EECS 4101/5101

Prof. Andy Mirzaian

Advanced Data Structures

COURSE THEMES

- Amortized Analysis
- Self Adjusting Data Structures
- Competitive On-Line Algorithms
- Algorithmic Applications

COURSE TOPICS

Phase I: Data Structures

- Dictionaries
- Priority Queues
- Disjoint Set Union

Phase II: Algorithmics

- Computational Geometry
- Approximation Algorithms

INTRODUCTION

- Amortization
- Self Adjustment
- Competitiveness

References:

- [CLRS] chapter 17
- Lecture Note 1 (and parts of LN11)

Amortization

Data Structure Analysis Methods

- What is the total computational time to execute a sequence of operations on a data structure?
  - These operations have correlated effect on the DS

- Worst-Case Analysis:
  - worst-case time per operation × number of operations
  - ignores the correlated effects on the DS
  - this upper-bound is most often too pessimistic

- Expected-Case Analysis:
  - needs probability distribution
  - probabilistic assumptions may not correspond to reality
  - does not give upper-bound

- Amortized Analysis:
  - simple and robust
  - correlated effects are taken into account
  - gives tighter upper-bound
  - is used to design self adjusting data structures

Amortized Analysis

- amortized time = average time of an operation over a worst-case sequence of operations.

- a sequence of n operations $s = (op_1 , op_2 , \ldots , op_n)$ on the data structure (starting from some given initial state)

- actual time costs per op in the sequence: $c_1 , c_2 , \ldots , c_n$
- total time over the sequence: $c_1 + c_2 + \ldots + c_n$
- worst-case total: $T(n) = \max_s (c_1 + c_2 + \ldots + c_n)$
- amortized time is (a good upper bound on) $T(n)/n$

- Three Methods of Amortized Analysis:
  - Aggregate Method: $T(n)/n$
  - Accounting Method: a banker's point of view
  - Potential Function Method: a physicist's point of view

Two running D.S. examples

1. Stack with Multipop:
   - A sequence of n intermixed Push(x,S), Pop(S), Multipop(S) operations on an initially empty stack S
   - Multipop example: **while** $S \neq \emptyset$ and $Top(S) < 0$ **do** Pop(S)
   - time cost unit = O(1) per Push and Pop
   - worst-case actual cost of a single Multipop = O(n)

2. Binary Counter with Increment operation:

- B-bit Binary Counter $A = A_{B-1} \ldots A_2 A_1 A_0$
- initially all 0 bits
- **procedure** Increment(A)

  $i \leftarrow 0$

  while $i < B$ and $A_i = 1$ **do** $A_i \leftarrow 0$; i++ **end-while**

  if $i < B$ **then** $A_i \leftarrow 1$

  end

- time cost unit = O(1) per bit-flip
- worst-case actual cost of a single Increment = O(B)

0000000
0000001
0000010
0000011
0000100
0000101
0000110
0000111
0001000
…

Aggregate Method: Stack with Multipop

Worst-case Analysis:

n operations.

worst-case per operation = $O(n)$. (This happens for a Multipop.)

total worst-case $T(n) = n * O(n) = O(n^2)$.

amortized cost per operation $\leq T(n)/n = O(n^2)/n = O(n)$.

this is too pessimistic and not tight!

Amortized Analysis:

Correlation: Each Pop (including those in Multipops) corresponds to a previously pushed item.

Pops $\leq$ # Pushes $\leq n$.

$T(n) \leq 2n$.

amortized cost per operation = $T(n)/n \leq 2 = O(1)$.

Aggregate Method: Binary Counter + Increment

Worst-case Analysis:

n operations.
worst-case per operation = O(B). (All bits could flip.)
total worst-case T(n) = n * O(B) = O(nB).
amortized cost per operation ≤ T(n)/n = O(nB)/n = **O(B).**
this is too pessimistic and not tight!

B=10	n	actual cost per op	aggregate cost	Bn	2n
0000000000	0	0	0	0	0
0000000001	1	1	1	10	2
0000000010	2	2	3	20	4
0000000011	3	1	4	30	6
0000000100	4	3	7	40	8
0000000101	5	1	8	50	10
0000000110	6	2	10	60	12
0000000111	7	1	11	70	14
0000001000	8	4	15	80	16
…	…	…	…	…	…

Aggregate Method: Binary Counter + Increment

Amortized Analysis:

Correlation: Count bit-flips column by column, instead of row by row.

0	0	0	0	0	0	0
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	0	1	1
0	0	0	0	1	0	0
0	0	0	0	1	0	1
0	0	0	0	1	1	0
0	0	0	0	1	1	1
0	0	0	1	0	0	0
.	.	.	.	.	.	.

$$T(n) \le \frac{n}{2^{B-1}} + \cdots + \frac{n}{4} + \frac{n}{2} + n < n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \le 2n$$

Amortized cost per Increment = T(n)/n < 2n/n = 2 = **O(1).**

Accounting Method

- Banker's point of view.
- Allows different amortized costs per operation

(amortized cost − actual cost)
- >0 credit "stored" in specific parts of the DS
- <0 debit used up from stored credits

- Must have [Total stored Credit] = Aggregate Amortized Cost − Aggregate Actual Cost ≥ 0

- This implies that the aggregate amortized time is an upper bound on aggregate actual time even for the worst-case sequence of operations.

- **CREDIT INVARIANT** states the credit/debit status of the DS at any state.

Accounting Method: Stack with Multipop

J	$1
I	$1
H	$1
G	$1
F	$1
E	$1
D	$1
C	$1
B	$1
A	$1

Credit Invariant: $1 stored with each stack item.

Amortized costs:

Push	$2	Pay $1 for push, store $1 with new item
Pop	$0	Use stored $1 to pop the item
Multipop	$0	Use the stored $1's for each item popped

Accounting Method: B. C. + Increment

- C := actual cost (could be as high as B)
- C-1 1-to-0 bit flips, but only one 0-to-1 bit flip.

← C →

X X . . . X 0 1 1 . . . 1 1

↓ Increment

X X . . . X 1 0 0 . . . 0 0

- Many 1-to-0 bits to flip. Expensive. They should have stored credits.

- Credit Invariant: \$1 stored with each 1-bit, \$0 with each 0-bit.
- For the C-1 1-to-0 bit flips, their stored \$1's will pay for their flips.
- For the one 0-to-1 bit flip: pay \$1 for the flip and store another \$1 at the new 1-bit to maintain the Credit Invariant.
- Amortized cost of an Increment = \$2

Potential Function Method

- Physicist's point of view.
- Potential energy = aggregate of prepaid credits "stored/assigned " to specific parts of the data structure.
- Potential energy can be released to pay for future operations.
- D_0 = initial data structure

  D_i = D.S. after the i^{th} operation op_i, i=1..n $\qquad D_{i-1} \xrightarrow[c_i]{op_i} D_i$

  c_i = actual cost of op_i, i=1..n

- Potential Function $\Phi : \mathcal{D} \rightarrow \Re$

  reals

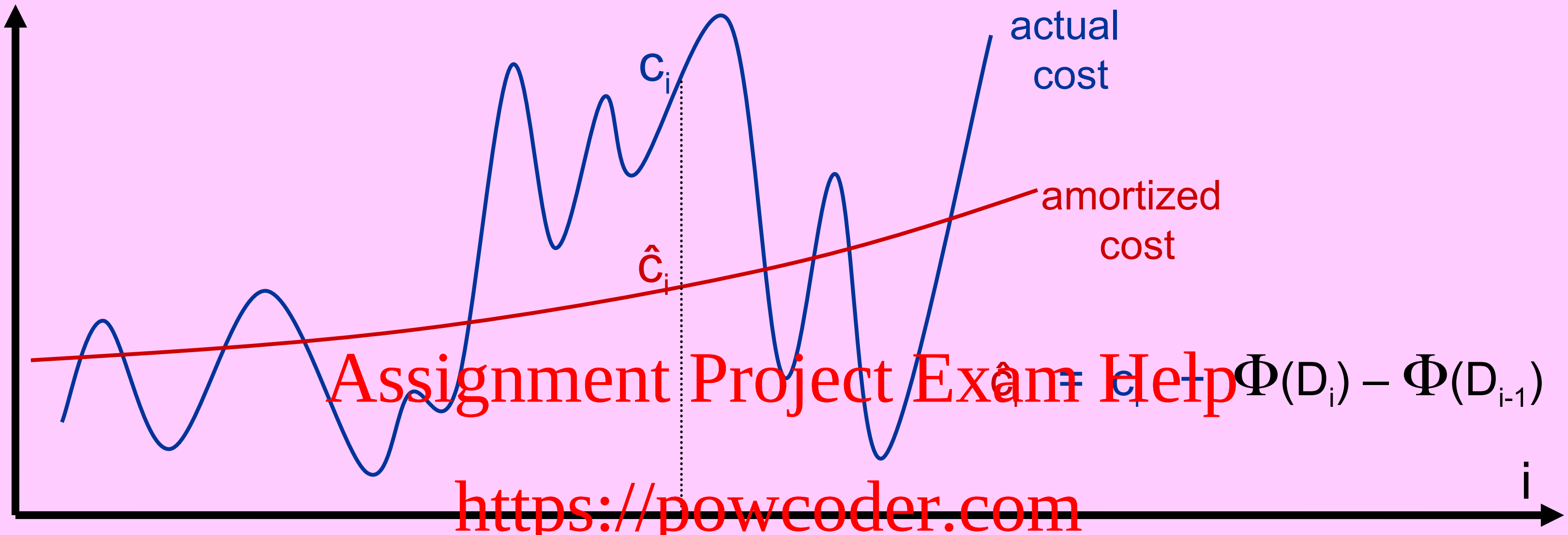
  set of all possible states of the DS

- Φ(D) = potential of the data structure at state D. Memoryless!
- Defining an effective potential function is the key to the analysis.

- $\hat{c}_i$ = amortized cost of op_i, i=1..n
- $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{change in potential (positive or negative) caused by } op_i}$

- $\hat{c} = c + \Delta\Phi$ (for short)

Potential Function Method



$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})] = \sum_{i=1}^{n} c_i + [\Phi(D_n) - \Phi(D_0)]$$

Regular Potential: $\Phi(D_0) = 0$, and $\Phi(D) \geq 0$ for all $D \in \mathcal{D}$

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

Potential Method: Stack with Multipop

$\Phi(S) = |S|$ (regular potential)

Amortized cost per operation:

Push: $\hat{c} = c + \Delta\Phi = 1 + 1 = 2$

Pop: $\hat{c} = c + \Delta\Phi = 1 - 1 = 0$

Multipop: $\hat{c} = c + \Delta\Phi = c - c = 0$

So, in all cases $\hat{c} = O(1)$.

Therefore, a sequence of n operations on an initially empty stack will cost at most O(n) time.

Potential Method: B. C. + Increment

$\Phi(A) = \sum_i A_i$ = # of 1-bits in A (regular potential)

Amortized cost per Increment:

$$\hat{c} = c + \Delta\Phi = c + [-(c-1) + 1] = 2.$$

0-to-1 bit flip

1-to-0 bit flips

Therefore, a sequence of n Increments on an initially 0 counter will cost at most O(n) time.

Self Adjustment

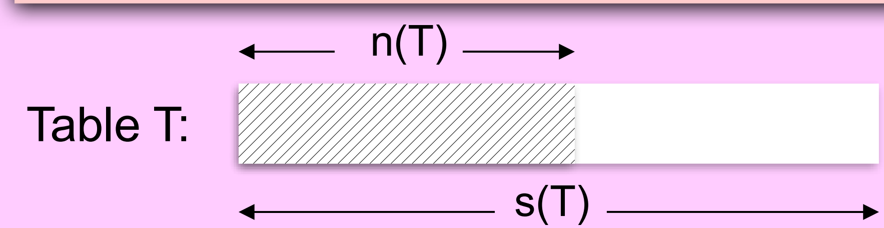
Self Adjusting Data Structures

- **Worst-case efficient data structures:**
  - Example: many variants of balanced search trees such as AVL trees.
  - Aim to keep the worst-case time per operation low.
  - Enforce explicit structural balance, and maintain extra balance information.
  - − Comparatively need more memory and computational overhead.
  - \+ Good for real-time applications.

- **Self-adjusting data structures:**
  - Efficient in amortized sense only.
  - Leave D.S. in arbitrary state, but update it in a simple uniform way, aiming to keep cost of future operations low.
  - \+ Much simpler than their worst-case efficient cousins. No explicit balance info.
  - \+ Adjust to pattern of usage and on some sequences of operations comparatively more efficient in the aggregate.
  - ± Tend to make more local changes, even during accesses, as well as updates. This could be bad for persistent data structures.
  - − Bad for real-time applications, since worst-case cost of an op could be high.
  - \+ Good for batch-mode applications.
  - We will study a number of important self-adjusting D.S. such as dynamic tables, lists, dictionaries, priority queues, disjoint set union.

discussed next

Dynamic Tables



[e.g., hash table, stack, heap, …]

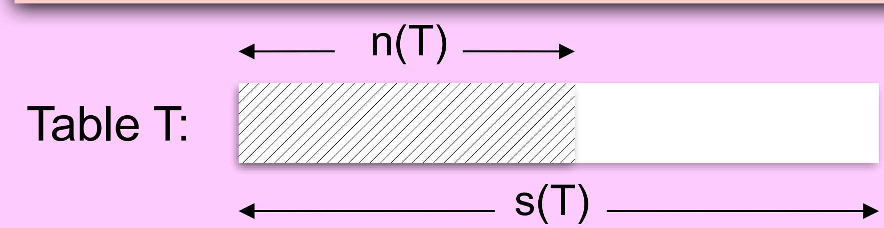
- s(T) = size of table T

- n(T) = # of items currently in T   [$0 \le n(T) \le s(T)$]

- $\alpha(T) = n(T)/s(T)$ load factor   [$0 \le \alpha(T) \le 1$]

- For good memory utilization we typically want to keep at least a certain fraction of the table full, e.g., $0.5 \le \alpha(T) \le 1$.

Dynamic Tables



[e.g., hash table, stack, heap, …]

- **Insert/Delete:** add to or remove a specified item from the table.

Case 1) Load factor remains within the allowable range.
Actual Cost O(1) time.

Case 2) Otherwise, need table Expansion / Contraction:

dynamically allocate a new table of larger/smaller size, and move all existing items from the old table to the new one, and insert/delete the specified item.
Load factor in new table must remain in the allowable range.
Actual cost O(n(T)).

Self-Adjustment: Is there a table expansion/contraction policy with amortized O(1) cost per insert/delete?

Semi-Dynamic Table: Insert only

- Need table expansion only.
- Before expansion: T_{old}, $s = s(T_{old})$, $n = n(T_{old})$, $\alpha = \alpha(T_{old})$
- After expansion: T_{new}, $s' = s(T_{new})$, $n' = n(T_{new})$, $\alpha' = \alpha(T_{new})$

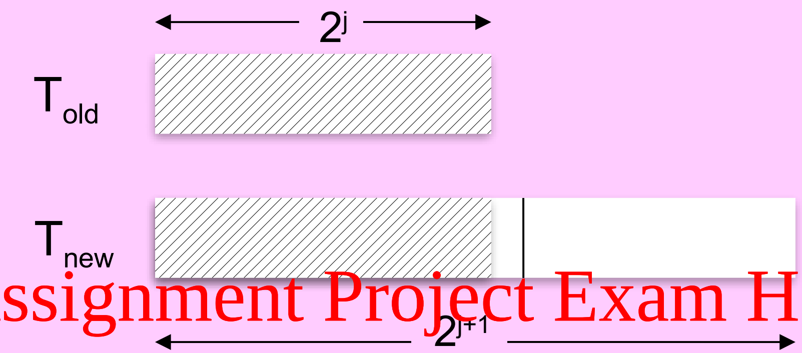
- **Expansion Policy:** when $\alpha = 1$ before an insertion,
  double table size: $s(T_{new}) = 2\, s(T_{old})$

- Right after expansion but before new item is inserted:
  $s' = 2s$, $\alpha' = n/s' = n/2s = \alpha/2 = ½$.
- Load factor range maintained: $½ \le \alpha \le 1$
- Insertion cost: 1 without expansion
  n+1 with expansion
- Worst-case cost of an insertion = O(n)
- What is the amortized cost of an insertion? We will derive it by
  - Aggregate Method
  - Accounting Method
  - Potential Function Method

S.-D. Table: Aggregate Method

With first insertion s(T) = 1 = 2^0. Afterwards it's doubled with each expansion.

Expansion:



Cost of i^{th} insertion: $c_i = \begin{cases} i & \text{if } i-1 = 2^j \text{ for some integer } j \geq 0 \\ 1 & \text{otherwise} \end{cases}$

$$T(n) = \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + \left(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots\right) \leq n + 2n = 3n$$

Amortized insertion cost = T(n)/n < 3n/n = 3 = O(1).

S.-D. Table: Accounting Method

Amortized cost: charge $3 for each insertion:

$1 to insert new item

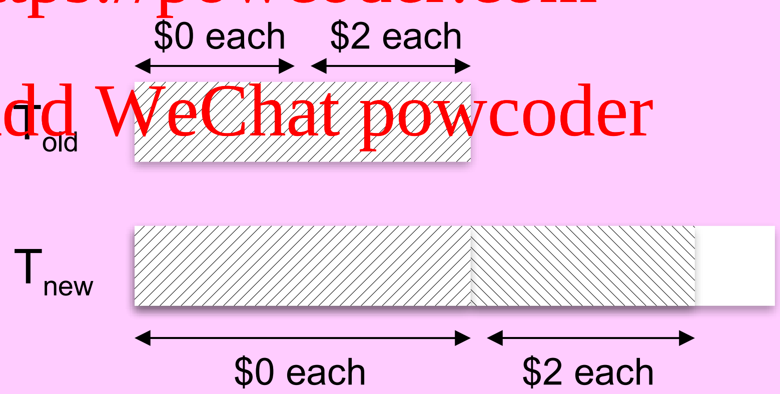
$2 to store with new item

Credit Invariant:

$2 stored with each item not previously moved (from an older table)

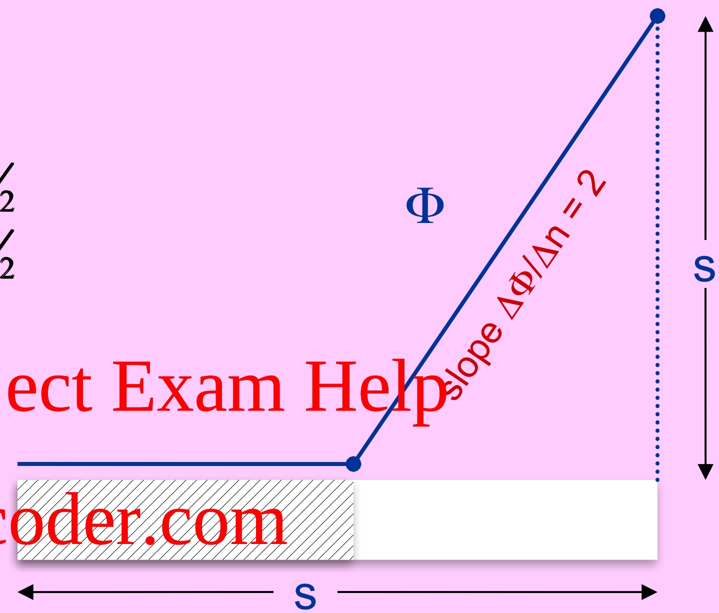
$0 stored with each item previously moved

Expansion:



S.-D. Table: Potential Fn Method

$$\Phi(T) = \begin{cases} 2n(T) - s(T) & \text{if } \alpha(T) \quad \tfrac{1}{2} \\ 0 & \text{if } \alpha(T) < \tfrac{1}{2} \end{cases}$$



Amortized cost of Insert:

Case 1) $\alpha<1$ (no expansion):

$$\hat{c} = c + \Delta\Phi = 1 + 2 = 3.$$

Case 2) $\alpha=1$ (w expansion): $n' = n+1 = s+1$

$$\hat{c} = c + \Delta\Phi = c + [\Phi_{after} - \Phi_{before}] = [s+1] + [2 - s] = 3.$$

In all cases $\hat{c} = O(1)$.

Dynamic Table: Insert & Delete

- First Try: $\frac{1}{2} \le \alpha \le 1$

when Delete
Contract: $s' = s/2$

when Insert
Expand: $s' = 2s$

- Does this strategy yield O(1) amortized cost per insert/delete?
- No. α is nearly at the other critical extreme after an expansion/contraction.
- Expensive sequence (assume n is power of 2):

$$\underbrace{\overbrace{\underbrace{I, I, I, \ldots\ldots, I, I,}_{\Theta(n)}}^{n/2-1} \overbrace{\underbrace{I, I, D, D,}_{\Theta(n)} \underbrace{I, I, D, D,}_{\Theta(n)} \ldots, \underbrace{I, I, D, D}_{\Theta(n)}}^{n/2+1}}_{T(n) = \Theta(n^2)}$$

Amortized cost per Insert/Delete = $T(n)/n = \Theta(n)$.

Dynamic Table: Insert & Delete

- Next Try: $\frac{1}{4} \le \alpha \le 1$

when Delete
Contract: s'=s/2

when Insert
Expand: s'=2s

- Does this strategy yield O(1) amortized cost per insert/delete?
- Yes. $\alpha = \frac{1}{2}$ just after an expansion/contraction (before the insert/delete).
  This is a fraction away from both extreme boundaries.

- See exercise on how to figure out good expansion/contraction policy to handle other pre-specified ranges for α.

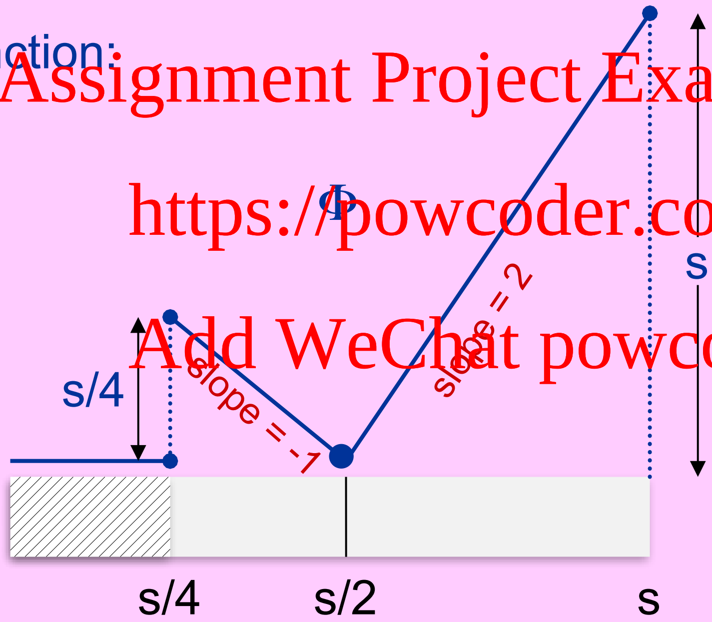
Dynamic Table: Insert & Delete

- Next Try: $\frac{1}{4} \leq \alpha \leq 1$

when Delete
Contract: s'=s/2

when Insert
Expand: s'=2s

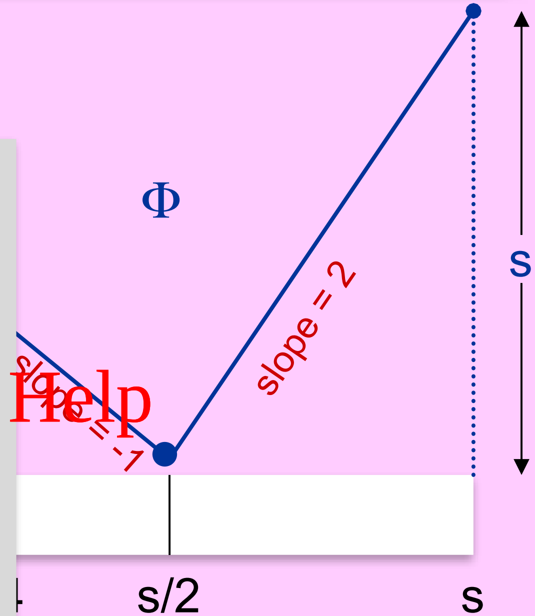
- Potential function:



$$\Phi(T) = \begin{cases} 2n(T) - s(T) & \text{if } \frac{1}{2} \leq \alpha(T) \leq 1 \\ -n(T) + s(T)/2 & \text{if } \frac{1}{4} \leq \alpha(T) \leq \frac{1}{2} \\ 0 & \text{if } 0 \leq \alpha(T) < \frac{1}{4} \end{cases}$$

Dynamic Table: Insert & Delete

Amortized cost for Insert:



se 1) $¼ \le \alpha < \alpha' \le ½$ $\quad$ $\hat{c}=c+\Delta\Phi = 1-1 = 0$

se 2) $\alpha < ½ < \alpha'$ $\quad$ $\hat{c}=c+\Delta\Phi \le 1+2 = 3$

se 3) $½ \le \alpha < 1$ $\quad$ $\hat{c}=c+\Delta\Phi = 1+2 = 3$

se 4) $\alpha = 1$

$$\Phi(T) = \begin{cases} 0 & \text{just after exp/con} \\ n(T) & \text{just before exp/con} \end{cases}$$

(needs expansion)

$\hat{c}=c+\Delta\Phi = [n+1] + [2-n] = 3.$

mmary: In all cases $\hat{c}=O(1)$.

Dynamic Table: Insert & Delete

Amortized cost for Delete:

case 1) $\frac{1}{2} \le \alpha' < \alpha \le 1$ $\hat{c} = c + \Delta\Phi = 1 - 2 = -1$

case 2) $\alpha' < \frac{1}{2} \le \alpha$ $\hat{c} = c + \Delta\Phi \le 1 + 0 = 1$

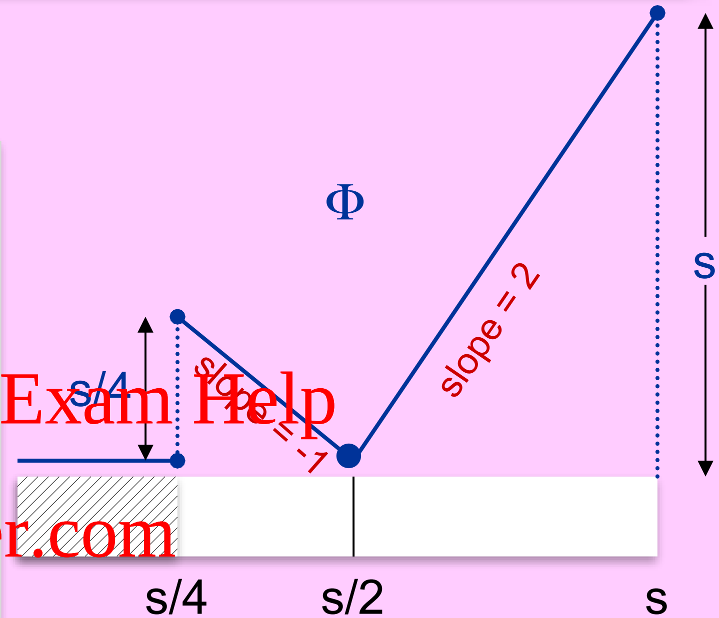
case 3) $\frac{1}{4} \le \alpha' < \alpha \le \frac{1}{2}$ $\hat{c} = c + \Delta\Phi = 1 + 1 = 2$

case 4) $\frac{1}{4} \approx \alpha$ ($\alpha' < \frac{1}{4}$ w/o contraction)

(needs contraction)

$\hat{c} = c + \Delta\Phi = n + [1 - n] = 1.$

Summary: In all cases $\hat{c} = O(1)$.



$$\Phi(T) = \begin{cases} 0 & \text{just after exp/con} \\ n(T) & \text{just before exp/con} \end{cases}$$

Competitiveness

on-line vs off-line algorithms

- off-line : complete input info in advance of any computation.
- on-line : partial input info. Additional piece-meal info provided as the on-line algorithm progresses through its computation.
- Question:

  How well can on-line algorithms do compared to off-line counter-parts?
- Application Areas:
  - Operating system process scheduler
  - Financial portfolio management in unpredictable financial market
  - Autonomous robot motion planning in unknown environment
  - On-line sequence of operations on a data structure
  - ∘ ∘ ∘
- Example Problems:
  - Ski Rental Problem
  - Cow Path Problem
  - K-Server Problem

Competitiveness of on-line algorithms

- A sequence of operations: $s = op_1, op_2, \ldots, op_n$.
- Off-line algorithm knows entire sequence s in advance of any computation.
- On-line algorithm must perform current op_i before next op_{i+1} is revealed to it.
- $C_A(s)$ = computational cost of algorithm A on sequence s.
- $C_{OPT}(s)$ = computational cost of optimum off-line algorithm on sequence s.
- Competitive ratio of on-line algorithm A:

$$\sigma_A = \max_s \frac{C_A(s)}{C_{OPT}(s)}$$

- σ_A could be a function of n.
- For any $\sigma \;\; \sigma_A$, we say algorithm A is σ-competitive.
- If $\sigma_A = O(1)$, we say A is competitive.

Ski Rental Problem

- A skier wants to ski for each of n days without knowing n in advance. Each day he should decide to buy a pair of skis (only once), or rent for that day.

  $1 = cost of renting a ski for one day ($r)

  $100 = cost of purchasing a pair of skis (P × $r)

  Minimize total cost over the n days.

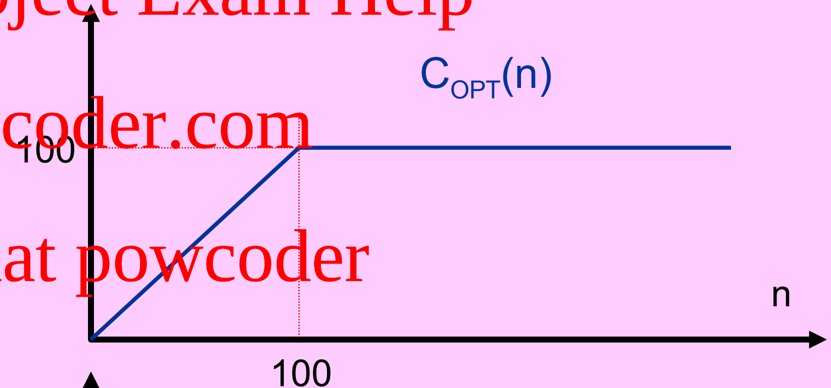
  What on-line buy/rent strategy is most competitive?

- Optimum off-line strategy:

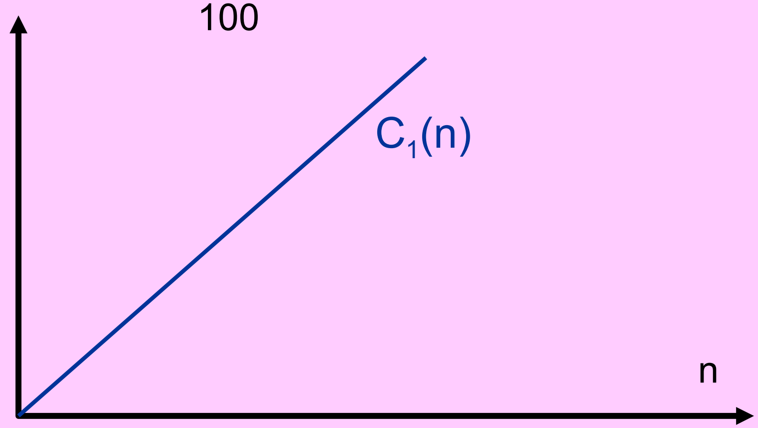
  Buy on day 1 if n 100.

  Otherwise, rent for n days.

  $C_{OPT}(n) = \min\{100, n\}$



- On-line strategy 1:
  Rent every day.



$\sigma_1 = \max_n C_1(n)/C_{OPT}(n) = \infty$

Ski Rental Problem

- A skier wants to ski for each of n days without knowing n in advance. Each day he should decide to buy a pair of skis (only once), or rent for that day.

  $1 = cost of renting a ski for one day ($r)

  $100 = cost of purchasing a pair of skis ($P \times \$r$)

  Minimize total cost over the n days.

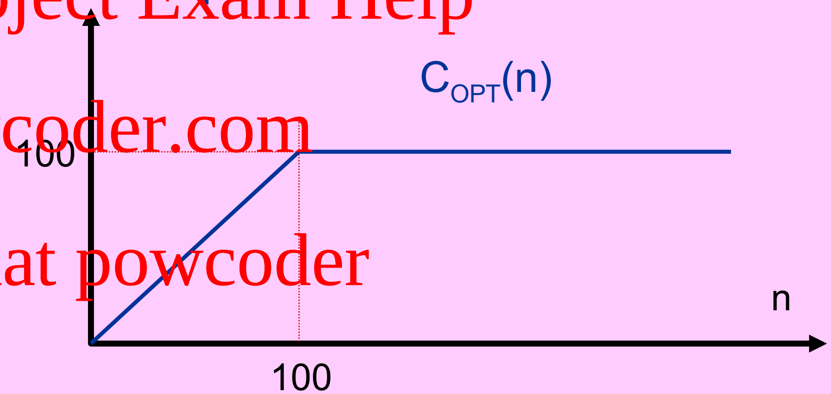
  What on-line buy/rent strategy is most competitive?

- Optimum off-line strategy:

  Buy on day 1 if n 100.

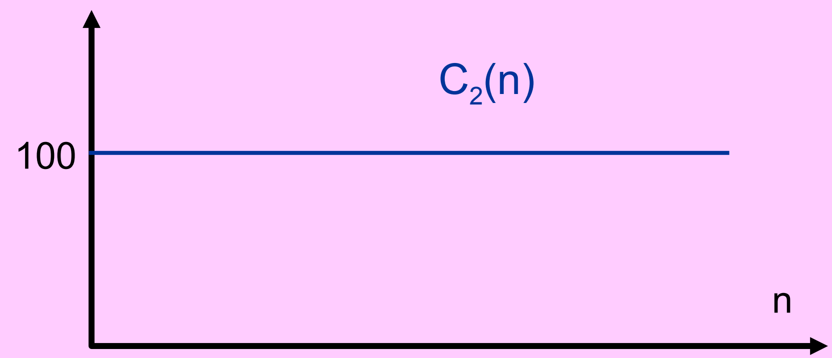
  Otherwise, rent for n days.

  $C_{OPT}(n) = \min\{100, n\}$



- On-line strategy 2:

  Buy on day 1.



$\sigma_2 = \max_n C_2(n)/C_{OPT}(n) = 100$

$= P$

Ski Rental Problem

- A skier wants to ski for each of n days without knowing n in advance. Each day he should decide to buy a pair of skis (only once), or rent for that day.

  $1 = cost of renting a ski for one day ($r)

  $100 = cost of purchasing a pair of skis (P × $r)

  Minimize total cost over the n days.

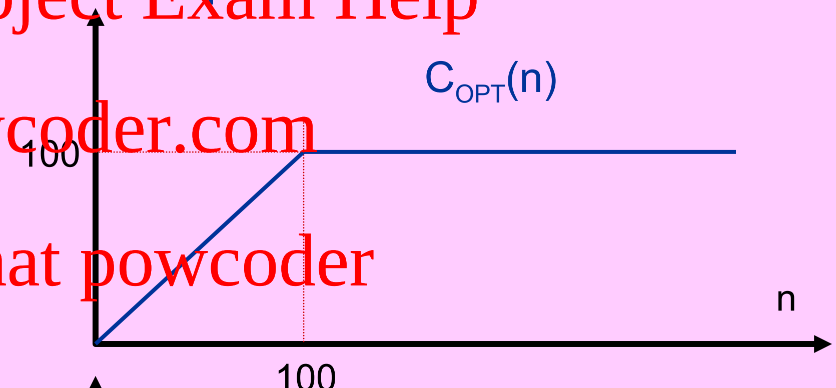
  What on-line buy/rent strategy is most competitive?

- Optimum off-line strategy:

  Buy on day 1 if n 100.

  Otherwise, rent for n days.

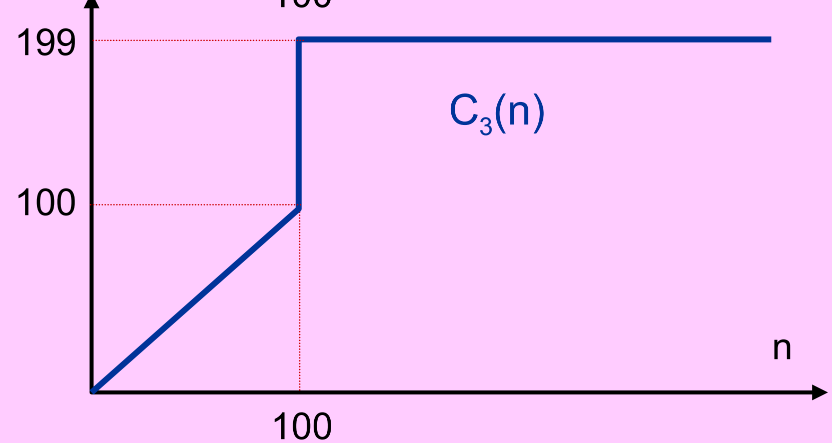
  $C_{OPT}(n) = \min\{100, n\}$



- On-line strategy 3:

  Rent the first min {99,n} days.

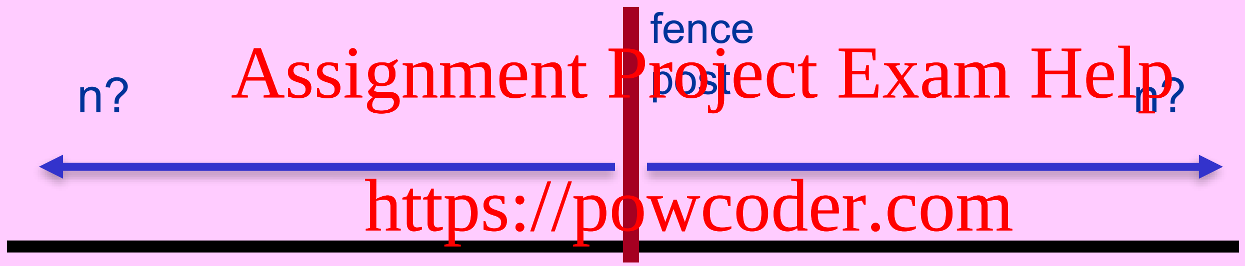
  Buy on day P=100 if n P=100.



$$\sigma_3 = \max_n C_3(n)/C_{OPT}(n) = 1.99$$
$$= 2 - 1/P$$

Cow Path Problem

- In the dark of night the farmer realizes his cow is not at the usual fence post. It has wondered off in unknown direction left or right an unknown distance of n meters away from the fence post.
  He has a flash light that shows only in front of his feet.
  Cost = # meters the farmer has to walk to find his cow.



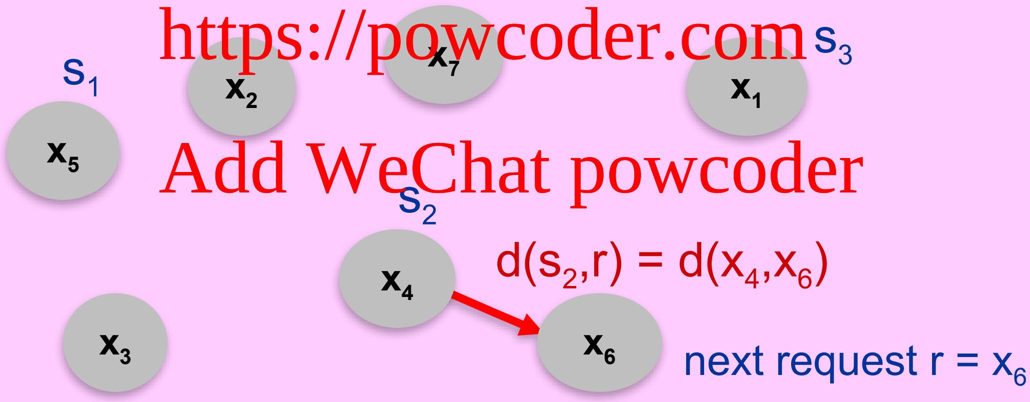
- What is the obvious optimum off-line strategy and its cost?
  (Only need to know which direction the cow went.)

- Question: Is there a competitive on-line strategy?
- Answer: Yes. Zig-zag repeated doubling.
- Show that this is a 9-competitive strategy.

- Randomized on-line: flip a coin on which direction to move each time. Gives slightly better expected competitive ratio.

K-Server Problem

- Customer locations $X = \{x_1, x_2, \ldots, x_m\}$ with metric inter-point distances $d(x_i, x_j)$, for all $i,j = 1..m$.
- k servers initially located at x_i, $i=1..k$.
- On-line service request stream $(r_1, r_2, \ldots, r_n)$, each $r_i \in X$.
- For each request site r, we must decide to move one of the servers from its current position, say s, to the requested point r, at a cost of $d(s,r)$.
- The goal is to minimize the total distance traveled by all servers over the entire request stream.



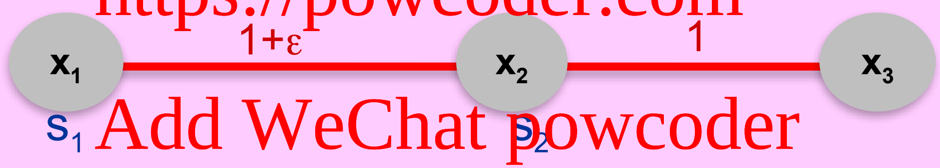
❑ **Exercise:** Optimum off-line algorithm in polynomial time
- by dynamic programming
- by reduction to the min-cost network flow problem.

Some on-line k-server algorithms

GREEDY: move the server “closest” to the request site, i.e., if the current request site is x, then select the server s that minimizes d(s,x).

Is GREEDY competitive? No. Not even for k=2.

Try request stream (x_3, x_2, x_3, x_2, x_3, x_2, …) below.



Some on-line k-server algorithms

HARMONIC: This is a randomized adaptation of GREEDY.
If the current request site is x, then select the server s with probability proportional to the inverse of d(s,x) (i.e., with probability $\alpha/d(s,x)$, where $1/\alpha = \Sigma_s 1/d(s,x)$).

[G] **Grove,** "The Harmonic online k-server algorithm is competitive," In Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, pp: 260-266, 1991.

Showed that in terms of expected cost, Harmonic is f(k)-competitive for the k-server problem, where $f(k) = 1.25k2^k - 2k$.

Some on-line k-server algorithms

BALANCE: Even out the total distance traveled by each server.

For each server s maintain its total distance D_s traveled so far.

If we select server s to serve x, it will add d(s,x) to D_s.

Select the server s that minimizes $D_s + d(s,x)$.

Is BALANCE competitive? No. Not even for k=2.

Try request stream ($x_4, x_3, x_2, x_1, x_4, x_3, x_2, x_1$ …) below.

Some on-line k-server algorithms

[MMS] Manasse, McGeoch and Sleator, "Competitive algorithms for server problems," Journal of Algorithms, 11, pp: 208-230, 1990.

Showed:

- BALANCE is k-competitive for k=|X|-1.
- Gave a 2-competitive algorithm for k=2. (A challenging exercise.)
- Proved that for no metric space is there a deterministic σ-competitive algorithm for the k-server problem with $\sigma < k$.

[IR] Irani and Rubinfeld, "A competitive 2-server algorithm," Information Processing Letters 39:85-91, 1991.

Showed:

- Modified-BALANCE: "minimize $D_s + 2\, d(s,x)$" is 10-competitive for k=2!

Note

Some on-line k-server algorithms

[CL] **Chrobak and Larmore,** "The server problem and on-line games," In Sleator and McGeoch (editors), On-line algorithms, volume 7 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp: 11-64. AMS, 1992.

- Proposed the **work-function** algorithm. This is an on-line dynamic programming strategy that determines the minimum work (or total cost) that transforms the initial state to any desirable final state.
- Computationally extensive: takes exponential time. But that is not part of the k-server's objective cost!
- Showed the work-function algorithm is 2-competitive for k=2.

[KP] **Koutsoupias and Papadimitriou,** "On the k-server conjecture," In Proceedings of the 26th Annual ACM Symposium on Theory of Computing, pp: 507-511, 1994.

- Proved the work-function algorithm is (2k–1)-competitive for all k.
- It is an open problem whether this algorithm is k-competitive.

[BEY] **Borodin and El-Yaniv,** "On-line computation and competitive analysis," Cambridge University Press, 1998.

- Is an interesting book on the subject.

Exercises

Recommendation:

Make a genuine effort on every exercise in this and the remaining Lecture Slides. They will reinforce your learning and induce a deeper level of understanding and mastery of the material.

Virtually all of your assignment questions and some of the test-exam questions may come from these sets of exercises.

1. **[CLRS, Exercise 17.2-1, page 458]** A sequence of stack operations is performed on a stack whose size never exceeds k. (Do not assume k is O(1) .) After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

2. **[CLRS, Exercise 17.2-3, page 459]** Suppose we wish not only to increment a counter but also reset it to zero (i.e., make all bits in it 0). Show how to implement a counter as an array of bits so that any sequence of n Increment and Reset operations takes time O(n) on an initially zero counter. [Hint: Keep a pointer to the high-order 1-bit.]

3. **[CLRS, Exercise 17.3-3, page 462]** We have a data structure *DS* that supports the operations *DSInsert* and *DSDelete* (that insert/delete a single item) in O(log n) worst-case time, where n is the number of items in *DS* before the operation. Define a potential function Φ such that the amortized cost of *DSInsert* is O(log n) and the amortized cost of *DSDelete* is O(1), and show that it works.

4. **[CLRS, Exercise 17.3-6, page 463]** Implement a queue by 2 stacks with O(1) amortized time per *Enqueue* and *Dequeue* operation. Analyze the amortized times by both the accounting method and the potential function method.

5. **[CLRS, Exercise 17.3-7, page 463]** Design a data structure to support the following two types of operations on an initially empty set S of real numbers:

   Insert(S,x) inserts item x into set S,

   DeleteLargerHalf(S) deletes the largest-valued $\lceil |S|/2 \rceil$ items from S.

   Explain how to implement this data structure with O(1) amortized cost per operation.

6. **[CLRS, Problem 17-2, page 473] Dynamizing Array Binary Search:**

Binary search of a sorted array takes logarithmic search time, but the time to insert a new item is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support Search and Insert on a set of n items. Let $k = \lceil \log(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0..k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively.

The total number of items held in all k arrays is therefore

$$\sum_{i=0}^{k-1} n_i 2^i = n.$$

Although each array is sorted, there is no particular relationship between items in different arrays.

(a) Describe how to perform the Search operation for this data structure.
Analyze its worst-case running time.

(b) Describe how to Insert a new item into this data structure.
Analyze its worst-case and amortized running times.

(c) Discuss how to implement Delete on this data structure.

7. **[Goodrich-Tamassia C-1.2. pp. 50-53]** Let $s = \langle op_1, op_2, \ldots, op_n \rangle$ be a sequence of n operations, each is either a red or blue operation, with op_1 being a red operation and op_2 being a blue operation. The running time of the blue operations is always constant. The running time of the first red operation is constant, but each red operation after that runs in time that is twice as long as the previous red operation in sequence s. What is the amortized time of the red and blue operations under the following conditions?
   (a) There are always $\Theta(1)$ blue operations between consecutive red operations.
   (b) There are always $\Theta(\ )$ blue operations between consecutive red operations.
   (c) The number of blue operations between a red operation op_i and the previous red operation op_j is always twice the number between op_j and its previous red operation.

8. **[Goodrich-Tamassia C-1.14. pp. 50-53]** Suppose you are given a set of small boxes, numbered 1 to n, identical in every respect except that each of the first i contain a pearl whereas the remaining $n - i$ are empty. You also have two magic wands that can each test if a box is empty or not in a single touch, except that a wand disappears if you test it on an empty box. Show that, without knowing the value of i, you can use the two wands to determine all the boxes containing pearls using at most o(n) wand touches. Express, as a function of n, the asymptotic number of wand touches needed. [Note: o(n) means strictly sub-linear.]

9. **[Goodrich-Tamassia C-1.25. pp. 50-53]** An evil king has a cellar containing n bottles of expensive wine, and his guards have just caught a spy trying to poison the king's wine. Fortunately, the guards caught the spy after he succeeded in poisoning only one bottle. Unfortunately, they don't know which one. To make matters worse, the poison the spy used was very deadly; just one drop diluted even a billion to one will still kill someone. Even so, the poison works slowly; it takes a full month for the person to die. Design a scheme that allows the evil king determine exactly which one of his wine bottles was poisoned in just one month's time while expending at most O(log n) of his taste testers.

10. **[Goodrich-Tamassia C-1.33. pp. 50-53]** In the dynamic table expansion scheme, instead of doubling the size (i.e., from size s to $2s$), we expand with $\lceil\ \rceil$ added size (i.e., from size s to $s + \lceil\ \rceil$). Show that performing a sequence of n insertions runs in $\Theta(n^{3/2})$ time in this case.

11. Dynamic Tables with pre-specified load factor range: Let c be an arbitrary given constant real number such that $0 < c < 1$. Show an expansion/contraction policy that maintains the load factor $\alpha(T)$ of a dynamic table T in the range $c \leq \alpha(T) \leq 1$ for which the operations insert and delete take *O(1)* amortized time. You should

(i) give the table expansion policy,
(ii) give the table contraction policy,
(iii) define the potential function for the amortized analysis, and
(iv) do the amortized analysis for the insert and delete operations.

[We showed a solution for c = ¼ . Show the general case following a similar approach.]

12. [Goodrich-Tamassia R-14.6, page 680] Consider the generalization of the ski rental problem where Alice can buy or rent her skis separate from her boots. Say that renting skis costs \$A, whereas buying skis costs \$B (A B). Likewise, say renting boots costs \$C, whereas buying boots costs \$D (C D). Describe a 2-competitive on-line algorithm for Alice to try to minimize the costs for going skiing subject to the uncertainty of her not knowing how many days she will continue to go skiing.

13. The Cow Path Problem: Show that the zig-zag repeated doubling strategy is 9-competitive.

14. The K-Server Problem: Design and analyze a 2-competitive on-line algorithm for the special case of the 2-server problem in 1-dimensional Euclidean metric space. That is, assume X (the customers) is a set of m points on the real line with Euclidean inter-point distances.

15. The Longest Smooth Subarray Problem:

Input: A Read-Only array $A[1..n]$ of n real numbers, and a positive real number D.

Output: The longest contiguous subarray S=A[$i..j$], $1 \le i \le j \le n$, such that no pair of elements of S has a difference more than D, that is,

$$\max(S) - \min(S) \le D .$$

Design and analyze an O(n) time algorithm to solve this problem.

[Note: array A is read-only, so you cannot rearrange its elements.

Hint: use Incremental algorithm and a variation of stack with multipop.]

16. The Largest D-Flat Subset and subarray problems:

Let D be a positive real number and S be a finite set of real numbers. We say S is **D-flat** if

$$\max(S) - \min(S) \le D \cdot (|S| - 1) .$$

(That is, the average difference between contiguous pairs of elements in the sorted permutation of S is ≤ D.)

Design and analyze efficient algorithms for the following problems:

(a) Given a set A of n elements and a D > 0, compute the largest D-flat subset of A.

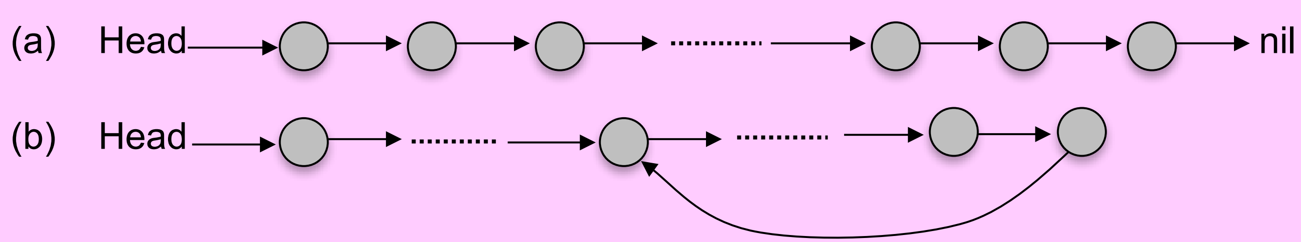
(b) Given an array A[1..n] of n elements and a D> 0, compute the longest D-flat contiguous subarray of A.

[Note: in part (a) any subset is a potential candidate solution, but in part (b) only those subsets that correspond to contiguous subarrays are allowed.

Hint: for part (b) use incremental algorithm and a variation of stack with multipop.]

17. Competitive On-line Linked-List Correctness Verification: We are given the head pointer to a read-only linked list. Each node of this list consists of a single field, namely, a pointer to the next node on the list. For the list to have correct structure, each node should point to its successor, except for the last node that points to *nil*, as in figure (a) below. However, due to a possible structural error, the list may look like that of figure (b), where a node points to one of the previous nodes (possibly itself). From that node on, we no longer have access to the rest of the nodes on the list. The task is to verify the structural correctness of the list. The off-line algorithm knows n, the number of accessible nodes on the list. So, it can easily verify the list's structural correctness, namely, starting from the head pointer, follow the list nodes for n steps and see if you reach a *nil* pointer. However, the on-line algorithm has no advance knowledge of the length of the list.

Design a competitive in-place on-line algorithm to test the structural correctness of the list. That is, your algorithm should take time linear in the number of accessible nodes, and work in-place, i.e., use only O(1) additional space. So, your algorithm may use a few local scalar variables, but not such things as additional lists or arrays. Also, the nodes on the list do not have any additional fields that you may use for some kind of marking, and your algorithm should not attempt to alter the structure of the list, not even temporarily.



END