains an upper bound

- This proves:

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n d_i \leq 2 * n \in O(n)$$

Choose d 's carefully

- Assume $d_{\text{push}}=1$, $d_{\text{pop}}=1$, $d_{\text{mpop}}=1$
 - Assume $Q=\{\text{push},\text{push},\text{push},\text{mpop}(3)\}$
 - $\sum c=6 > \sum d = 4$
- Assume $d_{\text{push}}=1$, $d_{\text{pop}}=0$, $d_{\text{mpop}}=0$
 - Assume $Q=\{\text{push},\text{push},\text{mpop}(2)\}$
 - $\sum c=4 > \sum d = 2$
- Assume $d_{\text{push}}=3$, $d_{\text{pop}}=0$, $d_{\text{mpop}}=0$
 - Fine as well, but not as tight (but also leads to $O(n)$)

Application to Bit-Counter

- Look at the **sequence Q' of flips** generated by a sequence Q
 - Every $+1$ creates a sequence of $[0, k]$ flip-to-0 and **$[0, 1]$ flip-to-1**
 - There is no „flip to 1“ if we roll-over
 - Since only flips cost, the cost of Q' is the same as the cost of Q
- Let's try $d_{\text{flip-to-1}}=2$ and $d_{\text{flip-to-0}}=0$
 - Clearly, $d_{\text{flip-to-1}}$ is an upper bound to $c_{\text{flip-to-1}}$
 - Note: We start with only 0 and can flip-to-0 any 1 only once
 - Before we flip-to-1 again, again enabling one flip-to-0 etc.
 - Idea: When we flip-to-1, we pay 1 for flipping and 1 for the **back-flip-to-0 that might happen** at some later time in Q'
 - There can be only one flip-to-0 per flip-to-1
 - Thus, the account is always an upper bound on the actual cost

Application to Bit-Counter -2-

- We know that the account is always an upper bound on the actual cost for any prefix of Q
- Every step of Q creates a sequence of flip-to-1 (at most one) and flip-to-0 in Q'
- This sequence costs at most 2
 - There can be only one flip-to-1, and all flip-to-0 are free
- Every step in Q costs at most 2
- Thus, Q is bound by $O(n)$
- qed.

-
- Two Examples
 - Two Analysis Methods
 - Accounting Method
 - Potential Method
 - Dynamic Tables
 - SOL - Analysis

Potential Method: Idea

- In the accounting method, we assign a cost to every operation and compare aggregated accounting costs of ops with aggregated real costs of ops
- In the **potential method**, we assign a **potential $\Phi(D)$ to the data structure D** manipulated by Q
- As ops from Q that change D , also change D 's potential
- The trick is to design Φ such that we can (again) use it to derive an **upper bound on the real cost** of Q

- "Accounting" and "potential" methods are quite similar – use whatever is **easier to apply** for a given problem

Potential Function

- Let D_0, D_1, \dots, D_n be the states of D when applying Q
- We define the **amortized cost** d_i of the i 'th operation as $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- We derive the **amortized cost of Q** as

$$\sum_{i=1}^n d_i = \sum_{i=1}^n (c_i + \phi(D_i) - \phi(D_{i-1})) = \sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0)$$

- Rough idea: If we find a Φ such that (a) we can obtain formulas for the amortized costs for all individual d_i and (b) $\Phi(D_n) \geq \Phi(D_0)$, we have an **upper bound for the real costs**

– Because then:

$$\sum_{i=1}^n d_i = \sum_{i=1}^n c_i + \phi(D_n) - \phi(D_0) \geq \sum_{i=1}^n c_i$$

Details: Always Pay in Advance

- Operations raise or lower the potential (\sim future cost) of D
- We need to find a function Φ such that
 - Req. 1: $\Phi(D_i)$ depends on a property of D_i
 - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ [and we will always have $\Phi(D_0) = 0$]
 - Req. 3: We can compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- As within a sequence we do not know its future, we also have to require that $\Phi(D_i)$ never is negative
 - Otherwise, the amortized cost of the prefix $Q[1-i]$ would not be an upper bound of the real costs at step i
- Idea: Always **pay in advance**

Example: mpop

- We use the **number of objects on the stack** as its potential
- Then
 - Req. 1: $\Phi(D_i)$ depends on a property of D_i
 - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0) = 0$
 - Req. 3: Compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ for all ops:
 - If op is push: $d_i = c_i + (x - (x-1)) = 1 + 1 = 2$
 - If op is pop: $d_i = c_i + (x - (x+1)) = 1 - 1 = 0$
 - If op is mpop(k): $d_i = c_i + (x - (x+k)) = k - k = 0$
- Thus, $2*n \geq \sum d_i \geq \sum c_i$ and Q is in $O(n)$

Example: Bit-Counter

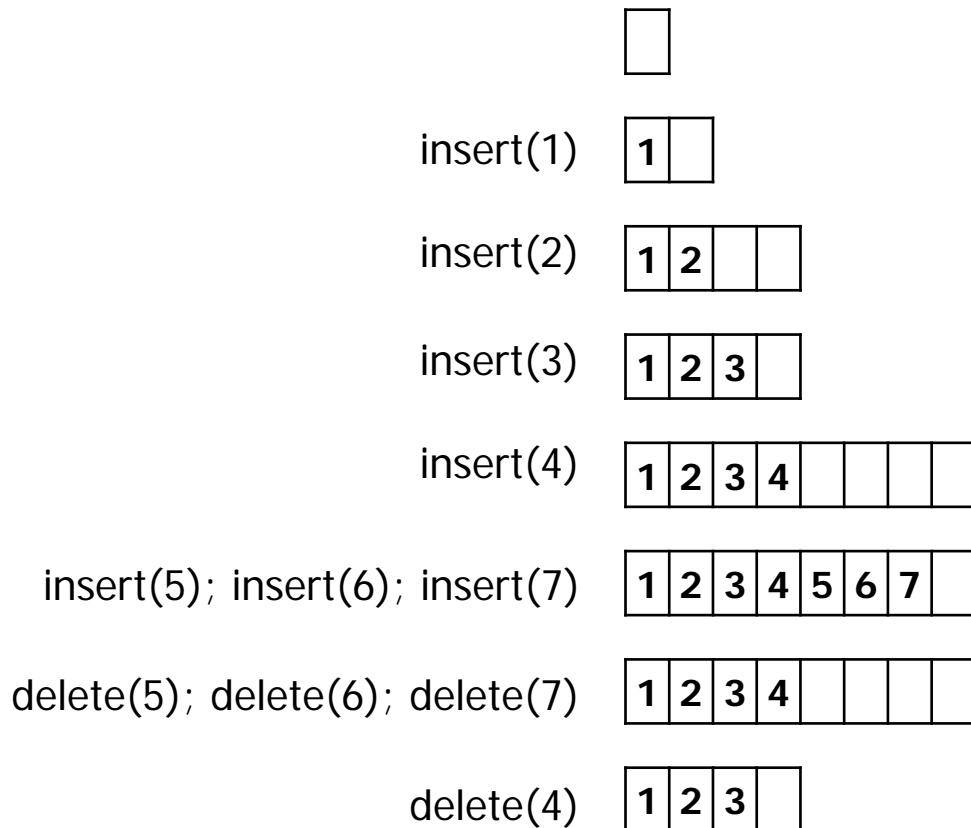
- We use the **number of „1“ in the bitstring** as its potential
- Then
 - Req. 1: $\Phi(D_i)$ depends on a property of D_i
 - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0) = 0$
 - Req. 3: We compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ for all ops
 - Let the i 'th operation incur t_i flip-to-0 and 0 or 1 flip-to-1
 - Thus, $c_i \leq t_i + 1$
 - If $\Phi(D_i) = 0$, then operation i has flipped all positions to 0; this implies that previously they were all 1, which means that $\Phi(D_{i-1}) = k$
 - If $\Phi(D_i) > 0$, then $\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$
 - In both cases, we have $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$
 - Thus, $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (\Phi(D_{i-1}) - t_i + 1) - \Phi(D_{i-1}) \leq 2$
- Thus, **$2 * n \geq \sum d_i \geq \sum c_i$** and Q is in $O(n)$

-
- Two Examples
 - Two Analysis Methods
 - **Dynamic Tables**
 - SOL are complicated ... we still try to get familiar with the analysis method using simpler problems ...
 - SOL - Analysis

Dynamic Tables

- We use amortized analysis for something more useful: Complexity of operations on a **dynamic table**
- Assume an array T and a sequence Q of insert/delete ops
- **Dynamic Tables**: Keep the array small, yet avoid overflows
 - Start with a table T of size 1
 - When inserting and T is full, **we double $|T|$** ; upon deleting and T is only half-full, we reduce $|T|$ by 50%
 - “Doubling”, “reducing” means: Copying data to a new array
 - Copying an element of an array costs 1
- Thus, any operation (ins or del) costs **either 1 or i**

Example



- Conventional WC analysis
- As i can be up to n , the complexity of any operation is $O(n)$
- Complexity of any Q is $O(n^2)$

With Potential Method

- 1: $\Phi(D_i)$ depends on a property of D_i
- 2: $\Phi(D_n) \geq \Phi(D_0)$
- 3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- Let $\text{num}(T)$ be the current number of elements in T
- We use potential $\Phi(T) = 2 * \text{num}(T) - |T|$
 - Intuitively a “potential”
 - Immediately before an expansion, $\text{num}(T) = |T|$ and $\Phi(T) = |T|$, so there is **much potential in T** (we saved for the expansion to come)
 - Immediately after an expansion, $\text{num}(T) = |T|/2$ and $\Phi(T) = 0$; **all potential has been used**, we need to save again for the next expansion
 - Formally
 - Requirement 1: Of course
 - Requirement 2: As T is always at least half-full, $\Phi(T)$ is always ≥ 0 ; we start with $|T|=0$, and thus $\Phi(T_n) - \Phi(T_0) \geq 0$

- | |
|--|
| 1: $\Phi(D_i)$ depends on a property of D_i
2: $\Phi(D_n) \geq \Phi(D_0)$
3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ |
|--|

Continuation

- Req. 3: Let's look at $d_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ for insertions

- Without expansion

$$\begin{aligned}
 d_i &= 1 + (2 * \text{num}(T_i) - |T_i|) - (2 * \text{num}(T_{i-1}) - |T_{i-1}|) \\
 &= 1 + 2 * \text{num}(T_i) - 2 * \text{num}(T_{i-1}) - |T_i| + |T_{i-1}| \\
 &= 1 + 2 + 0 \\
 &= 3
 \end{aligned}$$

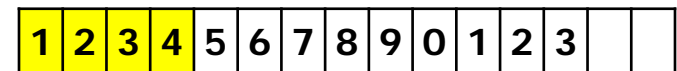
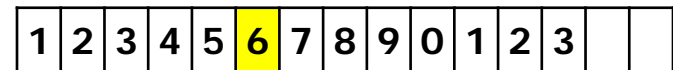
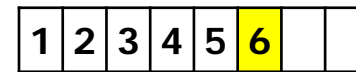
- With expansion

$$\begin{aligned}
 d_i &= \text{num}(T_i) + (2 * \text{num}(T_i) - |T_i|) - (2 * \text{num}(T_{i-1}) - |T_{i-1}|) \\
 &= \text{num}(T_i) + 2 * \text{num}(T_i) - |T_i| - 2 * \text{num}(T_{i-1}) + |T_{i-1}| \\
 &= \text{num}(T_i) + 2 * \text{num}(T_i) - 2 * (\text{num}(T_i) - 1) - 2 * (\text{num}(T_{i-1}) - 1) + \text{num}(T_i) - 1 \\
 &= 3 * \text{num}(T_i) - 2 * \text{num}(T_i) + 2 - 2 * \text{num}(T_{i-1}) + 2 + \text{num}(T_i) - 1 \\
 &= 3
 \end{aligned}$$

- Thus, $3 * n \geq \sum d_i \geq \sum c_i$ and Q is in $O(n)$ (for only insertions)

Intuition

- Think accounting method
- For insert', we deposit 3 because
 - 1 for the insertion (the real cost)
 - 1 for the time when we need to **copy this new element** at the next expansion
 - These 1's fill the account with $|T_i|/2$ before the next expansion
 - 1 for **one of the $|T_i|/2$ elements** already in A after the last expansion
 - These fill the account with $|T_i|/2$ before the next expansion
- Thus, we have enough credit at the next expansion



Problem: Deletions

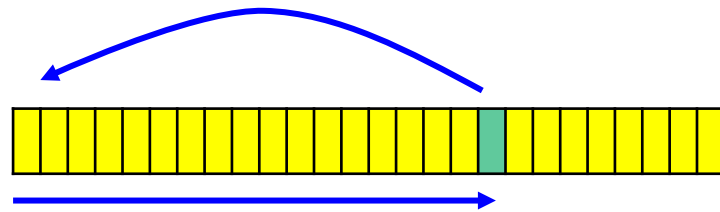
- Our strategy for deletions so far is not very clever
 - Assume a table with $\text{num}(T) = |T|$
 - Assume a sequence $Q = \{I, D, I, D, I, D, I \dots\}$
 - This sequence will perform $|T| + |T|/2 + |T| + |T|/2 + \dots$ real ops
 - As $|T|$ is $O(n)$, Q is in $O(n^2)$ and not in $O(n)$
- Simple trick: Do only contract when $\text{num}(T) = |T|/4$
 - Leads to amortized cost of $O(n)$ for any sequence of operations
 - We omit the proof (see [Cor03])

-
- Two Examples
 - Two Analysis Methods
 - Dynamic Tables
 - SOL – Analysis
 - Goal and idea
 - Preliminaries
 - A short proof

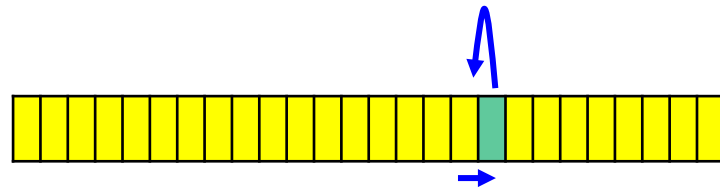
Re-Organization Strategies

- Recall self-organizing lists (SOL)
 - As usual: Accessing the i 'th element costs i
 - When searching an element, we **change the list L**
- Three popular **strategies**

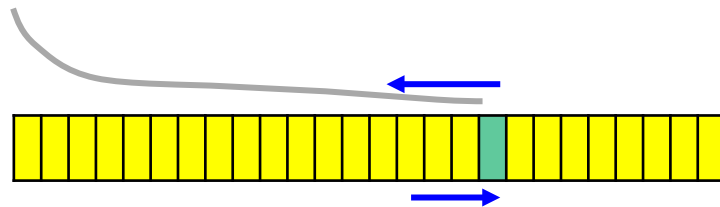
– MF, move-to-front:



– T, transpose:



– FC, frequency count:



Notation

- Assume we have a **strategy A** and a **workload S** on **list L**
- After accessing element i , **A may move i by swapping**
 - Swap with predecessor (to-front) or successor (to-back)
 - Let $F_A(l)$ be the number of front-swaps and $X_A(l)$ the number of back-swaps of step l when using strategy A
 - This means: F_{MF}/X_{MF} for strategy MF, $F_T/X_T \dots F_{FC}/X_{FC}$
 - We never back-swap: $\forall l: X_{MF}(l)=X_T(l)=X_{FC}(l)=0$
- Let **$C_A(S)$ be the total access cost** of A incurred by S
 - Again: C_{MF} for strategy MF, C_T for T, C_{FC} for FC
- With conventional worst-case analysis, we can only derive that $C_A(S)$ is in $O(|S| * |L|)$ – for any A

Theorem

- Theorem (Amortized costs)

*Let A be **any self-organizing strategy** for a SOL L , MF be the move-to-front strategy, and S be a sequence of accesses to L . Then*

$$C_{MF}(S) \leq 2 * C_A(S) + X_A(S) - F_A(S) - |S|$$

- What does this mean?
 - We don't learn more about the absolute complexity of SOLs
 - But we learn that **MF is quite good**
 - Any strategy following the same constraints (only series of swaps) will at best be **roughly twice as good as MF**
 - Assuming $C_A(S) \gg |S|$ and for $|S| \rightarrow \infty$: $X(S) \sim F(S)$ for any strategy
 - Despite its simplicity, **MF is a fairly safe bet** for all workloads

Idea of the Proof

- We will compare access costs in L between MF and **any A**
- Think of both strategies (MF, A) **running S on two copies** of the same initial list L
 - After each step, A and MF perform different swaps, so all list states except the first very likely are different
- We will compare list states by looking at the **number of inversions** (“Fehlstellungen”)
 - Actually, we only analyze how the number of inversions changes
- We will show that the **number of inversions defines a potential** of a **pair of lists** that helps to derive an upper bound on the differences in real costs

Content of this Lecture

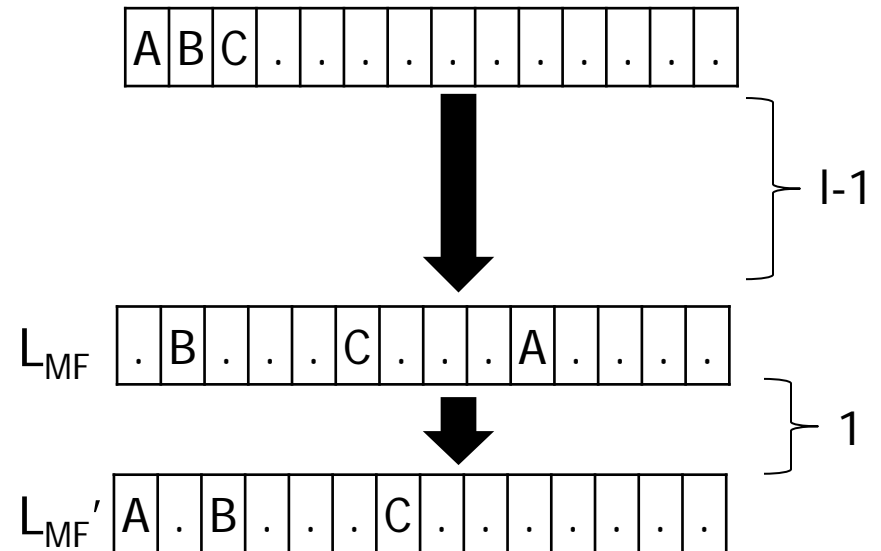
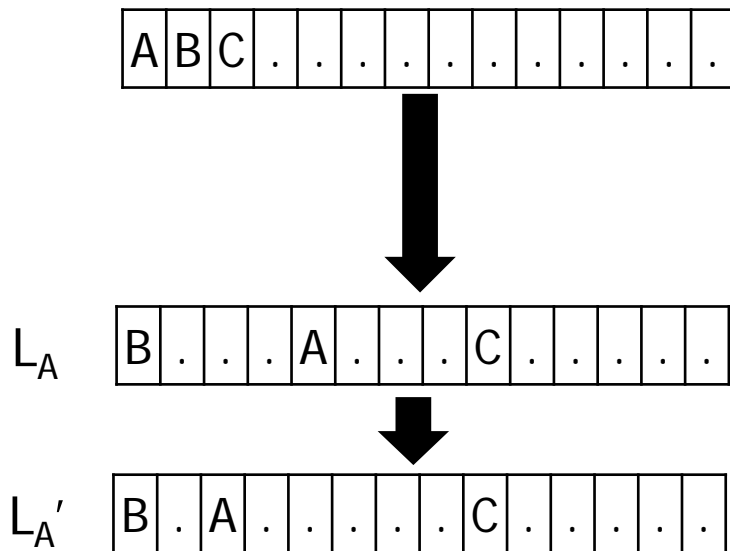
- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
 - Goal and idea
 - Preliminaries
 - A short proof

Inversions

- Let L and L' be permutation of the set $\{1, 2, \dots, n\}$
- Definition
 - A pair (i, j) is called an *inversion of L and L'* iff i and j are in different order in L than in L' (for $1 \leq i, j \leq n$ and $i \neq j$)
 - The number of inversions between L and L' is denoted by $inv(L, L')$
- Remarks
 - Different order: Once i before j , once i after j
 - Obviously, $inv(L, L') = inv(L', L)$
 - Example: $inv(\{4, 3, 1, 5, 7, 2, 6\}, \{3, 6, 2, 5, 1, 4, 7\}) = 12$
- Without loss of generality, we assume that $L = \{1, \dots, n\}$
 - Because we only look at changes in number of inversions and not at the actual set of inversions

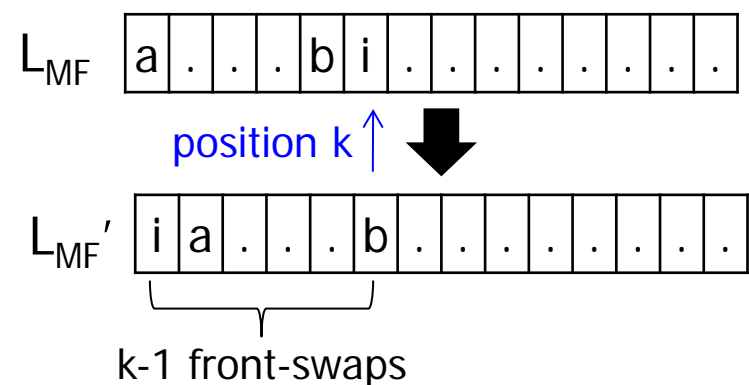
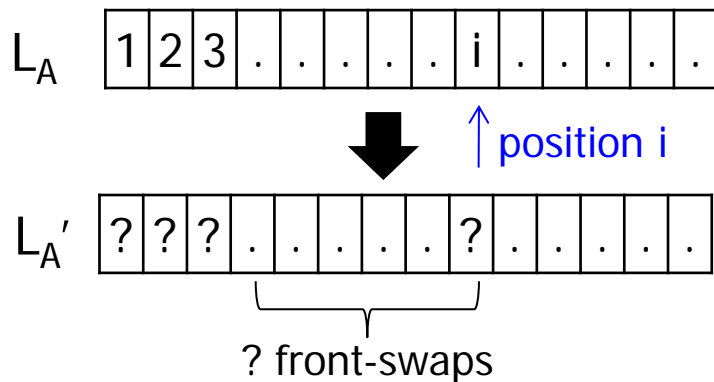
Sequences of Changes

- Assume we applied $I-1$ steps of S on L , creating L_{MF} using MF and L_A using A
- Let us consider the **next step I** , creating L_{MF}' and L_A'



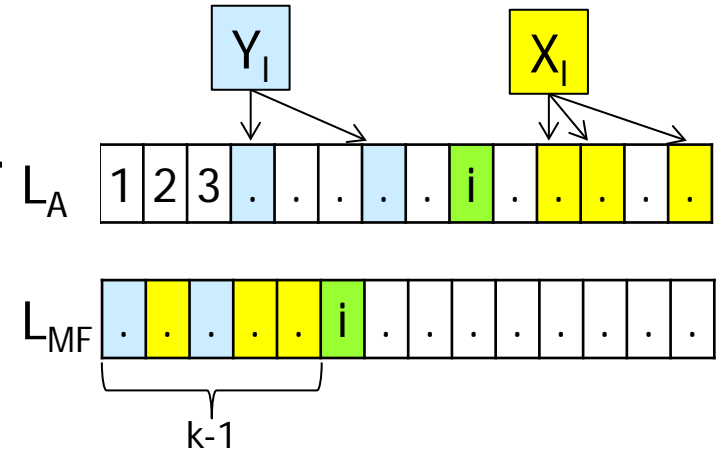
Inversion Changes

- How does **I** change the number of inv's between L_{MF} / L_A ?
- Can we compute $\text{inv}(L_{MF}', L_A')$ from $\text{inv}(L_{MF}, L_A)$?
 - Assume step I accesses element i from L_A
 - We may assume it is at position i
 - Let this element i be at position k in L_{MF}
 - Access in L_A costs i , access in L_{MF} costs k
 - After step I, A performs an unknown number of swaps; **MF** performs exactly $k-1$ front-swaps

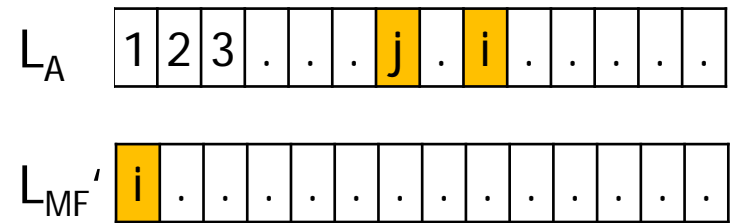


Counting Inversion Changes 1

- Let X_i be the set of values that are **before position k in L_{MF}** and **after position i in L_A**
- Let Y_i be the values **before position k in L_{MF}** and **before i in L_A**
 - Clearly, $|X_i| + |Y_i| = k-1$
- All pairs (i, c) with $c \in X_i$ are inversions between L_A and L_{MF}
 - There may be more; but only those with i are affected in this step
- After step I , MF moves element i to the front
 - Assume first that A does simply nothing
 - All inversions (i, c) with $c \in X_i$ disappear (there are $|X_i|$ many)
 - But $|Y_i| = k-1 - |X_i|$ new inversions appear
 - Thus: $\text{inv}(L_{MF}', L_A') = \text{inv}(L_{MF}, L_A) - |X_i| + k-1 - |X_i|$
 - But **A does something**



Counting Inversion Changes 2



- In step I, let A perform $F_A(I)$ front-swaps and $X_A(I)$ back-swaps
- Every front-swap (swapping i before any j) in L_A decreases $\text{inv}(L_{MF'}, L_A')$ by 1
 - Before step I, j must be before i in L_A (it is a front-swap), but after i in $L_{MF'}$ (because i now is the first element in $L_{MF'}$)
 - After step I, i is before j in both L_A' and $L_{MF'}$ – inversion removed
- Equally, every back-swap increases $\text{inv}(L_{MF'}, L_A')$ by 1
- Together: After step I, we have

$$\text{inv}(L_{MF'}, L_A') = \underbrace{\text{inv}(L_{MF}, L_A)}_{\text{Before step I}} - \underbrace{|X_I| + k-1-|X_I|}_{\text{through MF}} - \underbrace{F_A(I) + X_A(I)}_{\text{through A}}$$

Was $c_1 \dots$ was $d_1 \dots$ we switch to Cor notation

Amortized Costs

- Let $t_{MF}(l)$ be the real cost of strategy MF for step l
- We use the number of inversions as **potential function** $\Phi(L_A, L_{MF}) = \text{inv}(L_A, L_{MF})$ on the **pair** L_A, L_{MF}
- Definition
 - The *amortized costs of step l , called a_l* , are
$$a_l = t_{MF}(l) + \text{inv}(L_A(l), L_{MF}(l)) - \text{inv}(L_A(l-1), L_{MF}(l-1))$$
 - Accordingly, the *amortized costs of sequence S , $|S|=m$* , are
$$\sum a_l = \sum t_{MF}(l) + \text{inv}(L_A(m), L_{MF}(m)) - \text{inv}(L_A(0), L_{MF}(0))$$
- This is a proper potential function
 - 1: Φ depends on a property of the pair L_A, L_{MF}
 - 2: $\text{inv}()$ can never be negative, so $\forall l: \Phi(L_A(l), L_{MF}(l)) \geq \Phi(L, L) = 0$
- Let's look at **how operations change the potential**

Content of this Lecture

- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
 - Goal and idea
 - Preliminaries
 - **A short proof** (after much preparatory work)

Putting it Together

- We know for every step I from workload S accessing i :
 $\text{inv}(L_{MF'}, L_{A'}) = \text{inv}(L_{MF}, L_A) - |X_I| + k - 1 - |X_I| - F_A(I) + X_A(I)$
and thus
 $\text{inv}(L_{MF'}, L_{A'}) - \text{inv}(L_{MF}, L_A) = -|X_I| + k - 1 - |X_I| - F_A(I) + X_A(I)$
- Since $t_{MF}(I) = k$, we get **amortized costs** of
$$\begin{aligned} a_i &= t_{MF}(I) + \text{inv}(L_{A'}, L_{MF'}) - \text{inv}(L_A, L_{MF}) \\ &= k - |X_I| + k - 1 - |X_I| - F_A(I) + X_A(I) \\ &= 2(k - |X_I|) - 1 - F_A(I) + X_A(I) \end{aligned}$$
- Recall that Y_I ($|Y_I| = k - 1 - |X_I|$) are those elements before i in both lists. This implies that $k - 1 - |X_I| \leq i - 1$ or $k - |X_I| \leq i$
 - There can be at most $i - 1$ elements before position i in L_A
- Therefore: $a_i \leq 2i - 1 - F_A(I) + X_A(I)$

Putting it Together

- This is the **central trick!**
- Because we only looked at inversions (and hence the sequence of values), we can draw a connection between the **value that is accessed** and the **number of inversions** that are affected
- Recall that Y_i ($|Y_i|=k-1-|X_i|$) are those elements before i in both lists. This implies that $k-1-|X_i| \leq i-1$ or $k-|X_i| \leq i$
 - There can be at most $i-1$ elements before position i in L_A
- Therefore: $a_i \leq 2i - 1 - F_A(I) + X_A(I)$

Aggregating

- We also know the **cost of accessing i using A** : that's a_i
- Together: $a_i \leq 2C_A(i) - 1 - F_A(i) + X_A(i)$
- Aggregating this inequality **over all a_i in S** , we get

$$\sum a_i \leq 2 * C_A(S) - |S| - F_A(S) + X_A(S)$$

- By definition, we also have

$$\sum a_i = \sum t_{MF}(i) + \text{inv}(L_A^m, L_{MF}^m) - \text{inv}(L_A^0, L_{MF}^0)$$

- Since $\sum t_{MF}(i) = C_{MF}(S)$ and $\text{inv}(L_A^0, L_{MF}^0) = 0$, we get

$$C_{MF}(S) + \text{inv}(L_A^m, L_{MF}^m) \leq 2 * C_A(S) - |S| - F_A(S) + X_A(S)$$

- It finally follows ($\text{inv}() \geq 0$)

$$C_{MF}(S) \leq 2 * C_A(S) - |S| - F_A(S) + X_A(S)$$

Summary

- Self-organization creates a type of problem we were not confronted with before
 - Things change during program execution
 - But not at random – we follow a strategy
- Analysis is none-trivial, but
 - Helped to find a elegant and surprising conjecture
 - Very interesting in itself: We showed relationships between measures we never counted (and could not count easily)
 - But beware the assumptions (e.g., only single swaps)
 - Original work: Sleator, D. D. and Tarjan, R. E. (1985). "Amortized efficiency of list update and paging rules." *Communications of the ACM* 28(2): 202-208.