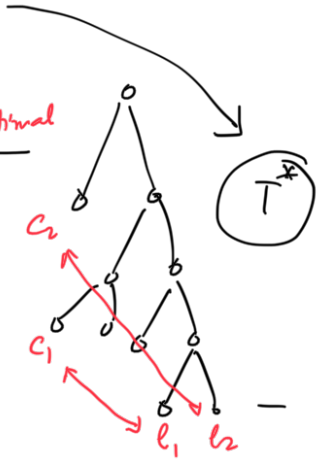
16/9/21

(i) We showed that there is optimal solution T^* where c_1, c_2 have a common parent.



(ii) Induction proof: Assume that the greedy alg. is optimal when there $\leq k-1$ letters.

k letters: $\{a_1, \dots, a_k\} = \Sigma$

At the first step: the greedy alg. combines letters $c_1, c_2 \to c$.

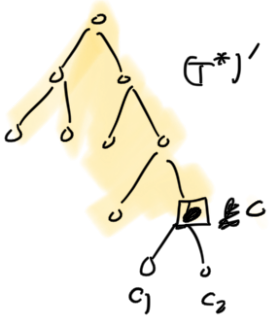
By (i), there is an opt. solution T^* where c_1, c_2 have a common parent

$\Sigma' = (\Sigma - \{c_1, c_2\}) \cup \{c\}$: $k-1$ letters.

By induction hypothesis, the greedy alg is opt. for Σ'.

$(T^*)'$: T^* with c_1, c_2 removed and the common parent labelled c.

Then, $(T^*)'$ is also a solution for Σ'!



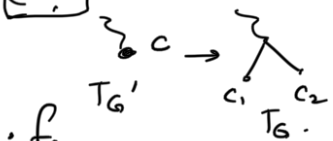
By IH: the encoding length of greedy on Σ' $\leq$ encoding length of $(T^*)'$

$+ f_{c_1} + f_{c_2}$ $+ f_{c_1} + f_{c_2}$

encoding length of Σ in T_G $\leq$ encoding length of T^*

T_G: greedy solution tree

$(T_G)'$: greedy tree for Σ'.



$T_G' \to c$

$\sum_{\ell \in \Sigma'} \text{depth}(\ell) \cdot f_\ell$ (ℓ in T_G')

$\text{depth}(c) \cdot f_c$

$= \text{depth}(c) \cdot f_{c_1} + \text{depth}(c) f_{c_2}$

$\sum_{\ell \in \Sigma} \text{depth}(\ell) \cdot f_\ell$ (ℓ in T_G)

$\text{depth}(c_1) \cdot f_{c_1} + \text{depth}(c_2) \cdot f_{c_2}$

diff $= f_{c_1} + f_{c_2}$ ∵

$\text{depth}(c_1) = \text{depth}(c_2) = \text{depth}(c) + 1$

Next topic: AMORTIZED COMPLEXITY.

$T(n)$: worst case running time

@ $\underbrace{i_1, i_2, \ldots, i_k}_{n \quad n \quad n}$ → A

$O(kT(n))$.

$\ll \Rightarrow kT(n)$.

Amortized Compl. = $\frac{\text{Total time}}{k}$

Example: (i) Bit increment operation:

$\leq m$ bits

worst case comp. $= m$

$011\ldots1 + 1 = 10\ldots00$

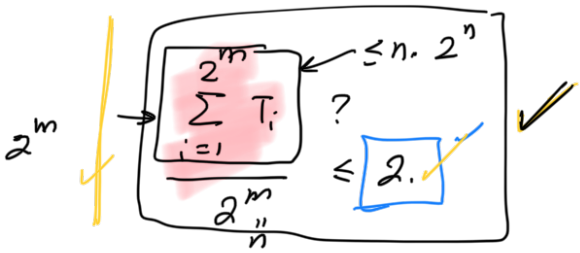
$0, +1, +1, \ldots, \underbrace{1\ldots1}_{m}$

T_i

x

$101\ 11\ 01 + 1 = 101\ 11\ 100$

#bit ops = 1 + # consec. 1's at the end.



Potential Function: Φ_i : non-negative number at the beginning of the i^{th} input.

[Idea: show that when the alg. takes lot of time, potential drops].

T_i : time taken by the alg. at the i^{th} step.

Goal: $T_1 + T_2 + \ldots + T_n \leq C \cdot n$: one way of showing it that $T_i \leq C(n)/n$. $= C$.

Modified time at step i (Amortized) $T_i' := T_i + \Delta\Phi_i = T_i + (\Phi_{i+1} - \Phi_i)$ (*)

(i) $\Phi_i \geq 0$ ✓

(ii) $\Phi_0 = 0$ ✓

(iii) $T_i' \leq C$ for each i

$\Rightarrow \sum_{i=1}^{n} T_i' \leq C \cdot n$

$\sum T_i \leq \sum T_i'$

$T_1' = T_1 + \Phi_2 - \Phi_1$

$T_2' = T_2 + \Phi_3 - \Phi_2$

$T_3' = T_3 + \Phi_4 - \Phi_3$

$\vdots$

$T_n' = T_n + \Phi_{n+1} - \Phi_n$

$C_n \geq T_1' + \ldots + T_n' = T_1 + \ldots + T_n + \Phi_{n+1} - \Phi_1$ (6)

$\Rightarrow$ $T_1 + \dots + T_n \le Cn$ ≥ 0

$T_i \le \lg n$

$0, +1, +1, +1, \dots, +1$ ($\lg n$) n times

$T_1 + \dots + T_n \le c \cdot n$ ←

$T_i \le c$ ✗

x_i : number at the beginning of ith step ($\equiv i$ in binary)

Φ_i = # 1's in the binary expansion of i ($\equiv$ # 1's in x_i)

$$T_i' = T_i + (\Phi_{i+1} - \Phi_i)$$

$\Phi_1 = 0$, $\Phi_i \ge 0$.

Claim: $T_i' \le 2$ $\Rightarrow$ $T_1 + \dots + T_n \le 2n$ as well!

$\Rightarrow \sum_i T_i \le 2n$

Pf: x_i

(i)
$$\begin{array}{r} 110110\dots10 \\ + \quad\quad\quad 1 \\ \hline 110110\dots11 \end{array}$$

$\Phi_{i+1} - \Phi_i = 1$

$T_i = 1$

$T_i' = 2$ ✓

(ii)
$$\begin{array}{rl} 1101110\dots0\overbrace{11111}^{\ell} & (i) \\ + \quad\quad 1 & \\ \hline 1101110\dots1\,0\dots0 & (i+1) \end{array}$$

$\Phi_{i+1} - \Phi_i = 1 - \ell$

$T_i = \ell + 1$.

$$\begin{array}{r} 10111 \\ + \quad 1 \\ \hline 11000 \end{array}$$

$\Phi_i = 4$

$\Phi_{i+1} = 2$.

$2 - 4$

$T_i' = 2$

$T_1 + \dots + T_n \le 2n$

(ii) Extendible Array:

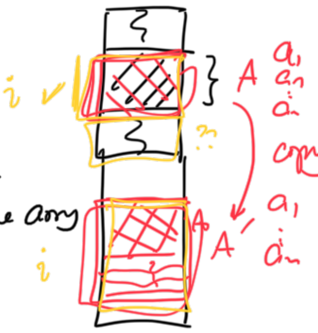
Array 1

$A[2]$

Adding x at the end of the array

$A[++i] = x$

n



$1, 2, 3, 4, \dots, i, \; i+1, \dots, n$

$\sum_{i=1}^{n} i \approx n^2$

Amortized Time: n.

$A[i]$ $O(1)$

n operatn

x_1, x_2, x_i, x_n

$\approx n$

Inc(x) {

if A is full

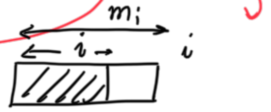
Copy A into an array of twice the size

$\approx n^2$.

→ then add x at the end
else add x at the end.

$1 + (2) + (3) + 1 + (5) + 1+1+1 + (9) + 1+1+1+\dots+1 + \dots \leq 2n$

want to show: total time taken after n inserts $\leq 2n$



m_i = size of the array

Φ :

— x —

$T_i' = T_i + \Phi_i - \Phi_{i-1}$ ✓

when T_i is large, $\Phi_i \ll \Phi_{i-1}$

$T_i' \leq 2$. always.

T_i is large when array is full.

$\Phi_i := -m_i$

$\Phi_i \geq 0$

(1) the array size does not change

$T_i = 1$

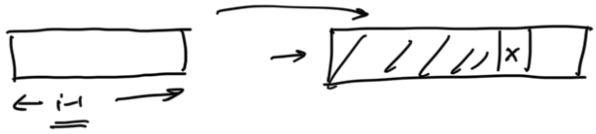
$\Phi_i - \Phi_{i-1} = 0$

$T_i' = 1. \longrightarrow T_i' = 1 + [\ 2\ ] = 3.$

$\Phi_i = 2i - m_i \geq 0$

2 (# elements) − size of the array. ≥ 0

(2) the array is full at the beginning of this step:



size of the array $\leq$ 2 (# elements).

$T_i = (i-1) + 1 = i$

$T_i' = i + (-(i-1))$ ✓

$T_i \quad \Delta\Phi$

$= 1$

$-(i-1) \to -(2i-2)$

$T_i' = i + [2 - (i-1)]$

$= 3.$

$T_i' = T_i + \Phi_i - \Phi_{i-1}$

$T_1' + \dots + T_n' = T_1 + \dots + T_n + \Phi_n - \Phi_1$

$\leq kn$

$\Rightarrow T_1 + \dots + T_n \leq kn + \Phi_1 - \Phi_n$

$\leq kn$

This shows that amortized time complexity is ≤ 3.

— x —

$T_1 + \dots + T_n \leq kn - \Phi_n$, $\Phi_n \geq 0$

$\leq kn$.

$T_1 + \dots + T_n \leq 3n$ ✓

$T_i \approx i$

$T_1' + \dots + T_n' \leq 3n - \Phi_n \Rightarrow T_1 + \dots + T_n \leq 3n.$

$\Phi_n \in [0, n]$.

Resizable Array: At each step, (i) Insert a new element at the end of the array

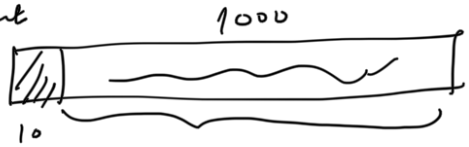
$n \leftarrow$ # elements (ii) remove the last element.

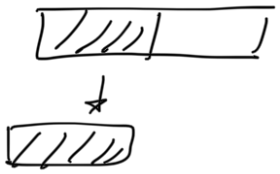
push/pop.

(i) n++, A[n] = x
(ii) n--

→ if the array is full,
double the size of the array & copy elements

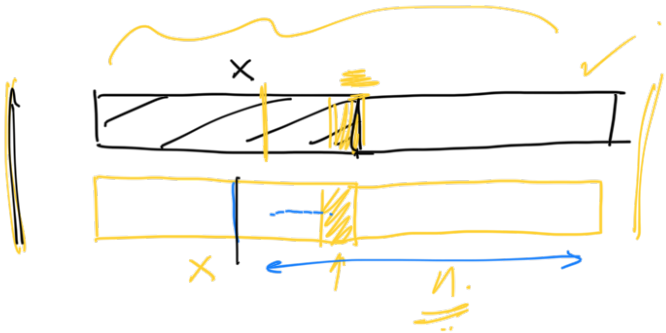
want that: not too space is wasted.

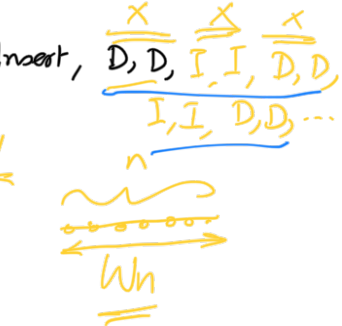




Insert(x) {
  if array is not full
    n++, A[n] = x
  else
    copy the array into an array of twice the size, A (new array)
    n++, A[n] = x
    n = 2n. ✓
}.

Remove Last {
  if (# element ≤ n/2) n/4.
    copy the elements into an array of half the size, and update n ← n/2
  n-- ✓
}.





What is the potential function??

m_i : size of the array at step i

n_i : # elements in the array at step i. (easier $n_i = i$)

$\Phi_i = 2n_i - m_i$, $+ m_i$

$T_i' = T_i^* + \Phi_i - \Phi_{i-1}$

When is T_i^* is a lot??

$2n_i - m_i$ if $n_i = m_i$ and we are inserting

when we are deleting and $n_i = m_i/4$

$m_i/4$

$4n_i - m_i$

$$\Phi_i = \begin{cases} = 2n_i - m_i \geq 0 & \text{(i) if the array is more than half full} \\ = m_i - 2n_i \geq 0 & \text{(ii) if the array is less than half full} \end{cases}$$

Is $\Phi_i \geq 0$??

(i) : $2n_i \geq m_i$

(ii) : $m_i \geq 2n_i$

Insert Operation {

(a) If the array is not full

$T_i = 1.$

$T_i' = 1 + [\ 2 \ ] - 2$

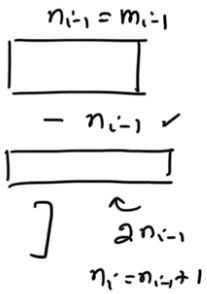
$\leq 3.$

(b) If the array if full

$T_i = n_i + 1$

$T_i' = n_{i-1} + 1 + [\ 2 - n_{i-1}\ ]$

$= \leq 3.$



change in potential

$= [\ 2(n_{i-1}+1) - 2n_{i-1}]$

$= 2.$

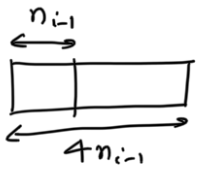
Delete Op :

(a) If the array is more the 1/4 -full :

$T_i = 1$

$T_i' = T_i + \Delta\Phi \leq 3.$

≤ 2



(b) If the array is 1/4- full :

$T_i = n_{i-1} + 1$

$T_i' = n_{i-1} + 1 + \Delta\Phi$

$- 2n_{i-1}$

$\leq 1 - n_{i-1} \leq 3$

Initial pot := $3n_{i-1}$

Final pot := n_{i-1}

$= 2n_{i-1} - n_{i-1}$

$= n_{i-1}$

We have shown that $T_i' \leq 3$ always. ∴, amortized to ≤ 3.

CLRS :

30/9/21.

Queue from two stacks.

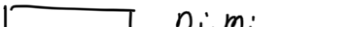
S_1, S_2

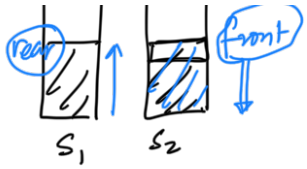
a b c d

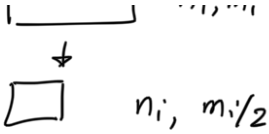
what if the potential is

$4n_i - m_i$ ↑

↑ tbl. ↑ size of the arr.







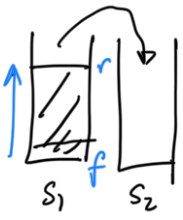
$T_i = \frac{m_i}{2}$

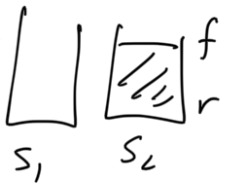
T_i' = T_i + change in Φ

$m_i/2$ $\quad$ $m_i/2$

```
Enqueue(x) {
    S1.push(x)   }.        (1)

Dequeue(·) {
    if (S2 is not empty)
        S2.pop()           (1)
    else
        while S1 is not empty
            S2.push(S1.pop()).      ← list of operations
                                      n
    S2.pop()
}
```





N

Want to show that amortize complexity is $O(1)$.

$T_i' = T_i + \Phi_i - \Phi_{i-1}$

$\Phi_i = 2$(#elements in stack S_1) $- 2$(# ... S_2).

Enq: $1 + 2 \cdot 1 = 3$.

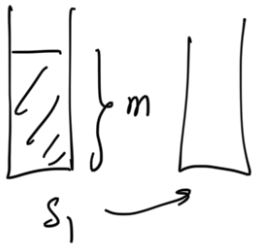
Deq: (i) $T_i' = 1$.

(ii) $T_i = 2m + 1$ ✓

$\Phi_{i-1} = 2m$

$\Phi_i = 0$

$T_i' = 1$



$T_i' \le c$.

$T_1 + \dots + T_N \le T_1' + \dots + T_N' \le cN$

$- \Phi_N$

Amortized Cost ≤ Average Cost ≤ Worst Cost

$O_1, O_2, \dots, O_N$.

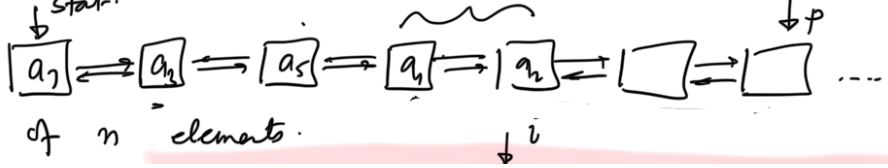
we say that amortized compl is c if the total running time $\le c \cdot N$.

$i_1, \dots, i_L$

running time(i_1) + $\dots$ + running time(i_L) / L

max running time of i on input n

Move-to-Front Heuristic (Sleator, Tarjan '82).



Linked List of n elements.

Input : $a_5, a_7, a_3, a_5, a_3, a_5, a_1, \ldots \ldots, a_{t_N}$

$3 + 1 + 2 + 3 + \ldots.$

"Cost" := total time to access the elements.

At time i, the alg. sees the next request a_{t_i}

A **Online alg** :- Move-to-front heuristic. : At time i, when a_{t_i} is accessed, move it to the front.

Offline Alg :- knows the entire seq. beforehand.

$1, 1, 1, 1, 7, 7, 7, 1, 1, 7, 7, 7, 5, 5, 5, 5, \ldots, 5$

↓

OPT : the optimal cost of an off-line alg.

Thm : For any sequence $\sigma = a_{t_1}, \ldots, a_{t_N}$

total cost (A) $\leq 4 \cdot$ total cost (OPT for σ).

A_i : cost incurred by the alg. at time i

O_i : " opt. time i

Φ_i : potential at time i

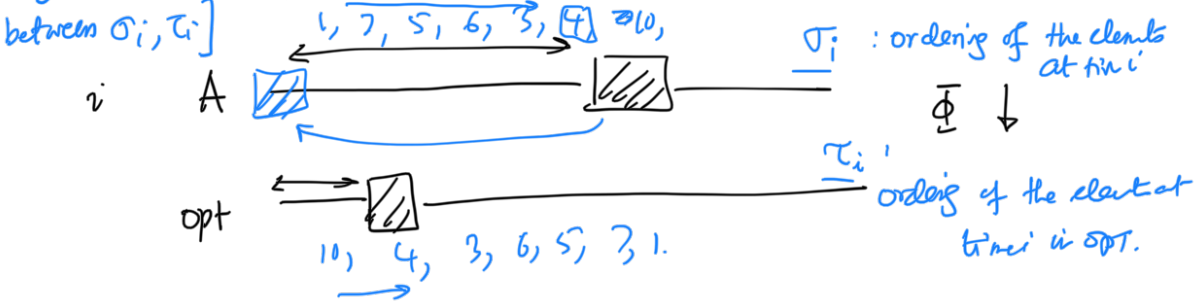
Amortize cost

$$A_i' = A_i + \Phi_i - \Phi_{i-1}$$

we will show that $A_i' \leq 4 O_i$ for every step i ← $\not\Rightarrow A_i \leq 4 O_i$?!!

$$A_1 + \ldots + A_N \leq A_1' + \ldots + A_N' \leq 4(O_1 + \ldots + O_N).$$

Φ_i := [# inversions between σ_i, τ_i]



Def : Given two permutations σ, τ of some elements $a_1, \ldots, a_n$ the # inversions between σ and τ is defined as

the number of pairs (a_i, a_j) which appear in opposite order in the two sequences.

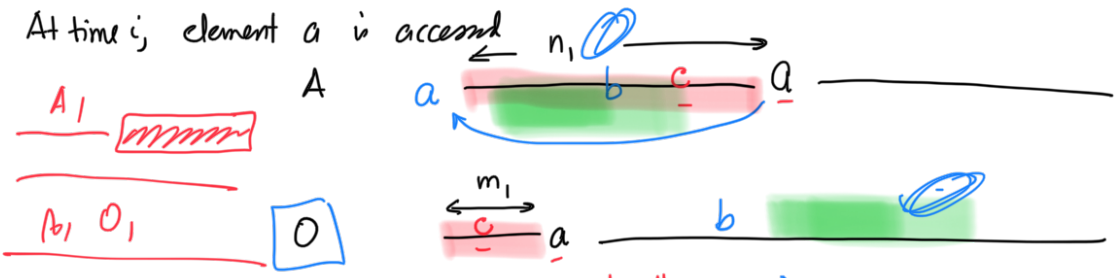
σ : —— a_i —— a_j —— (a_i, a_j)

τ : ——— a_j — a_i ———

1, 2, 3, 4 n^2 (1,4), (2,3) ← { 1, 2, 3, 4 / 4, 1, 3, 2

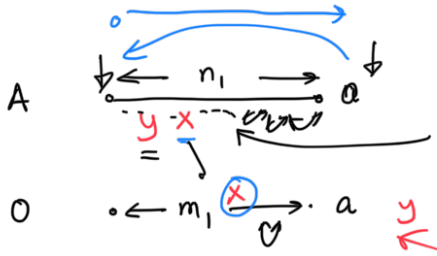
4, 3, 2, 1 (3,4).

At time i, element a is accessed



(i) ↓ ‖ (b, a) was inversion earlier, but now it is not an inv.

(ii) ↑ (c, a) was not an inversion, but now it ✓ becomes an inversion



Let I_1 / x be the elements here which appear before a in the opt. sequence as well.

I_2 element — a which appear after a in the opt. list.

(i) $|I_1| + |I_2| = n_1 - 1$

(ii) $A_i = \frac{n_1 + 1}{m_1}$ $2n_1$

(iii) $O_i = m_1 \geq |I_1|$

(iv) change in potential? $\Phi_i - \Phi_{i-1} = |I_1| - |I_2|$

$A_i = 0$.

$A_i \leq 2O_i$

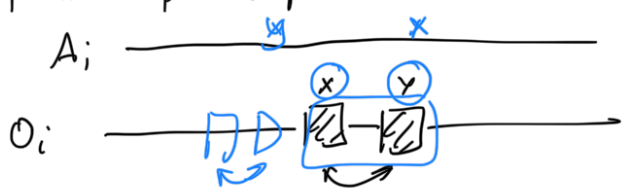
- $A_i' = A_i + \Phi_i - \Phi_{i-1} \overset{?}{\leq} 2O_i$

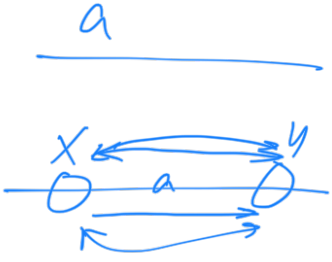
$= (n_1 + 1) + |I_1| - |I_2|$

$= |I_1| + |I_2| + |I_1| - |I_2|$

$= 2|I_1| \leq 2m_1 = 2O_i$

- what if the opt. swaps two elements?





$0 + \Delta\Phi \leq 4 * 1$

≤ 1

$0 + \Delta\Phi \leq 4 * 1$