COMP 7720 - Online Algorithms

List Update & Compression

Shahin Kamali

University of Manitoba
List Update Problem
Problem Statement

List Accessing Problem

- The input is a set of requests to items in a list.
- The cost of accessing an item in index $i$ is $i$.

< d b b d c a c >

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e}
\end{array}
\]
Problem Statement

List Accessing Problem

- The input is a set of requests to items in a list.
- The cost of accessing an item in index $i$ is $i$.

\[\langle d \ b \ b \ d \ c \ a \ c \rangle\]

\[\text{cost: } 4\]

```
\begin{array}{cccccc}
a & b & c & d & e \\
\end{array}
```
Problem Statement

List Accessing Problem

- The input is a set of requests to items in a list.
- The cost of accessing an item in index $i$ is $i$.

$$< d \ b \ b \ d \ c \ a \ c >$$

**cost:** 4 + 2

```
 a ──── b ──── c ──── d ──── e
```
Problem Statement

List Accessing Problem

- The input is a set of requests to items in a list.
- The cost of accessing an item in index $i$ is $i$.

```
< d b b d c a c >
cost: 4+2+2
```

```
< d b b d c a c >
cost: 4+2+2
```
The input is a set of requests to items in a list.

The cost of accessing an item in index $i$ is $i$.

$$< d \ b \ b \ d \ c \ a \ c >$$

cost: $4 + 2 + 2 + 4$
**Problem Statement**

**List Accessing Problem**

- The input is a set of *requests* to items in a list.
- The cost of accessing an item in index $i$ is $i$.

\[
< \text{d b b d c a c} >
\]

**cost:** $4 + 2 + 2 + 4 + 3$
The input is a set of *requests* to items in a list.

The cost of accessing an item in index $i$ is $i$.

\[<d\ b\ b\ d\ c\ a\ c>\]
\[\text{cost: } 4+2+2+4+3+1\]
List Accessing Problem

- The input is a set of requests to items in a list.
- The cost of accessing an item in index $i$ is $i$.

\[ \text{cost: } 4+2+2+4+3+1+3 = 19 \]
Introduction to List Update

- An instance of self-adjusting data structures.
- The structure adjusts itself based on the input queries.
Introduction to List Update

- An instance of **self-adjusting data structures**.
- The structure adjusts itself based on the input queries.
- List update was formulated in 1984 by Sleator and Tarjan
  - This result of Sleator and Tarjan made online algorithms popular in the following two decades
  - There are applications in data-compression!
Problem Statement

**Self-Adjusting Lists**

- Update a list of length $k$ to adjust it to the patterns in the input sequence of length $n$ ($n \gg k$).
Update a list of length $k$ to adjust it to the patterns in the input sequence of length $n$ ($n \gg k$).

- Free exchanges: Move a requested item closer to the front without any cost.

$< d \ b \ b \ d \ c \ a \ c >$

Cost: 4
Self-Adjusting Lists

- Update a list of length $k$ to adjust it to the patterns in the input sequence of length $n$ ($n \gg k$).
  - Free exchanges: Move a requested item closer to the front without any cost.

< d b b d c a c >
cost: 4

Diagram:
```
a -> d -> b -> c -> e
```
Problem Statement

Self-Adjusting Lists

- Update a list of length $k$ to adjust it to the patterns in the input sequence of length $n$ ($n \gg k$).
  - Free exchanges: Move a requested item closer to the front without any cost.
  - Paid exchanges: Swap positions of two consecutive items with a cost 1.

\[
\langle d \, b \, b \, d \, c \, a \, c \rangle
\]

\[
\text{cost: } 4
\]
Problem Statement

Self-Adjusting Lists

Update a list of length $k$ to adjust it to the patterns in the input sequence of length $n$ ($n \gg k$).

- Free exchanges: Move a requested item closer to the front without any cost.
- Paid exchanges: Swap positions of two consecutive items with a cost 1.

< d b b d c a c >

Cost: 4
Problem Statement

List Update Problem

- In the offline version of the problem, you have access to the whole set at the beginning.
  - The problem is NP-hard.
List Update Problem

- In the offline version of the problem, you have access to the whole set at the beginning.
  - The problem is NP-hard.

- In the online setting, the requests appear in an online, sequential manner.
  - An online algorithm should reorder the list without looking at the future requests.
Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

\[
\text{cost: 4}
\]

< d b b d c a c >
Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

< d b b d c a c >

cost: 4
Online Algorithms for List Update

Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

< d b b d c a c >

cost: 4+3
Online Algorithms for List Update

Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

\[
< \text{d b b d c a c} >
\]

cost: \(4 + 3\)
Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

\[ <d \, b \, b \, d \, c \, a \, c > \]

**cost:** \(4+3+1\)
Online Algorithms for List Update

Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

< d b b d c a c >

cost: 4 + 3 + 1 + 2
After each access, move the requested item to the front.

- It only uses free exchanges.

\[
< d \ b \ b \ d \ c \ a \ c > \\
\text{cost:} \quad 4+3+1+2+4
\]
Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

< d b b d c a c >

Cost: 4 + 3 + 1 + 2 + 4 + 4
Online Algorithms for List Update

Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

\[
\langle d\ b\ b\ d\ c\ a\ c \rangle
\]

cost: \[4 + 3 + 1 + 2 + 4 + 4 + 2\]
Online Algorithms for List Update

Move-To-Front (MTF)

- After each access, move the requested item to the front.
- It only uses free exchanges.

< d b b d c a c >

cost: 4+3+1+2+4+4+2
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

Do nothing if such an item $y$ does not exist.

$\langle d \ b\ b\ d\ c\ a\ c \rangle$

cost: 4
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

- Do nothing if such an item $y$ does not exist.

\[< \text{d b b d c a c}>\]

cost: 4+2
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

Do nothing if such an item $y$ does not exist.

$< \text{d b b d c a c} >$

Cost: $4 + 2 + 2$
After an access to x, move x to the front of the first item y which has been requested at most once since the last access to x.

- Do nothing if such an item y does not exist.

\[
< d \ b \ b \ d \ c \ a \ c >
\]

\[\text{cost: } 4+2+2\]
Online Algorithms

After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

Do nothing if such an item $y$ does not exist.

$<d\ b\ b\ d\ c\ a\ c>$

cost: $4+2+2+4$
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

- Do nothing if such an item $y$ does not exist.

\[
\langle d\ b\ b\ d\ c\ a\ c \rangle
\]

\text{cost: } 4+2+2+4
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

Do nothing if such an item $y$ does not exist.

\[ \langle d \ b \ b \ d \ c \ a \ c \rangle \]
\[ \text{cost: } 4+2+2+4+4 \]
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

Do nothing if such an item $y$ does not exist.

< d b b d c a c >

cost: 4+2+2+4+4+3
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

- Do nothing if such an item $y$ does not exist.

$\langle d\ b\ b\ d\ c\ a\ c \rangle$

Cost: $4+2+2+4+4+3+4$
After an access to $x$, move $x$ to the front of the first item $y$ which has been requested at most once since the last access to $x$.

- Do nothing if such an item $y$ does not exist.

< d b b d c a c >

cost: 4+2+2+4+4+3+4
Online Algorithms

Optimal Static Algorithm

- Look at the sequence of requests, sort items by the frequency of their accesses.
  - The most accessed item will be at the beginning of the list.
Optimal Static Algorithm

- Look at the sequence of requests, sort items by the frequency of their accesses.
  - The most accessed item will be at the beginning of the list.
- The cost of the algorithm would be at most $nk/2$. 
Lower Bound for Competitive Ratio

- Consider a **cruel** sequence in which the adversary always asks for the last item in the list!
- What will be the cost of the algorithm?
Consider a cruel sequence in which the adversary always asks for the last item in the list!

What will be the cost of the algorithm?

- It will be $nk$. 

What is the cost of $Opt$?

We know the optimal static algorithm has a cost of $n(k + 1)/2$. So the cost of $Opt$ is no more than $n(k + 1)/2$.

The competitive ratio of any online list update algorithm is at least $nk/nk = 2$. 
Consider a **cruel** sequence in which the adversary always asks for the last item in the list!

- What will be the cost of the algorithm?
  - It will be $nk$.

- What is the cost of $OPT$?
  - We know the optimal static algorithm has a cost of $n(k + 1)/2$.
  - So the cost of $OPT$ is no more than $n(k + 1)/2$. 

Consider a cruel sequence in which the adversary always asks for the last item in the list!

What will be the cost of the algorithm?
- It will be $nk$.

What is the cost of $OPT$?
- We know the optimal static algorithm has a cost of $n(k + 1)/2$.
- So the cost of $OPT$ is no more than $n(k + 1)/2$.

The competitive ratio of any online list update algorithm is at least $\frac{nk}{nk/2} = 2$. 
Competitiveness of MTF
On the Nature of Opt

There is an optimal algorithm that only uses paid exchanges!
On the Nature of Opt

- There is an optimal algorithm that only uses paid exchanges!
- Assume $OPT$ uses a free exchange after accessing item $x$ at position $i$ to move it closer to the front to position $j$
  - The cost will be $i$. 

![Diagram showing the movement of an item from position $i$ to position $j$.]
On the Nature of Opt

- There is an optimal algorithm that only uses paid exchanges!
- Assume \( \text{OPT} \) uses a free exchange after accessing item \( x \) at position \( i \) to move it closer to the front to position \( j \)
  - The cost will be \( i \).

In a new scheme, before the access, apply \( i - j \) paid exchanges to move \( i \) to position \( j \).
  - The new cost will be \( i - j \) for paid exchanges and \( j \) for the access, which sums to \( i \).
Theorem

Move-To-Front has competitive ratio of 2.

- We prove it through potential function method
  - And it takes a few slides :'-)
Competitiveness of MTF

Inversions

At a given time, two items $x$ and $y$ form an inversion if their relative order is different in the lists of MTF and OPT.

**Question:** what is the maximum number of inversions for a list of length $k$?

- (a) $k/2$
- (c) $k(k - 1)/2$
- (c) $k$
- (d) $k^2 - k/2$

On the diagram: for the list of MTF, the sequence is $a \rightarrow d \rightarrow c \rightarrow e \rightarrow b$, and for the list of OPT, the sequence is $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$. 
Assume MTF and $OPT$ are running the same input in parallel.

Assume we are at the $t$'th request, and there is a request to an item $x$.

Define the potential at time $t$ to be the total number of inversions before accessing $x$. 
Assume MTF and OPT are running the same input in parallel.

Assume we are at the $t$’th request, and there is a request to an item $x$.

Define the potential at time $t$ to be the total number of inversions before accessing $x$.

Inversions are $(b, c), (b, d), (b, e), (c, d)$

So, $\Phi(t) = 4$. 
Intuitively, if we have a high potential, we are in bad state.

**Amortized Cost**

Define the amortized cost at time $t$ (when answering the $t$th request) as:

$$\text{amortized cost} (t) = \text{actual cost} (t) + \Phi(t+1) - \Phi(t)$$

Example: assume at time $t$, there is a request to $b$, and Opt does not rearrange the list for accessing $t$.

For MTF, we have $\text{actual cost} (t) = 5$, $-\Phi(t) = 4$ and $\Phi(t+1) = 2$. The amortized cost is $5 + 2 - 4 = 3$. 
Intuitively, if we have a high potential, we are in bad state.

Define the **amortized cost** at time \( t \) (when answering the \( t \)th request) as:

\[
amortized\_cost(t) = \text{actual}\_cost(t) + \Phi(t + 1) - \Phi(t)
\]
Intuitively, if we have a high potential, we are in bad state.

Define the amortized cost at time $t$ (when answering the $t$th request) as:

$$amortized\_cost(t) = actual\_cost(t) + \Phi(t + 1) - \Phi(t)$$

Example: assume at time $t$, there is a request to $b$, and $OPT$ does not rearrange the list for accessing $t$.
Intuitively, if we have a high potential, we are in bad state.

Define the **amortized cost** at time \( t \) (when answering the \( t \)th request) as:

\[
amortized\_cost(t) = \text{actual\_cost}(t) + \Phi(t + 1) - \Phi(t)
\]

Example: assume at time \( t \), there is a request to \( b \), and \( \text{OPT} \) does not rearrange the list for accessing \( t \).
Intuitively, if we have a high potential, we are in bad state.

Define the **amortized cost** at time $t$ (when answering the $t$th request) as:

$$amortized\_cost(t) = actual\_cost(t) + \Phi(t + 1) - \Phi(t)$$

Example: assume at time $t$, there is a request to $b$, and $OPT$ does not rearrange the list for accessing $t$.

- For MTF, we have $actual\_cost(t) = 5$, $-\Phi(t) = 4$ and $\Phi(t + 1) = 2$.
- $amortized\_cost$ is $5 + 2 - 4 = 3$. 

\[\text{list of MTF} \quad \begin{array}{c}
\text{t} \quad a \rightarrow d \rightarrow c \rightarrow e \rightarrow b
\end{array} \quad \begin{array}{c}
(b,c) \quad (b,d) \\
(b,e) \quad (c,d)
\end{array} \quad \text{list of Opt} \quad \begin{array}{c}
\text{t} \quad a \rightarrow b \rightarrow c \rightarrow d \rightarrow e
\end{array} \quad \begin{array}{c}
\text{t+1} \quad b \rightarrow a \rightarrow d \rightarrow c \rightarrow e
\end{array} \quad \begin{array}{c}
\text{t+1} \quad b \rightarrow a \rightarrow d \rightarrow c \rightarrow e
\end{array} \quad \begin{array}{c}
\text{t+1} \quad \text{X} \quad \text{X} \\
\text{X} \quad \text{X} \quad \text{X} \quad (a,b)\end{array}\]
Lemma

At any time $t$, $amortized\_cost(t) \leq 2\,OPT(t)$, i.e., the amortized cost of MTF for the $t$'th request is at most twice that of $OPT$. 
How many inversions are removed by moving $x$ to front?

- Before moving to front, there are $i - 1$ items before $x$ in MTF list.
- At most $j - 1$ of them can also appear before $x$ in $\text{OPT}$ list (are non-inversions) $\Rightarrow$ the rest, at least, $i - 1 - (j - 1) = i - j$ are inversions $\Rightarrow$ By moving to front at least $i - j$ inversions are removed.
How many inversions are added by moving $x$ to front?

- $x$ is in front of MTF list after the move and at position $j$ of $OPT$’s list
- Items that appear after $x$ in MTF and before $x$ in $OPT$ are at most $j - 1$
- At most $j - 1$ inversions are added
When moving $x$ to front:

- Actual cost is $i$, at least $i - j$ inversions are removed, at most $j - 1$ inversions are added.
Potential Function Method (cntd.)

- When moving $x$ to front:
  - Actual cost is $i$, at least $i - j$ inversions are removed, at most $j - 1$ inversions are added.

- Assume $\text{OPT}$ makes $k$ paid exchanges.
  - Recall that it does no free exchange.
  - The cost of $\text{OPT}$ will be $j + k$.
  - Each paid exchange increases potential by 1 $\rightarrow$ potential increases by at most $k$. 

Amortized cost $= \text{actual cost} + \Phi(t + 1) - \Phi(t) \leq i + 2j + k - i - 1 = 2j + k - 1$.

Cost of $\text{OPT}$ is $j + k$ and amortized cost is less than $2j + k$. 

Lemma: At any time $t$, amortized cost $(t) < 2 \cdot \text{OPT}(t)$. 

When moving \( x \) to front:
- Actual cost is \( i \), at least \( i - j \) inversions are removed, at most \( j - 1 \) inversions are added
- Assume \( \text{OPT} \) makes \( k \) paid exchanges.
  - Recall that it does no free exchange.
  - The cost of \( \text{OPT} \) will be \( j + k \).
  - Each paid exchange increases potential by 1 \( \rightarrow \) potential increases by at most \( k \).

\[
\Phi(t + 1) - \Phi(t) = \text{added inversions} - \text{removed inversions} \\
(j + k - 1) - (i - j) = 2j + k - i - 1.
\]
When moving $x$ to front:

- Actual cost is $i$, at least $i - j$ inversions are removed, at most $j - 1$ inversions are added

Assume $OPT$ makes $k$ paid exchanges.

- Recall that it does no free exchange.
- The cost of $OPT$ will be $j + k$.
- Each paid exchange increases potential by 1 $\rightarrow$ potential increases by at most $k$.

$$\Phi(t + 1) - \Phi(t) = \text{added inversions} - \text{removed inversions} \leq (j + k - 1) - (i - j) = 2j + k - i - 1.$$ 

$$\text{amortized cost} = \text{actual cost} + \Phi(t + 1) - \Phi(t) \leq i + 2j + k - i - 1 = 2j + k - 1$$
Potential Function Method (cntd.)

- When moving $x$ to front:
  - Actual cost is $i$, at least $i - j$ inversions are removed, at most $j - 1$ inversions are added.

- Assume $OPT$ makes $k$ paid exchanges.
  - Recall that it does no free exchange.
  - The cost of $OPT$ will be $j + k$.
  - Each paid exchange increases potential by 1 → potential increases by at most $k$.

- $\Phi(t+1) - \Phi(t) = \text{added inversions} - \text{removed inversions} \leq (j + k - 1) - (i - j) = 2j + k - i - 1$.

- $\text{amortized cost} = \text{actual cost} + \Phi(t+1) - \Phi(t) \leq i + 2j + k - i - 1 = 2j + k - 1$

- Cost of $Opt$ is $j + k$ and $\text{amortized cost}$ is less than $2j + k$.

**Lemma**

At any time $t$, $\text{amortized cost}(t) < 2 \cdot OPT(t)$. 
A Quick Example

Assume at time $t$:
- the list of MTF is
  $$8 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$$
- the list of OPT is
  $$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$$

Assume $x$ is 3, which means $i = 6$ and $j = 3$.

The number of removed inversions in this case is at least $i - j = 3$.
In fact, it turns out to be 5 because all 1, 2, 4, 5, 6, 7, 8 form inversions with 3 which will be removed by moving 3 to the front.

The number of new inversions will be at most $j - 1 = 2$. In fact, it is 0 as no new inversion is added.
For the cost of MTF, we have

$$MTF = actual\_cost(1) + actual\_cost(2) + \ldots + actual\_cost(n)$$

$$= (actual\_cost(1) + \Phi(2) - \Phi(1))$$
$$+ (actual\_cost(2) + \Phi(3) - \Phi(2))$$
$$+ \ldots$$
$$+ (actual\_cost(n) + \Phi(n + 1) - \Phi(n)) - (\Phi(n + 1) - \Phi(1))$$

$$= \text{amortized}\_\text{cost}(1) + \ldots + \text{amortized}\_\text{cost}(n) - (\Phi(n + 1) - \Phi(1))$$

$$< 2\text{Opt}(1) + \ldots + 2\text{OPT}(n) - O(L^2) \approx 2\text{Cost\_Opt}(n)$$

$$\{\text{recall that} \quad n \gg L\}$$

Note that in the second line, we just added and removed values (i.e., we added
$$\Phi(1) - \Phi(1) + \Phi(2) - \Phi(2) + \ldots + \Phi(n + 1) - \Phi(n + 1) = 0$$).
Competitiveness of MTF

Theorem

*Competitive ratio of $\text{MTF}$ is at most 2*

- No deterministic algorithm can have a competitive ratio better than 2.
  - MTF is an optimal list-update algorithm.
  - Timestamp is another optimal deterministic algorithm.
Theorem

*Competitive ratio of MTF is at most 2*

- No deterministic algorithm can have a competitive ratio better than 2.
  - MTF is an optimal list-update algorithm.
  - Timestamp is another optimal deterministic algorithm.
- Later, we will see randomized algorithms that achieve better competitive ratios.
Competitiveness of MTF

Theorem

*Competitive ratio of MTF is at most 2*

- No deterministic algorithm can have a competitive ratio better than 2.
  - MTF is an optimal list-update algorithm.
  - Timestamp is another optimal deterministic algorithm.

- Later, we will see randomized algorithms that achieve better competitive ratios.

- Potential function method is a general framework for analysis of many online algorithms!
Assume you face an online problem where the input is a sequence of requests that require you to change the state of a problem.

- These states should be finite and independent of the input length!
Competitiveness of MTF

Potential Function Framework

- Assume you face an online problem where the input is a sequence of requests that require you to change the state of a problem.
  - These states should be finite and independent of the input length!
- Define a ‘potential’ as a function of the state of the algorithm and that of \( \text{OPT} \) (e.g. no. inversions).
  - This is the critical part :-)
Potential Function Framework

Assume you face an online problem where the input is a sequence of requests that require you to change the state of a problem.

- These states should be finite and independent of the input length!

Define a ‘potential’ as a function of the state of the algorithm and that of $OPT$ (e.g. no. inversions).

- This is the critical part :-)

Define the amortized cost at a given time $t$ as the actual cost algorithm plus the difference in potential after the request is served (same for all problems).
Assume you face an online problem where the input is a sequence of requests that require you to change the state of a problem.

These states should be finite and independent of the input length!

Define a ‘potential’ as a function of the state of the algorithm and that of $\text{OPT}$ (e.g. no. inversions).

This is the critical part :-)

Define the amortized cost at a given time $t$ as the actual cost algorithm plus the difference in potential after the request is served (same for all problems).

The potential should be defined in a way so that you can show $amortized\_cost(t) \leq c \text{OPT}(t)$. 
Competitiveness of MTF

Potential Function Framework

- Assume you face an online problem where the input is a sequence of requests that require you to change the state of a problem.
  - These states should be finite and independent of the input length!
- Define a ‘potential’ as a function of the state of the algorithm and that of \( \text{OPT} \) (e.g. no. inversions).
  - This is the critical part :-)
- Define the amortized cost at a given time \( t \) as the actual cost algorithm plus the difference in potential after the request is served (same for all problems).
- The potential should be defined in a way so that you can show \( \text{amortized cost}(t) \leq c \text{OPT}(t) \).
- Using a telescopic sum, the competitive ratio will be at most \( c \) (same for all problems).
Competitiveness of MTF

Potential Function Method

Did we survive?
Transpose Algorithm

- **Transpose**: Move the accessed item one unit closer to the front.

- **Question**: What is the competitive ratio of Transpose for a list of length \( m \)?
  
  (a) 1.5  
  (b) 2  
  (c) \( \Theta(m) \)  
  (d) \( \Theta(m^2) \)
Competitiveness of MTF

Transpose Algorithm

- **Transpose:** Move the accessed item one unit closer to the front.
- **Question:** What is the competitive ratio of Transpose for a list of length $m$?
  - (a) 1.5
  - (b) 2
  - (c) $\Theta(m)$
  - (d) $\Theta(m^2)$

sequence: $a_m$

$\overrightarrow{a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{m-1} \rightarrow a_m}$
Transpose Algorithm

- **Transpose**: Move the accessed item one unit closer to the front.
- **Question**: What is the competitive ratio of Transpose for a list of length $m$?
  - (a) 1.5
  - (b) 2
  - (c) $\Theta(m)$
  - (d) $\Theta(m^2)$

Sequence: $(a_m, a_{m-1})$
Transpose Algorithm

- **Transpose**: Move the accessed item one unit closer to the front.
- **Question**: What is the competitive ratio of Transpose for a list of length $m$?
  - (a) 1.5
  - (b) 2
  - (c) $\Theta(m)$
  - (d) $\Theta(m^2)$

  $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{m-1} \rightarrow a_m$

  sequence: $(a_m \ a_{m-1})^k$

- The cost of Transpose after $n$ requests on a list of length $m$ will be $n \cdot m$

- What does $\text{OPT}$ do?
Transpose Algorithm

- **Transpose**: Move the accessed item one unit closer to the front.
- **Question**: What is the competitive ratio of Transpose for a list of length \( m \)?
  
  \[
  \text{(a) 1.5} \quad \text{(b) 2} \quad \text{(c) } \Theta(m) \quad \text{(d) } \Theta(m^2)
  \]

  sequence: \((a_m, a_{m-1})^k\)

- The cost of Transpose after \( n \) requests on a list of length \( m \) will be \( n \cdot m \).
- What does OPT do?
  - It moves \( a_m \) and \( a_{m-1} \) to the front using \( 2m - 3 \) paid exchanges, and does not move them after.
  - The cost for accesses to \( a_{m-1} \) and \( a_m \) are respectively 2 and 1.
  - The cost of OPT will be \( 2m - 3 + n/2 \cdot 1 + n/2 \cdot 2 \approx 1.5n + 2m \).
Competitiveness of MTF

Transpose Algorithm

- **Transpose**: Move the accessed item one unit closer to the front.
- **Question**: What is the competitive ratio of Transpose for a list of length \( m \)?
  
  \[
  \begin{align*}
  \text{(a) } & 1.5 \quad \text{(b) } 2 \quad \text{(c) } \Theta(m) \quad \text{(d) } \Theta(m^2)
  \end{align*}
  \]

  sequence: \((a_m a_{m-1})^k\)

- The cost of Transpose after \( n \) requests on a list of length \( m \) will be \( n \cdot m \).
- **What does OPT do?**
  - It moves \( a_m \) and \( a_{m-1} \) to the front using \( 2m - 3 \) paid exchanges, and does not move them after.
  - The cost for accesses to \( a_{m-1} \) and \( a_m \) are respectively 2 and 1.
  - The cost of OPT will be \( 2m - 3 + n/2 \cdot 1 + n/2 \cdot 2 \approx 1.5n + 2m \).
- The competitive ratio will be at least \( \frac{n \cdot m}{1.5n+2m} \in \Theta(m) \).
Consider an algorithm that moves a requested item at index \( i \) half way to front.
Consider an algorithm that moves a requested item at index \( i \) half way to front.

What is the competitive ratio of this algorithm?
Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

\[ a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \]

sequence: $a_{2m}$
Consider an algorithm that moves a requested item at index \( i \) half way to front.

What is the competitive ratio of this algorithm?

\[
a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{2m} \rightarrow a_{m+1} \rightarrow \ldots \rightarrow a_{2m-2} \rightarrow a_{2m-1}
\]

sequence: \( a_{2m} \; a_{2m-1} \)
Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

$$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{2m-1} \rightarrow a_{2m} \rightarrow \ldots \rightarrow a_{2m-3} \rightarrow a_{2m-2}$$

Sequence: $a_{2m} \ a_{2m-1} \ a_{2m-2}$
Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

sequence: $a_{2m} \ a_{2m-1} \ a_{2m-2} \ldots \ a_{m+1}$
Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

\[
a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m}
\]

sequence: \((a_{2m} \ a_{2m-1} \ a_{2m-2} \ldots \ a_{m+1})^k\)
Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

sequence: $(a_{2m} \ a_{2m-1} \ a_{2m-2} \cdots a_{m+1})^k$

→ Alg’s cost after $n$ requests is $n \cdot 2m$
Consider an algorithm that moves a requested item at index $i$ halfway to front.

What is the competitive ratio of this algorithm?

$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m}$

sequence: $(a_{2m} \ a_{2m-1} \ a_{2m-2} \ldots \ a_{m+1})^k$

$\rightarrow$ Alg’s cost after $n$ requests is $n \cdot 2m$

$OPT$ moves the requested $m$ items to the front (using $O(m^2)$ paid exchanges at the beginning) and does not move after that.

$a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \rightarrow a_1 \rightarrow a_2 \ldots \rightarrow a_m$
Other Deterministic Algorithms

Consider an algorithm that moves a requested item at index $i$ half way to front.

What is the competitive ratio of this algorithm?

\[ a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \]

sequence: \((a_{2m} \ a_{2m-1} \ a_{2m-2} \ldots a_{m+1})^k\)

→ Alg’s cost after $n$ requests is $n \cdot 2m$

OPT moves the requested $m$ items to the front (using $O(m^2)$ paid exchanges at the beginning) and does not move after that.

\[ a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \rightarrow a_1 \rightarrow a_2 \ldots \rightarrow a_m \]

Opt’s cost after $n$ requests is $n \cdot m/2 + O(m^2)$. 
Consider an algorithm that moves a requested item at index \( i \) half way to front.

What is the competitive ratio of this algorithm?

\[
\begin{align*}
    a_1 & \rightarrow a_2 \rightarrow \ldots \rightarrow a_m \rightarrow a_{m+1} \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \\
\end{align*}
\]

sequence: \((a_{2m} \ a_{2m-1} \ a_{2m-2} \ldots a_{m+1})^k\)

\(\rightarrow\) Alg’s cost after \( n \) requests is \( n \cdot 2m \)

Opt moves the requested \( m \) items to the front (using \( O(m^2) \) paid exchanges at the beginning) and does not move after that.

\[
\begin{align*}
    a_{m+1} & \rightarrow a_{m+2} \rightarrow \ldots \rightarrow a_{2m-1} \rightarrow a_{2m} \rightarrow a_1 \rightarrow a_2 \ldots \rightarrow a_m \\
\end{align*}
\]

Opt’s cost after \( n \) requests is \( n \cdot m/2 + O(m^2) \).

Theorem

The competitive ratio is at least \( \frac{n \cdot 2m}{n \cdot m/2 + O(m^2)} \approx 4 \).
Consider an algorithm that moves a requested item $x$ to the front of the list on every-other-access to $x$. The competitive ratio of this algorithm is indeed 2.5.
Consider an algorithm that moves a requested item $x$ to the front of the list on every-other-access to $x$.

The competitive ratio of this algorithm is indeed 2.5.

**Theorem**

*The best existing deterministic algorithms are Move-To-Front and Timestamp (and some algorithms which combine them). Other list update algorithms do not achieve competitive ratio of 2.*
Randomized List Update Algorithms

- What is a good randomized algorithm?
- Algorithm 1: on access to an item, move it to front with probability $1/2$. 

Randomized List Update Algorithms

- What is a good randomized algorithm?
  - Algorithm 1: on access to an item, move it to front with probability 1/2.
    - This turns out to have a competitive ratio of 2.
  - Is it good?
Consider an algorithm that maintains a bit for each item
- Initially all bits are set uniformly at random.
- On an access to $x$, flip the bit of $x$ and move it to front if the bit becomes ‘1’.
Consider an algorithm that maintains a bit for each item:

- Initially all bits are set uniformly at random.
- On an access to $x$, flip the bit of $x$ and move it to front if the bit becomes ‘1’.

Let's consider a sequence: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

The corresponding `bits` sequence is:

```
```

The sequence of items accessed is `d`.
Consider an algorithm that maintains a bit for each item

- Initially all bits are set uniformly at random.
- On an access to \( x \), flip the bit of \( x \) and move it to front if the bit becomes ‘1’.

\[
\text{sequence: } d
\]

---

**BIT Algorithm**

- Consider an algorithm that maintains a bit for each item
  - Initially all bits are set uniformly at random.
  - On an access to \( x \), flip the bit of \( x \) and move it to front if the bit becomes ‘1’.

\[
\begin{align*}
  a & \rightarrow b \rightarrow c \rightarrow d \rightarrow e \\
  \text{bits} : & a[0] \ b[1] \ c[1] \ d[1] \ e[1]
\end{align*}
\]
Consider an algorithm that maintains a bit for each item

- Initially all bits are set uniformly at random.
- On an access to $x$, flip the bit of $x$ and move it to front if the bit becomes ‘1’.

$\rightarrow d \rightarrow a \rightarrow b \rightarrow c \rightarrow e$

$bits: a[0] \ b[1] \ c[1] \ d[1] \ e[1]$

sequence: $d$
Consider an algorithm that maintains a bit for each item

- Initially all bits are set uniformly at random.
- On an access to $x$, flip the bit of $x$ and move it to front if the bit becomes ‘1’.

```
d → a → b → c → e
```

Bits: $a[0] \ b[1] \ c[1] \ d[1] \ e[1]$

Sequence: $d \ e$
Competitiveness of MTF

BIT Algorithm

Consider an algorithm that maintains a bit for each item:
- Initially all bits are set uniformly at random.
- On an access to \( x \), flip the bit of \( x \) and move it to front if the bit becomes ‘1’.

\[
\rightarrow d \rightarrow a \rightarrow b \rightarrow c \rightarrow \downarrow e
\]

bits: \( a[0] \ b[1] \ c[1] \ d[1] \ e[0] \)

sequence: \( d \ e \)
Theorem

**BIT has a competitive ratio of 1.75.**
Theorem

*BIT has a competitive ratio of 1.75.*

- The best randomized algorithm is COMB with a competitive ratio of 1.6.
  - Apply Bit with probability 0.8 and TimeStamp with probability 0.2.
Theorem

BIT has a competitive ratio of 1.75.

- The best randomized algorithm is COMB with a competitive ratio of 1.6.
  - Apply Bit with probability 0.8 and TimeStamp with probability 0.2.
- No randomized algorithm can have a competitive ratio better than 1.5.
  - There is a gap between the competitive ratio of the best algorithm (1.6 of COMB) and the best lower bound (1.5).
We use a **projective property** to prove upper bounds for competitive ratios of most algorithms.

An algorithm is projective if the relative order of any two items only depend on accesses to those items.
We use a **projective property** to prove upper bounds for competitive ratios of most algorithms.

An algorithm is projective if the relative order of any two items only depend on accesses to those items.

- **Move-To-Front**: an item $x$ appears before $y$ if and only if the last access to $x$ is more recent than the last access to $y$ (and request to another item $z$ does not change it).
We use a **projective property** to prove upper bounds for competitive ratios of most algorithms.

An algorithm is projective if the relative order of any two items only depend on accesses to those items.

- Move-To-Front: an item $x$ appears before $y$ if and only if the last access to $x$ is more recent than the last access to $y$ (and request to another item $z$ does not change it).

In order to show a projective algorithm is $c$-competitive, it suffices to show that it is $c$-competitive for lists of length 2.

- Most existing algorithms for list-update are projective.
How many bits of advice to achieve an optimal solution?

For a sequence of length \( n \), \( O(n) \) bits are sufficient and \( \Omega(n) \) are required!

- We see the proof in another lecture (hopefully).

- What about advice of small size?
Consider the following three deterministic algorithms:

- Timestamp (competitive ratio is 2)
- MTFO: Move-To-Front on every-other-access on odd accesses (competitive ratio is 2.5).
- MTFE: Move-To-Front on every-other-access on even accesses (competitive ratio is 2.5).

The better algorithm among the above algorithms has a competitive ratio of $1$. For any sequence, at least one of these algorithms is has a cost no more than $1$ times the cost of Opt. This is better than any possible deterministic algorithm.

What does it mean in terms of advice? With only two bits of advice, the better algorithm can be specified.
Consider the following three deterministic algorithms:

- Timestamp (competitive ratio is 2)
- MTFO: Move-To-Front on every-other-access on odd accesses (competitive ratio is 2.5).
- MTFE: Move-To-Front on every-other-access on even accesses (competitive ratio is 2.5).

The better algorithm among the above algorithms has a competitive ratio of 1.66.

- For any sequence, at least one of these algorithms is has a cost no more than 1.66 times the cost of \( \text{OPT} \).
- This is better than any possible deterministic algorithm.

What does it mean in terms of advice?

- With only two bits of advice, the better algorithm can be specified.
For deterministic case, Move-To-Front and TimeStamp have competitive ratio of 2, and it is the best ratio that a deterministic algorithm can have.
For deterministic case, Move-To-Front and TimeStamp have competitive ratio of 2, and it is the best ratio that a deterministic algorithm can have.

For randomized case, COMB is the best existing algorithm with competitive ratio 1.6. No randomized algorithm can be better than 1.5-competitive. There is a gap here!
For deterministic case, Move-To-Front and TimeStamp have competitive ratio of 2, and it is the best ratio that a deterministic algorithm can have.

For randomized case, COMB is the best existing algorithm with competitive ratio 1.6. No randomized algorithm can be better than 1.5-competitive. There is a gap here!

Potential Project topics:
- Continue studying advice complexity of list update algorithms
  - What algorithms complement each other in a better way that we saw for MTF, MTFE, MTFO?
For deterministic case, Move-To-Front and TimeStamp have competitive ratio of 2, and it is the best ratio that a deterministic algorithm can have.

For randomized case, COMB is the best existing algorithm with competitive ratio 1.6. No randomized algorithm can be better than 1.5-competitive. There is a gap here!

Potential Project topics:
- Continue studying advice complexity of list update algorithms
  - What algorithms complement each other in a better way that we saw for MTF, MTFE, MTFO?
- Assume we do not allow free exchanges? What will be the best algorithm? (paid exchange model)
List Update & Compression
One important application of list update is in data compression. Given a data-sequence (e.g., an English text), we want to compress it. We should be able to recover the exact text from the compressed one → **Lossless compression**
Basics of Compression

- How to encode some data (e.g., an English text)?
Basics of Compression

- How to encode some data (e.g., an English text)?
- Solution 1: write the ASCII or Unicode code for each character
  - The code for ‘A’ has the same length as ‘Q’.
How to encode some data (e.g., an English text)?

- Solution 1: write the ASCII or Unicode code for each character
  - The code for ‘A’ has the same length as ‘Q’.
- Solution 2: let more common characters have smaller length
  - In Huffman code ‘A’ is encoded shorter than ‘Q’ :)

"List Update & Compression"

"Basics of Compression"
How to encode some data (e.g., an English text)?

Solution 1: write the ASCII or Unicode code for each character
  - The code for ‘A’ has the same length as ‘Q’.

Solution 2: let more common characters have smaller length
  - In Huffman code ‘A’ is encoded shorter than ‘Q’ :) 
  - The ‘context’ is ignored: the code for ‘TH’ is longer than ‘Q’ :( 

MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[ S = \text{INEFFICIENCIES} \]

\[ C = \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[
\begin{array}{cccccccccccccccccccccccc}
\end{array}
\]

\[
S = \text{INEFFICIENCIES}
\]

\[
C = 8
\]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[ S = \text{INEFFICIENCIES} \]

\[ C = 8 \ 13 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[ S = \text{INEFFICIENCIES} \]
\[ C = 8 \ 13 \ 6 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
E N I A B C D F G H J K L M O P Q R S T U V W X Y Z

\[ S = \text{INEFFICIENCIES} \]
\[ C = 8 \ 13 \ 6 \ 7 \]
MTF Encoding

Solutions 3: use MTF index to encode the characters

\[ S = \text{INEFFICIENCIES} \]

\[ C = 8 \ 13 \ 6 \ 7 \ 0 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[ S = \text{INEFFICIENCIES} \]
\[ C = 8 \ 13 \ 6 \ 7 \ 0 \ 3 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>F</td>
<td>E</td>
<td>N</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

\[ S = \text{INEFFICIENCIES} \]

\[ C = 8 \ 13 \ 6 \ 7 \ 0 \ 3 \ 6 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

C I F E N A B D G H J K L M O P Q R S T U V W X Y Z

\[ S = \text{INEFFICIENCIES} \]

\[ C = 8 \ 13 \ 6 \ 7 \ 0 \ 3 \ 6 \ 1 \]
MTF Encoding

- Solutions 3: use MTF index to encode the characters

\[
S = \text{INEFFICIENCIES}
\]

\[
C = 8\ 13\ 6\ 7\ 0\ 3\ 6\ 1\ 3\ 4\ 3\ 3\ 3\ 3\ 18
\]

- What does a run in \( S \) encode to in \( C \)?
- This results in good compression if we have high locality in the input.
Burrows-Wheeler Transform

- Increase locality using Burrows-Wheeler Transform!
Increase locality using Burrows-Wheeler Transform!

How it works?

- Create all rotations of a given sequence.
- Sort those rotations into lexicographic order.
- Take as output the last column!
Increase locality using Burrows-Wheeler Transform!

How it works?
- Create all rotations of a given sequence.
- Sort those rotations into lexicographic order.
- Take as output the last column!

Why it is useful?
- Creates output with high locality!
- This is reversible

BWT(banana) = annb$aa
Why Burrows-Wheeler outputs have high locality?

Consider an example of English text; there are many 'the's such text.
Why Burrows-Wheeler outputs have high locality?

Consider an example of English text; there are many 'the's such text.

- When we sort, rotations starting with 'he' appear together.
- The last column for these rotations has character 't', i.e., we will have a run of t's.
Assume we want to compress a data sequence $S$.

Apply BWT on $S$ to increase its locality

$baanana\$

Then apply MTF on the BWT output and encode the indices in the list

$a \rightarrow b \rightarrow \ldots \rightarrow n \rightarrow \ldots \rightarrow z \rightarrow \ldots$

$\rightarrow 0 13 0 2 27 3 0$

You expect to see a lot of 0's and 1's.

Use run-length encoding to store these indices.

$\langle 1 1 1 1 1 2 2 2 2 1 1 4 4 4 \rangle \rightarrow \langle (1 5) (2 4) (1 2) (4 3) \rangle$
**List Update & Compression**

**B-Zip2 compression scheme**

- Assume we want to compress a data sequence $S$.
- Apply BWT on $S$ to increase its locality
  - $baanana$ $\rightarrow$ $annb$ $aa$
- Apply MTF on BWT output and encode the indices in the list
  
  $a \rightarrow b \rightarrow \ldots \rightarrow n \rightarrow \ldots \rightarrow z \rightarrow$

  $annb$ $aa$ $\rightarrow$ $0$ $13$ $0$ $2$ $27$ $3$ $0$

- You expect to see a lot of 0’s and 1’s.

$\langle 1$ $1$ $1$ $1$ $1$ $2$ $2$ $2$ $2$ $1$ $1$ $4$ $4$ $4$ $\rangle \rightarrow \langle (1$ $5)$ $(2$ $4)$ $(1$ $2)$ $(4$ $3)$ $\rangle$
B-Zip2 compression scheme

- Assume we want to compress a data sequence $S$.
- Apply BWT on $S$ to increase its locality
  
  $\text{baanana}\$ $\rightarrow$ $\text{annb}$\textit{aa}$

- Apply MTF on BWT output and encode the indices in the list

  $\text{a} \rightarrow \text{b} \rightarrow \ldots \rightarrow \text{n} \rightarrow \ldots \rightarrow \text{z} \rightarrow \$

  $\text{annb}$\textit{aa} $\rightarrow$ $0 \ 13 \ 0 \ 2 \ 27 \ 3 \ 0$

- You expect to see a lot of 0’s and 1’s.

- Use run-length encoding to store these indices
  
  Write down the length of each run!

  $\langle 1 \ 1 \ 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2 \ 1 \ 1 \ 4 \ 4 \ 4 \rangle \rightarrow \langle (1 \ 5) \ (2 \ 4) \ (1 \ 2) \ (4 \ 3) \rangle$
Decompression

- Assume we are given the indices in the compressed file
Decompression

- Assume we are given the indices in the compressed file
- Follow the steps of MTF and write down the character of each index

\[
\begin{align*}
a & \rightarrow b & \rightarrow \ldots & \rightarrow n & \rightarrow \ldots & \rightarrow z & \rightarrow \$
\end{align*}
\]

\[0 13 0 2 27 3 0 \implies annb$aa\]
Decompression

- Assume we are given the indices in the compressed file
- Follow the steps of MTF and write down the character of each index

\[ a \rightarrow b \rightarrow \ldots \rightarrow n \rightarrow \ldots \rightarrow z \rightarrow $ \]
\[ 0 \ 13 \ 0 \ 2 \ 27 \ 3 \ 0 \ \Rightarrow \ annb$aa \]

- Can we replace MTF by another algorithm?
Decompression

- Assume we are given the indices in the compressed file
- Follow the steps of MTF and write down the character of each index

\[ a \rightarrow b \rightarrow \ldots \rightarrow n \rightarrow \ldots \rightarrow z \rightarrow $ \]

\[ 0 \hspace{1em} 13 \hspace{1em} 0 \hspace{1em} 2 \hspace{1em} 27 \hspace{1em} 3 \hspace{1em} 0 \hspace{1em} \Rightarrow \hspace{1em} annb$aa \]

- Can we replace MTF by another algorithm?
  - Yes, any online list update algorithm can be used.
  - The quality of compression might change!
Decompression

- Assume we are given the indices in the compressed file
- Follow the steps of MTF and write down the character of each index

\[a \rightarrow b \rightarrow \ldots \rightarrow n \rightarrow \ldots \rightarrow z \rightarrow \$

\[0 \ 13 \ 0 \ 2 \ 27 \ 3 \ 0 \longrightarrow \text{annb}$aa\]

- Can we replace MTF by another algorithm?
  - Yes, any online list update algorithm can be used.
  - The quality of compression might change!

- What about an algorithm with advice?
Any list update algorithm with advice can be used to replace MTF in bzip2.

The advice bits should be included in the compressed file.

- E.g., we store two bits of advice in the compressed file for text $T$.
- Advice bits indicate whether Timestamp, MTFE, or MTFO is better and used for compressing $T$. 
Any list update algorithm with advice can be used to replace MTF in bzip2.

The advice bits should be included in the compressed file.
- E.g., we store two bits of advice in the compressed file for text $T$.
- Advice bits indicate whether Timestamp, MTFE, or MTFO is better and used for compressing $T$.

Project topic: implement and compare different list-update algorithms for compression purposes. Explore, how advice can be helpful in achieving more compressed results.
Self-Adjusting Trees
Self-Adjusting Trees
The input is a set of requests to items in a list of length L

- The goal is to update the list to adjust it into patterns in the input.
- There is a lot of locality in the input sequence: \langle 2 2 2 2 2 1 1 3 3 3 3 3 3 1 1 2 2 2 \rangle
- Move-To-Front and Timestamp have competitive ratio of 2, and they are the best deterministic list-update algorithm
The input is a set of requests to items in a BST of size N.

- The goal is to update the tree to adjust it into patterns in the input.
- There is a lot of locality in the input sequence.
- Can we apply Move-To-Front for trees?
Splay Trees Idea

- When there is a request to item \( a \), adjust the tree so that \( a \) becomes root in the new tree!
- Use tree rotations to ‘bubble up’ the accessed item.
- We say that we splay \( a \) to become root in the adjusted tree
  - It is a natural extension of Move-To-Front to the lists.
Self-Adjusting Trees

Splay Trees Idea

- When there is a request to item $a$, adjust the tree so that $a$ becomes root in the new tree!
- Use tree rotations to ‘bubble up’ the accessed item.
- We say that we splay $a$ to become root in the adjusted tree.
  - It is a natural extension of Move-To-Front to the lists.
Consider accessed item $a$, its parent $p$ and grand-parent $g$ (if they exist).

Reorder $a$, $p$, and $g$ so that $a$ appears ‘above’ the other two

- If $a$ is smallest/largest, $p$ and $g$ will be in one side of $a$.
- If $a$ is in between, $p$ and $g$ will be on its left and right.
Consider accessed item \( a \), its parent \( p \) and grand-parent \( g \) (if they exist).

Reorder \( a \), \( p \), and \( g \) so that \( a \) appears ‘above’ the other two

- If \( a \) is smallest/largest, \( p \) and \( g \) will be in one side of \( a \).
- If \( a \) is in between, \( p \) and \( g \) will be on its left and right.

After re-ordering \( a \), \( p \), and \( g \), ‘place’ the following four subtrees in their appropriate position to save BST property:

- the two subtrees of \( a \)
- the sibling of \( a \) in the subtree of \( p \)
- the sibling of \( p \) in the subtree of \( g \)
Splay Example

E.g., Access $a = 12$
Self-Adjusting Trees

Splay Example

E.g., Access $a = 12$
Self-Adjusting Trees

Splay Example

E.g., Access $a = 12$
Self-Adjusting Trees

Splay Example

E.g., Access \(a = 12\)
Self-Adjusting Trees

Splay Example

E.g., Access \( a = 12 \)
Self-Adjusting Trees

Splay Example

E.g., Access $a = 12$
Splay Example

E.g., Access \( a = 12 \)
Splay Example

E.g., Access $a = 12$
Splay Example

E.g., Access $a = 12$
E.g., Access $a = 12$
Splay Example

E.g., Access $a = 12$
The accessed node $a$ is either

- Root
- Child of the root
- Has both parent ($p$) and grandparent ($g$):
  - Zig-zig pattern: $g \rightarrow p \rightarrow a$ is left-left or right-right
  - Zig-zag pattern: $g \rightarrow p \rightarrow a$ is left-right or right-left
Self-Adjusting Trees

Access root

- if $x$ is root, do nothing!
When $x$ is child of the root, do a single rotation to move it above its parent.

- It is called a zig operation.
Self-Adjusting Trees

Access LR or RL grandchild

- When $x$ is left-child (resp. right-child) of $P$ and $p$ is right-child (resp. left-child) of $g$, do a double rotation.

- It is called a zig-zag operation.
Reverse the order of $a, p,$ and $g$.

It is called a **zig-zig** operation.
Splay Example

E.g., Access $a = 6$
Splay Example

E.g., Access \( a = 6 \)
Splay Example

E.g., Access $a = 6$
Splay Example

E.g., Access \( a = 6 \)
Self-Adjusting Trees

Splay Example

E.g., Access $a = 4$
Self-Adjusting Trees

Splay Example

E.g., Access $a = 4$

![Diagram of splay tree transformation](attachment:image.png)
**Self-Adjusting Trees**

**Splay Example**

- E.g., Access $a = 4$

![Diagram of splay tree example](image)
Splaying: Intuition

- The accessed node is moved to ‘front’ (i.e., is now root)
- Let $b$ be a node on the access path from root to the accessed node $a$. If $b$ is at depth $d$ before the splay, its at about depth $d/2$ after the splay.
  - ‘Deeper nodes’ on the access path tend to move closer to the root

![Diagram of tree transformation](image)
Splaying: Intuition

- The accessed node is moved to ‘front’ (i.e., is now root).
- Let $b$ be a node on the access path from root to the accessed node $a$. If $b$ is at depth $d$ before the splay, its at about depth $d/2$ after the splay.
  - ‘Deeper nodes’ on the access path tend to move closer to the root.
- Splaying gets amortized $O(\log N)$ amortized time.
  - $N$ is the number of nodes in the tree.
Self-Adjusting Trees

BST-Update problem

- So far, we learned how Splay trees work; they are equivalent of self-adjusting lists updated with MTF.
Self-Adjusting Trees

**BST-Update problem**

- So far, we learned how Splay trees work; they are equivalent of self-adjusting lists updated with MTF.

- BST-Update problem:
  - The input is an online sequence of requests to items in a BST.
  - Each probe for finding an item $x$ has cost 1.
  - On the path traversed from the root to $x$, the algorithm can make any number of rotations at a cost of 1 per rotation.
So far, we learned how Splay trees work; they are equivalent of self-adjusting lists updated with MTF.

**BST-Update problem:**

- The input is an online sequence of requests to items in a BST.
- Each probe for finding an item $x$ has cost 1.
- On the path traversed from the root to $x$, the algorithm can make any number of rotations at a cost of 1 per rotation.

**Dynamic Optimality Conjecture:** Splay tree is a competitive solution, i.e., it has a competitive ratio independent of the size $N$ of tree and length $n$ of sequence.

- We know the competitive ratio of splay trees is $O(\log N)$
BST-Update problem

- So far, we learned how Splay trees work; they are equivalent of self-adjusting lists updated with MTF.

- BST-Update problem:
  - The input is an online sequence of requests to items in a BST.
  - Each probe for finding an item $x$ has cost 1.
  - On the path traversed from the root to $x$, the algorithm can make any number of rotations at a cost of 1 per rotation.

- **Dynamic Optimality Conjecture**: Splay tree is a competitive solution, i.e., it has a competitive ratio independent of the size $N$ of tree and length $n$ of sequence.
  - We know the competitive ratio of splay trees is $O(\log N)$
  - The best existing algorithm is provided by self-adjusting Tango Trees, and has a competitive ratio of $O(\log \log N)$
Potential Project Topics

- Write a survey of the self-adjusting data structures (other than linked lists).
  - In particular, think of BSTs and other structures.
  - For example, is there any self-adjusting hash table? what about self-adjusting skip lists?
- Think about advice BST-Update algorithms with advice?
  - How many bits are sufficient to achieve an optimal algorithms?