AALBORG UNIVERSITY

Advanced Algorithms

Lecture 5
Amortized Analysis

Tung Kieu
tungkvt@cs.aau.dk

daisy
Center for Data-intensive Systems

ILO of Lecture 5

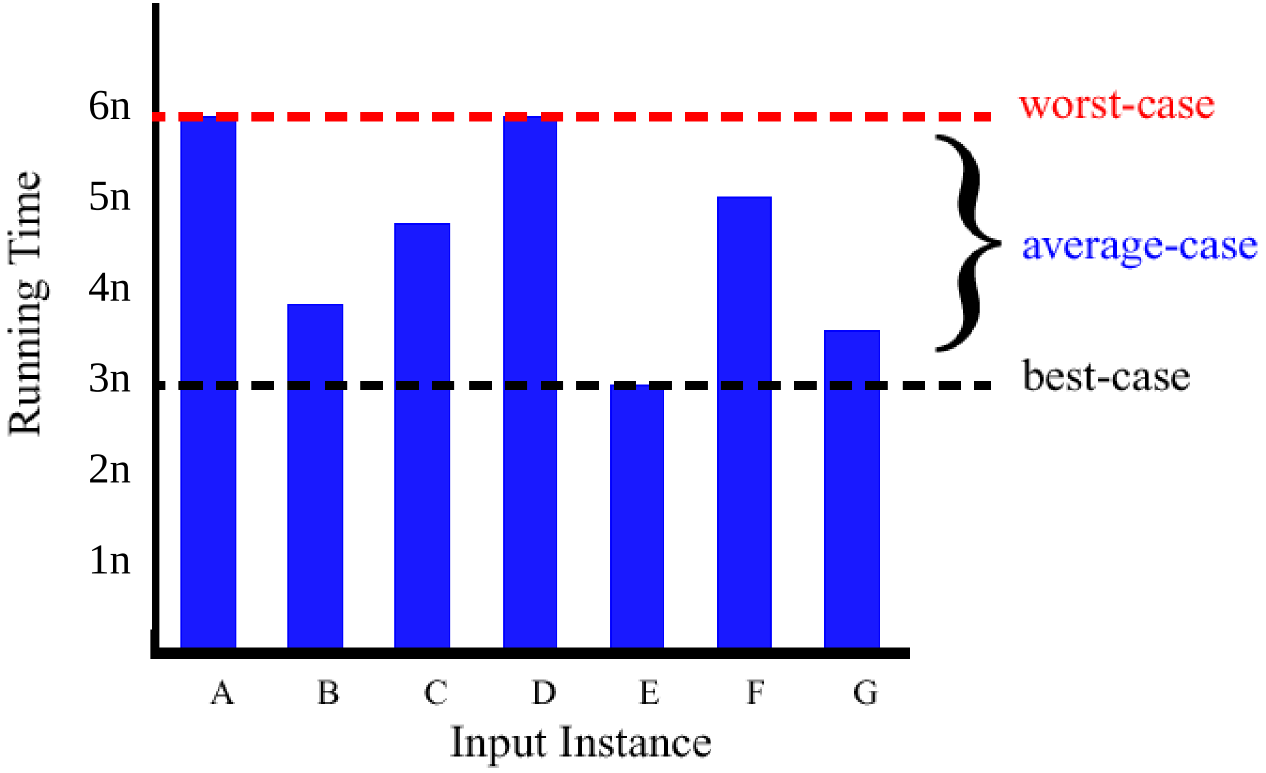
- Amortized analysis
  - to understand what is ***amortized analysis***, when is it used, and how it differs from the average-case analysis;
  - to be able to apply the techniques of ***the aggregate analysis***, ***the accounting method, and the potential method*** to analyze operations on simple data structures.

Agenda

- Amortized analysis
- Aggregate analysis
- Accounting method
- Potential method
- Dynamic tables

Best/Worst/Average Case (1)

- For a specific input size *n*, investigate running times T(n) for different input instances
  - Intuitive graphic illustration
    - A, B, C, …G are input instances of size *n*
  - Formal definitions?



Best/Worst/Average Case (2)

- Suppose algorithm *P* accepts *k* different input instances of size *n*. Let $T_i(n)$ be the time complexity of *P* on the *i*-th input instance, for $1 \le i \le k$, and p_i is the probability that the *i*-th instance occurs.
- Worst case time complexity: $W(n) = \max_{1\le i\le k} T_i(n)$
  - The ***maximum*** running time over all *k* inputs of size *n*
  - It is the most interesting/important!
- Average case time complexity: $A(n) = \Sigma_{1\le i\le k}\, p_i T_i(n)$
  - The ***expected*** running time over all *k* inputs of size n
  - Need assumptions about statistical distributions of input instances.
  - E.g., uniform distribution that each instance is equally likely.
- Best case time complexity: $B(n) = \min_{1\le i\le k} T_i(n)$
  - The ***minimum*** running time over all *k* inputs of size *n*
  - *Can be cheating*

Amortized Analysis

- The problem setting:
  - We have a **data structure**.
  - We perform a ***sequence of operations*** on the data structure.
  - Operations may be of different types (e.g., insertions, deletions).
  - Depending on the state of the structure the actual cost of an operation may differ (e.g., inserting into a sorted array).
  - Just analyzing the worst-case time of a single operation may not say too much.
  - We want the amortized running time of an operation.
- Amortized analysis vs. average-case analysis
  - ***Probability*** is not involved in amortized analysis but is involved in average-case analysis.

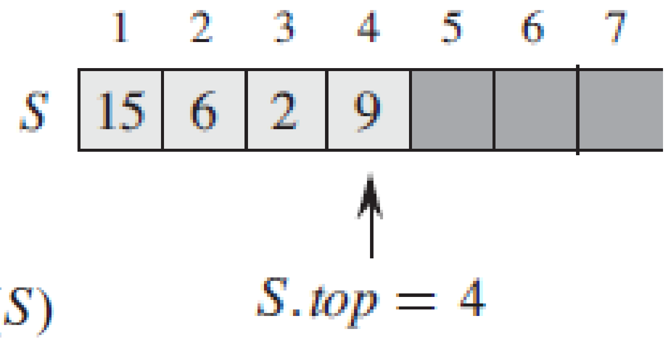
Example: stacks with multipop

- A **stack** is a container of objects. Objects are inserted and removed according to the last-in-first-out (**LIFO**) principle.
  - Sequence of elements $<a_1, a_2, \ldots, a_i>$, but only a_i is accessible as the “top” of the stack
- *Push(S, x)* inserts an element *x* into the stack S.
- *Pop(S)* pops/deletes the element on top of the stack S.
  - Does not take an element argument.
- Stack-Empty(S) returns whether the stack is empty.

- We introduce a new operation multipop(S, k)
  - Removes the k top objects of stack S.
  - Popping the entire stack if the stack contains fewer than k objects.

Operations on a stack

- Implement a stack of at most n elements using an array S[1..n].




```
STACK-EMPTY(S)
  if S.top == 0
      return TRUE
  else return FALSE
```

```
PUSH(S, x)
  S.top = S.top + 1
  S[S.top] = x
```

```
POP(S)
  if STACK-EMPTY(S)
      error "underflow"
  else S.top = S.top - 1
      return S[S.top + 1]
```

```
MULTIPOP(S, k)
  while not STACK-EMPTY(S) and k > 0
      POP(S)
      k = k - 1
```

Multipop

- An example
  - We have a stack S.
  - Multipop(S, 4)
  - Multipop(S, 5)

top → 23		
17		
6		
39		
10	top → 10	
47	47	

- Analysis
  - Assuming that there are s elements in stack S.
  - The while loop iterates min(s, k) times.
- Push, pop: O(1), constant time.
- Multipop: O(min(s, k))

MULTIPOP(S, k)

```
while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k - 1
```

A sequence of operations

- Consider a sequence of ***n*** operations on an initially empty stack.
  - An operation here can be push, pop, or multipop.
  - The worst-case cost of a multipop in the sequence is O(n).
  - Thus, the worst-case cost of any stack operation is O(n).
  - In total, we have *n* operations. Thus, $O(n^2)$.

- Observation:
  - We can pop each object from the stack at most once for each time we have pushed it into the stack.
  - The number of times that pop can be called on a nonempty stack, including the pop calls within multipop, is at most the number of push operations, which is at most n.
  - Thus, the total run-time of *n* operations is O(n).

Agenda

- Amortized analysis
- Aggregate analysis
- Accounting method
- Potential method
- Dynamic tables

Aggregate analysis

- Dividing total cost of n operations by n yields the average cost per operation, i.e., the amortized cost.
  - We do not consider the worst-case cost of each operation individually.
  - Instead, we consider the worst-case cost of a sequence of n operations, and then dividing it by the number of operations.

- Since the total run-time of n stack operations is O(n), then we say that all three stack operations have an amortized cost of O(1).
  - When not using amortized analysis, the worst-case cost of an stack operation is O(n).

- Here, we do not use probabilistic reasoning.
  - For example, we do not assume we have 20% pop operations, 50% push operations, and 30% multipop operations.
  - We actually consider a worst-case sequence of operations.
    - At most n elements can be inserted into an initially empty stack using n stack operations.

Incrementing a binary counter

- Consider a *k*-bit binary counter that counts upward from 0.
- An array A[0 .. k-1] of bits, where A.length=k, is used as the counter.
- A binary number *x* that is stored in the counter has its lowest-order bit in A[0] and the highest-order bit in A[k-1].
  - $x=\sum_{i=0}^{k-1} A[i] * 2^i$
  - A=1011, $x=1*2^3+0*2^2+1*2^1+1*2^0=11$

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

00000000,
00000001,
00000010,
00000011,
00000100,
00000101,
00000110,
00000111,
00001000

Analysis

- The cost of each increment operation is linear in the number of bits flipped.
- In the worst case, a single increment operation takes O(k).
  - Mini quiz: can you think when this happens?
- A sequence of n increment operations takes O(nk).
- Observation:
  - Not all bits flip each time when an increment operation is called.

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

Aggregate Analysis

- The bit in A[0]: flip every time, or every $1=2^0$ time.
- The bit in A[1]: flip every $2=2^1$ times.
- The bit in A[2]: flip every $4=2^2$ times.
- The bit in A[3]: flip every $8=2^3$ times.
- The bit in A[4]: flip every $16=2^4$ times.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

- A[i] flips every 2^i times.
- A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n increment operations.
  - Assume n = 10
    - A[0] flips 10 times.
    - A[1] flips 5 times.
    - A[2] flips 2 times.
    - A[3] flips 1 times.
    - A[4] flips 0 times.

Aggregate Analysis

- Based on the above, we have

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

In total, a sequence of n increment operations takes 2n flips, which is O(n).

Then, according to the aggregate analysis, the amortized cost per operation is O(n)/n=O(1), i.e., constant time.

Geometric series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a *geometric* or *exponential series* and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \qquad \text{(A.5)}$$

When the summation is infinite and $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} . \qquad \text{(A.6)}$$

Agenda

- Amortized analysis
- Aggregate analysis
- Accounting method
- Potential method
- Dynamic tables

Accounting method

- Charge different operations with different ***amortized costs***
  - Some operations charged **more** than they actually cost.
  - Some operations charged **less** than they actually cost.

- If ***amortized cost*** (denoated as $\hat{c}_i$) **>** actual cost (denoted as c_i), store the remained amount on **specific objects** as **credit.**

- If ***amortized cost*** $\hat{c}_i$ **<** actual cost c_i, **credit** is used to compensate.

- Requirement: total credit always ≥ 0 i.e., $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$
  - The total amortized cost $\sum_{i=1}^{n} \widehat{c_i}$ gives an ***upper bound*** on the total actual cost $\sum_{i=1}^{n} c_i$.

Stack with multipop

- Consider a sequence of n stack operations.

Operation	Actual cost	Amortized cost
push	1	2
pop	1	0
multipop(k)	min(k, s)	0

- When an object is pushed into the stack, pay 2:
  - 1 is paid for the actual cost of the push operation.
  - 1 is stored to spend when the object is popped out of the stack when being called by a pop operation.
- When an object is popped out of the stack, pay 0.
  - Pay its actual cost of the pop operation using the credit stored in the stack from pop operations.
- We charge multipop operations nothing.
  - If there is an element in the stack, it is always associated with a stored credit for paying the actual cost of a pop operation.

Total cost

Operation	Actual cost	Amortized cost
push	1	2
pop	1	0
multipop(k)	min(k, s)	0

Each stack operation at most takes cost 2.

Then, a sequence of n stack operations is at most 2n, thus O(n).

Incrementing a binary counter

Operation	Actual cost	Amortized cost
Assign a bit to 0 (line 3)	1	0
Assign a bit to 1 (line 6)	1	2

- When flip a bit from 0 to 1, we pay 2.
  - 1 is for the flip, the other 1 is a saving when flipping it back to 0.
- When flip a bit from 1 to 0, we pay 0.
  - At any point in time, every bit with 1 in the counter must have 1 credit in the savings, and thus we do not need to pay anything.

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

Each increment(A) operation can at most execute line 6 once, with cost 2.

Then, a sequence of n increment(A) operations will be at most 2n, thus O(n).

Agenda

- Amortized analysis
- Aggregate analysis
- Accounting method
- Potential method
- Dynamic tables

Potential method

- Consider the credit as potential stored within the ***entire data structure***, but not to specific objects (compared to the accounting method).
- Framework
  - D_0: an initial data structure.
  - D_i : the data structure after the i-th operation
  - c_i : actual cost of the i-th operation
- A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$.
- $\hat{c}_i$: amortized cost of the i-th operation
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$
- If we can define a potential function such that $\boldsymbol{\Phi(D_n) \geq \Phi(D_0)}$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

Stack operations

- We define the potential function Φ to return ***the number of objects*** in the stack.
- For an empty stack, we have $\Phi(D_0)=0$.
- Since the number of objects in the stack is never negative, we have
  - $\Phi(D_i) \geq 0 = \Phi(D_0)$
  - This satisfies the requirement $\Phi(D_n) \geq \Phi(D_0)$.
- Let's consider the three stack operations.
- If the i-th operation is a push operation on a stack having s objects
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $= 1 + (s+1) - s$
  - $=2$

Stack operations

- If the i-th operation is a pop operation on a stack having s objects
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $= 1 + (s-1) - s$
  - $=0$
- If the i-th operation is a multipop operation on a stack having s objects. Let's denote k'=min(k, s).
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $= k' + (s-k') - s$
  - $=0$
- So far, we have shown that the amortized cost of each of the three operation is constant O(1), and thus the total amortized cost of a sequence of n operations is O(n).
  - Since amortized cost is an upper bound of actual cost, the worst-case cost of n operations is therefore O(n).

Incrementing a binary counter

- We define the potential function Φ to be the number of 1s in the binary counter.
  - D_{i-1} = 00000111, Φ (D_{i-1}) =3
  - D_i = 00001000, Φ (D_i) =1
- In the beginning, the binary counter has all zeros, thus Φ (D_0) =0.
  - We have Φ (D_i) ≥ 0 = Φ (D_0)
  - This satisfies the requirement Φ(D_n) ≥ Φ(D_0).
- Assume that the i-th increment operation resets t_i bits, the actual cost of the i-th increment operation is t_i+1.
  - "Reset" means setting 1 to 0.

$c_i=t_i+1$

  - t_i : set t_i bits from 1 to 0.
  - 1: set one bit from 0 to 1.
  - E.g., 00000111 -> 00001000,
  - c_i=3+1=4

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

Incrementing a binary counter

- We distinguish two cases
- If Φ (D_i) >0, then Φ $(D_i) = \Phi$ (D_{i-1}) $-t_i+1$.
  - D_{i-1}=00000111, D_i=00001000, Φ $(D_{i-1})=3$, Φ $(D_i)=1$
  - 00000111 -> 00001000, Φ $(D_i) = \Phi$ (D_{i-1}) $-t_i+1=3-3+1=1$
- If Φ (D_i) =0, then the i-th increment operation resets all k bits, and so Φ $(D_{i-1}) = t_i = k$.
  - D_{i-1}=11111111, D_i=00000000, Φ $(D_{i-1})=8$, Φ $(D_i)=0$
  - 11111111->00000000, Φ (D_i) =0,Φ (D_{i-1}) =k,
  - Thus, we have Φ $(D_i) < \Phi$ (D_{i-1}) $-t_i+1$
- For both cases, we have $\boxed{\Phi (D_i) \leq \Phi (D_{i-1}) - t_i+1.}$
- The amortized cost is therefore
  - $\hat{c}_i = \boxed{c_i} + \Phi(D_i) - \Phi(D_{i-1})$
  - $= \boxed{t_i+1} + \boxed{\Phi(D_i)} - \Phi(D_{i-1})$
  - $\boxed{\leq} t_i+1 + \boxed{\Phi (D_{i-1}) - t_i+1} - \Phi(D_{i-1})$
  - $= 2$

$c_i=t_i+1$

Incrementing a binary counter

- We have shown that $\hat{c}_i = 2$.
- This means that the amortized cost of each increment operation is constant time O(1).
- Thus the total amortized cost of a sequence of n operations is O(n).
- Since amortized cost is an upper bound of actual cost, the worst-case cost of n operations is therefore O(n).

Agenda

- Amortized analysis
- Aggregate analysis
- Accounting method
- Potential method
- Dynamic tables

Dynamic tables

- It is often useful to have a dynamic table:
  - We do not always know in advance how many objects some applications need to store in a table.
- The table **expands** and **contracts** as necessary when new objects are added or deleted.
  - Expands when insertion is done and the table is already full
  - Contracts when deletion is done and there is “too much” free space
- Contracting or expanding involves relocating
  - Allocate new memory space of the new size.
  - Copy all elements from the table into the new space.
  - Free the old space.
- Worst-case time for insertions and deletions:
  - Without relocation: O(1)
  - With relocation: O(m), m is the number of objects in the table.

Some concepts

- Load factor α(T)= num/size
  - *num* – current number of objects in the table
  - *size* – the total number of objects that can be stored in the table
  - Empty table has load factor 0.
  - Full table has load factor 1.

- It would be nice to have these two properties:
  - Amortized cost of insert and delete is constant
  - The load factor is always above some constant
    - That means that the table is not too empty

Table expansion

- Let's only consider insertion first.
- Assume that a table is allocated as an array of slots.
- A table fills up when all slots have been used or, equivalently, when its load factor is 1.
- A common heuristic allocates a new table with twice as many slots as the old one.

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

If we perform n insertions, the worst-case cost of an insertion is O(n).

In total, $O(n^2)$.

Aggregate Analysis

- Actual cost for an insert operation c_i:
  - If the table is not full: **1.** Just insert the object.
  - If the table is full: **i.** Insert the previous i-1 objects into the new table and insert the object into the new table.
- Since in each expansion, we double the size of the table.
  - $2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6$
- Thus, for c_i
  - i: if i-1 is an exact power of 2, meaning that the table is already full.
  - 1: otherwise

i-1	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	8	9	10
Table size	1	2	4	4	8	8	8	8	16	16
Expansion		E	E		E				E	
c_i	1	2	3	1	5	1	1	1	9	1

Aggregate Analysis

- Consider a sequence of n insertions, we have the total actual cost

$$\boxed{\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j}$$
$$< n + 2n$$
$$= 3n ,$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

1. Every insertion, you need to insert a new object.
2. When expansions happen, also copy objects from an old table to a new table.

Amortized cost for each insertion is 3, i.e., O(1) constant time.

c_i = i: if i-1 is an exact power of 2
1: otherwise

i-1	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	8	9	10
c_i	1	2	3	1	5	1	1	1	9	1
	1	$1+2^0$	$1+2^1$	1	$1+2^2$	1	1	1	$1+2^3$	1

Accounting method

- How shall we understand the amortized cost for each insertion is 3? Why do we need to pay 3 for each insertion.
  - 1: for inserting the new object itself.
  - 1: for moving the new object itself when the table needs to expand in the future.
  - 1: for moving another object that has already been moved once when the table needs to expand in the future.

Example

- A (2)
- Expands: A(1), B(2)
- Expands: A(0), B(1), C(2)
- A(0), B(1), C(2), D(2)
- Expands: A(0), B(0), C(0), D(1), E(2)
- A(0), B(0), C(0), D(1), E(2), F(2), G(2), H(2)
- Expands: A(0), B(0), C(0), D(0), E(0), F(0), G(0), H(1), I(2)
- …

Objects that have already been moved once

- By paying 3 per insertion, you guarantee that you have enough money to move all objects in an old table to a newly allocated table.

Potential method

- $\Phi(T) = 2\ T.num - T.size$
- Idea:
  - we have a 0 potential immediately after an expansion;
  - the potential increases to the table size by the time when the table is full, so that you have enough potential to move all objects to a new table.
- When an insertion does not trigger an expansion.
  - $c_i=1$
  - $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$
  - $= 1 + 2\ T_i.num - T_i.size - (2\ T_{i-1}.num - T_{i-1}.size)$
  - $= 1 + 2\ (T_i.num - T_{i-1}.num) - (T_i.size - T_{i-1}.size)$
  - $= 1 + 2 * 1 - 0 = 3$

The two tables have the same size: $T_i.size = T_{i-1}.size$
Table T_i has one more object: $T_i.num = T_{i-1}.num + 1$

Potential method

- When an insertion triggers an expansion
  - $c_i = i$
  - $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$
  - $= c_i + 2\, T_i.\mathrm{num} – T_i.\mathrm{size} – (2\, T_{i-1}.\mathrm{num} – T_{i-1}.\mathrm{size})$
  - $= \boxed{c_i} + 2\, (\boxed{T_i.\mathrm{num} - T_{i-1}.\mathrm{num}}) - (\boxed{T_i.\mathrm{size} - T_{i-1}.\mathrm{size}})$
  - $= \boxed{T_{i-1}.\mathrm{num} + 1} + 2 * \boxed{1} – \boxed{T_{i-1}.\mathrm{size}}$
  - $= 3$
- Thus, constant amortized cost for an insertion.

- ***Actual cost c_i: move all objects in** T_{i-1} **and insert an new object. Thus, $c_i = T_{i-1}.num + 1$***
- ***The number of objects in T_i is one more than the number of objects in T_{i-1}. $T_i.num = T_{i-1}.num+1$.***
- ***The size of T_i is twice of the size of T_{i-1}: $T_i.size = 2*T_{i-1}.size$***
- ***Since we need to expand the table in the i-th insertion, then we have $T_{i-1}.num = T_{i-1}.size$.***

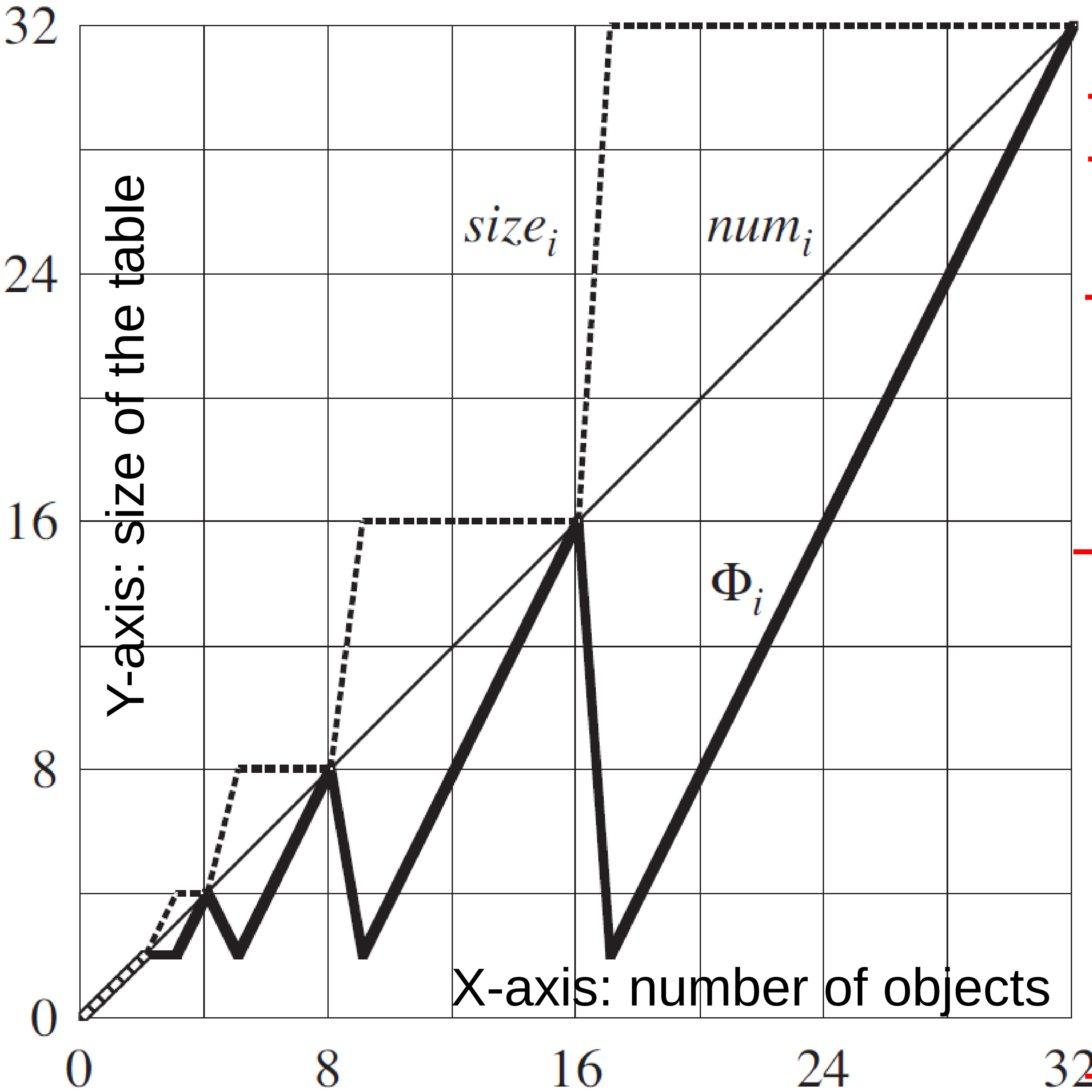
Mini-quiz (also on Moodle)

- Assume that we have a table T_{i-1} that has the following objects.
  - A, B, C, D, E, F, G, H
- In the i-th step, we insert I into table T_{i-1} and get T_i.
- What are the potential for T_i , the potential for T_{i-1} , and the amortized cost $\hat{c}_i$ for the i-th insertion?

- $\Phi(T) = 2$ T.num – T.size

Mini-quiz (also on Moodle)

- Assume that we have a table T_{i-1} that has the following objects.
  - A, B, C, D, E, F, G, H
- In the i-th step, we insert I into table T_{i-1} and get T_i.
- What are the potentials for T_i and T_{i-1} ?
- $\Phi(T) = 2\ T.num – T.size$
- T_i=2*9-16=2
- T_{i-1}=2*8-8=8
- $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1}) = (8+1)+2-8=3$



num	size	Φ
1	1	1
2	2	2
3	4	2
4	4	4
5	8	2
6	8	4
7	8	6
8	8	8
9	16	2
10	16	4
11	16	6
12	16	8
13	16	10
14	16	12
15	16	14
16	16	16
17	32	2

Mini-quiz

- What if we expand by a constant number of slots, but not double the size? Do we still get constant time amortized insertion? And why?

- Let’s say each expansion we add 4 more slots.

- Aggregate analysis
- $\sum_{i=1}^{n} c_i$ = n + 4+8+12+…+ $\lfloor n/4 \rfloor$ * 4
- = n + 4*(1+2+3+…+ $\lfloor n/4 \rfloor$)
- = n+2*(1+ $\lfloor n/4 \rfloor$)* $\lfloor n/4 \rfloor$
- =O(n^2)
- Each insertion is with linear amortized runtime O(n), but not constant anymore.

i-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c_i	1	1	1	1	5	1	1	1	9	1	1	1	13	1	1	1	17
	1	1	1	1	1+ 4	1	1	1	1+ 2*4	1	1	1	1+ 3*4	1	1	1	1+ 4*4

Contraction

- Let’s consider both insertions and deletions.
- When a table is full, double its size.
- When the number of objects in a table is less than ¼ of its size, then contract the table by halving its size.

insertions

deletions

step	num	size
1	1	1
2	2	2
3	3	4
4	4	4
5	5	8
6	6	8
7	7	8
8	8	8
9	9	16
10	8	16
11	7	16
12	6	16
13	5	16
14	4	16
15	3	8
16	2	8
17	1	4

Summary

- Aggregate analysis vs. accounting method
  - In aggregate analysis, all operations have same cost
  - In the accounting method, different operations can have different costs

- Accounting method vs. Potential method
  - Accounting method stores credit with specific objects
  - Potential method stores potential in the data structure as a whole

- Potential method is the most flexible one

ILO of Lecture 5

- Amortized analysis
  - to understand what is ***amortized analysis***, when is it used, and how it differs from the average-case analysis;
  - to be able to apply the techniques of the ***aggregate analysis***, ***the accounting method, and the potential method*** to analyze operations on simple data structures.

Lecture 6

- Computational geometry algorithms
  - Basic geometric operations
  - Sweeping techniques