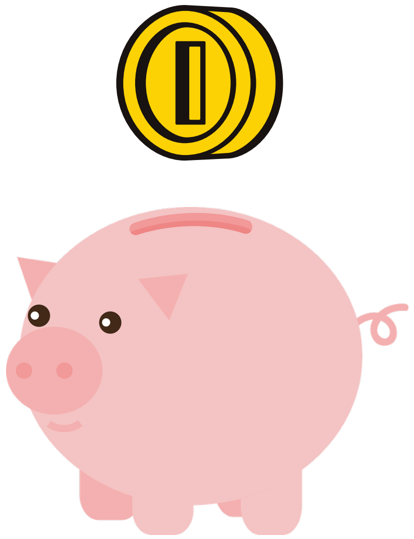
Amortized Analysis

Amortized Analysis



First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & 8 & 6 & 3 \\ \hline\end{array}$

$S = \begin{array}{|c|c|c|c|c|c|c}\hline & & & & & & \\ \hline\end{array}$

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$A =$

5	7	9	2	8	6	3

$S =$

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$A = \begin{array}{|c|c|c|c|c|c|c|}\hline \boxed{5} & 7 & 9 & 2 & 8 & 6 & 3 \\ \hline\end{array}$

$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & & & & & & \\ \hline\end{array}$

PUSH(S, x) : inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$A =$

5	$\boxed{7}$	9	2	8	6	3

$S =$

5						

PUSH(S, x) : inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

S = empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

$A =$

5	$\boxed{7}$	9	2	8	6	3

$S =$

5	7					

PUSH(S, x) : inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$A =$

5	7	9	2	8	6	3

$S =$

5	7					

PUSH(S, x) : inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$A =$

5	7	9	2	8	6	3

$S =$

5	7	9				

$\text{PUSH}(S, x)$: inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$A =$

5	7	9	$\boxed{2}$	8	6	3

$S =$

5	7	9				

PUSH(S, x) : inserts element x into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$A =$

5	7	9	2	8	6	3

$S =$

5	7	9				

```
MULTI-POP(S, k)
for i = 1 to k
        POP(S)
```

Push(S, x) : inserts element x into S

Pop(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline 5 & 7 & 9 & \boxed{2} & 8 & 6 & 3 \\ \hline \end{array}$$

$$S = \begin{array}{|c|c|c|c|c|c|c} \hline 5 & 7 & & & & & \\ \hline \end{array}$$

MULTI-POP(S, k)

```
for i = 1 to k
        POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & \boxed{2} & 8 & 6 & 3 \\ \hline\end{array}$

$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & & & & & & \\ \hline\end{array}$

MULTI-POP(S, k)

```
for i = 1 to k
        POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline 5 & 7 & 9 & \boxed{2} & 8 & 6 & 3 \\ \hline \end{array}$$

$$S = \begin{array}{|c|c|c|c|c|c|c} \hline 5 & 2 & & & & & \\ \hline \end{array}$$

```
MULTI-POP(S, k)
for i = 1 to k
        POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & \boxed{8} & 6 & 3 \\ \hline\end{array}$$

$$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & 2 & 8 & & & & \\ \hline\end{array}$$

MULTI-POP(S, k)

```
for i = 1 to k
        POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

$A =$

5	7	9	2	8	$\boxed{6}$	3

$S =$

5	2	8	6			

MULTI-POP(S, k)
for $i = 1$ **to** k
    POP(S)

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

$$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & 8 & 6 & \boxed{3} \\ \hline\end{array}$$

$$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & 2 & 8 & 6 & & & \\ \hline\end{array}$$

MULTI-POP(S, k)
for $i = 1$ **to** k
    POP(S)

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

$A =$

5	7	9	2	8	6	$\boxed{3}$

$S =$

5	2	8				

MULTI-POP(S, k)
for $i = 1$ **to** k
    POP(S)

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

S = empty stack
for $i = 1$ **to** A.*length*
  if $A[i] \leq S$.*size*
    MULTI-POP$(S, A[i])$
  PUSH$(S, A[i])$

$A =$

5	7	9	2	8	6	$\boxed{3}$

$S =$

5	2					

MULTI-POP(S, k)
for $i = 1$ **to** k
  POP(S)

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
    if A[i] ≤ S.size
        MULTI-POP(S, A[i])
    PUSH(S, A[i])
```

$$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & 8 & 6 & \boxed{3} \\ \hline \end{array}$$

$$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & & & & & & \\ \hline \end{array}$$

```
MULTI-POP(S, k)
for i = 1 to k
    POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

First Example: Multi-Pop Stack

What is the running time of this algorithm for an array A of length n?

```
S = empty stack
for i = 1 to A.length
        if A[i] ≤ S.size
                MULTI-POP(S, A[i])
        PUSH(S, A[i])
```

$A =$

5	7	9	2	8	6	$\boxed{3}$

$S =$

5	3					

MULTI-POP(S, k)

```
for i = 1 to k
        POP(S)
```

PUSH(S, x) : inserts element x into S

POP(S): removes and returns the element last inserted into S

Quiz: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

S = empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & 8 & 6 & \boxed{3} \\ \hline\end{array}$

$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & 3 & & & & & \\ \hline\end{array}$

MULTI-POP(S, k)
for $i = 1$ **to** k
    POP(S)

A: $\Theta(n)$

B: $\Theta(n \log n)$

C: $\Theta(n^2)$

Quiz: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

S = empty stack
for $i = 1$ **to** A.*length*
  if $A[i] \leq S$.*size*
    MULTI-POP$(S, A[i])$
  PUSH$(S, A[i])$

$A = \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 7 & 9 & 2 & 8 & 6 & \boxed{3} \\ \hline\end{array}$

$S = \begin{array}{|c|c|c|c|c|c|c}\hline 5 & 3 & & & & & \\ \hline\end{array}$

MULTI-POP(S, k)
for $i = 1$ **to** k
  POP(S)

A: $\Theta(n)$

B: $\Theta(n \log n)$

C: $\Theta(n^2)$

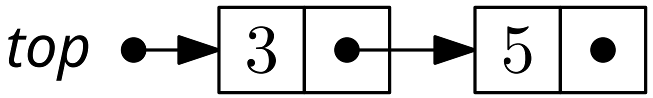
Simple Analysis: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        Multi-Pop$(S, A[i])$
    Push$(S, A[i])$

Multi-Pop(S, k)
for $i = 1$ **to** k
    Pop(S)

Push, Pop, *size*: $O(1)$ with simply linked list



size = 2

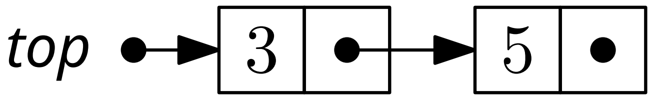
Simple Analysis: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size*
        MULTI-POP$(S, A[i])$
    PUSH$(S, A[i])$

MULTI-POP(S, k) $O(\min(k, S.\textit{size}))$
for $i = 1$ **to** k
    POP(S)

PUSH, POP, *size*: $O(1)$ with simply linked list



size = 2

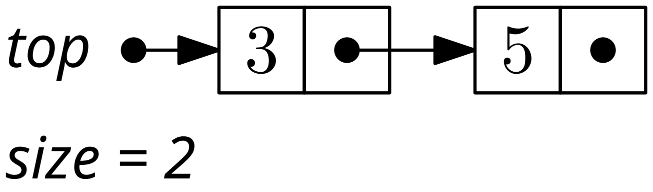
Simple Analysis: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

$S =$ empty stack
for $i = 1$ **to** A.*length*
  if $A[i] \leq S$.*size* $O(1)$
    MULTI-POP$(S, A[i])$ $O(i)$
  PUSH$(S, A[i])$ $O(1)$

MULTI-POP(S, k) $O(\min(k, S.\textit{size}))$
for $i = 1$ **to** k
  POP(S)

PUSH, POP, *size*: $O(1)$ with simply linked list



Simple Analysis: Multi-Pop Stack

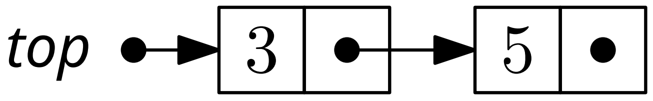
What is the running time of the this algorithm for an array A of length n?

$S =$ empty stack — $O(1)$
for $i = 1$ **to** A.*length*
    if $A[i] \leq S$.*size* — $O(1)$
        MULTI-POP$(S, A[i])$ — $O(i)$
    PUSH$(S, A[i])$ — $O(1)$

$\sum_{i=1}^{n} O(i) = O(n^2)$

MULTI-POP(S, k) $\quad O(\min(k, S.\textit{size}))$
for $i = 1$ **to** k
    POP(S)

PUSH, POP, *size*: $\quad O(1)$ with simply linked list



size = 2

Simple Analysis: Multi-Pop Stack

What is the running time of the this algorithm for an array A of length n?

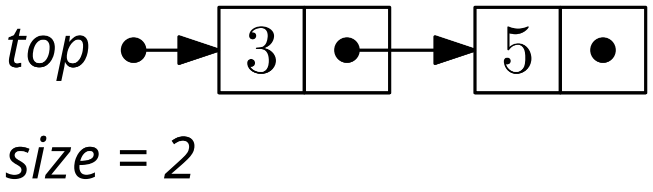
S = empty stack — $O(1)$
for $i = 1$ **to** A.*length*
    if $A[i] \le S$.*size* — $O(1)$
        MULTI-POP$(S, A[i])$ — $O(i)$
    PUSH$(S, A[i])$ — $O(1)$

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

too pessimistic!

MULTI-POP(S, k) $\quad O(\min(k, S.\textit{size}))$
for $i = 1$ **to** k
    POP(S)

PUSH, POP, *size*: $\quad O(1)$ with simply linked list



Amortized Analysis

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Amortized Analysis

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis
- Accounting method
- Potential method

Amortized Analysis

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis
- Accounting method
- Potential method

Examples:

- Multi-pop stack
- Binary counter
- Dynamic array

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of PUSH(S, x), POP(S) and MULTI-POP(S, k)

```
MULTI-POP(S, k)
1  for i = 1 to k
2          POP(S)
```

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of PUSH(S, x), POP(S) and MULTI-POP(S, k)

```
MULTI-POP(S, k)
for i = 1 to k
      POP(S)
```

Running time $T(n)$:

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of PUSH(S, x), POP(S) and MULTI-POP(S, k)

```
MULTI-POP(S, k)
1  for i = 1 to k
2      POP(S)
```

Running time $T(n)$:

- $\#$ PUSH Operationen $\leq n$: $O(n)$

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of Push(S, x), Pop(S) and Multi-Pop(S, k)

```
Multi-Pop(S, k)
1  for i = 1 to k
2        Pop(S)
```

Running time $T(n)$:

- $\#$ Push Operationen $\leq n$: $O(n)$
- $\#$ Pop $\leq \#$ Push $\leq n$: $O(n)$, including Pop in Multi-Pop!

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum") Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of PUSH(S, x), POP(S) and MULTI-POP(S, k)

```
MULTI-POP(S, k)
1  for i = 1 to k
2        POP(S)
```

Running time $T(n)$:

- $\#$ PUSH Operationen $\leq n$: $O(n)$
- $\#$ POP $\leq \#$ PUSH $\leq n$: $O(n)$, including POP in MULTI-POP!

Amortized running time per operation:

$$T(n)/n = O(n)/n = O(1)$$

Aggregate Analysis: Multi-Pop Stack

Aggregate („total sum”) Analysis:

- Compute worst-case running time $T(n)$ for sequence of n operations
- amortized cost of one operation: $\frac{T(n)}{n}$

Multi-Pop Stack

Given: Starting with an empty stack S, mixed sequence of n operations of PUSH(S, x), POP(S) and MULTI-POP(S, k)

```
MULTI-POP(S, k)
1  for i = 1 to k
2      POP(S)
```

Running time $T(n)$:

- $\#$ PUSH Operationen $\leq n$: $O(n)$
- $\#$ POP $\leq \#$ PUSH $\leq n$: $O(n)$, including POP in MULTI-POP!

Amortized running time per operation:

$$T(n)/n = O(n)/n = O(1)$$

$\rightarrow$ hence the $\Theta(n)$ runtime in first example

Accounting Method

Assign to each operation an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = assigned coins

Accounting Method

Assign to each operation an amortized cost (“coins”)

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = assigned coins

larger, equal or smaller than c_i

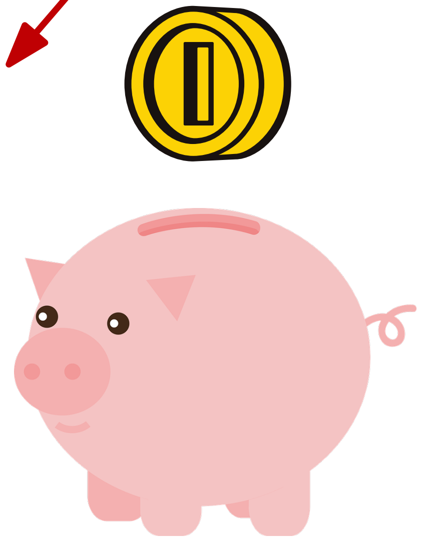
Accounting Method

Assign to each operation an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = assigned coins

- If $\hat{c}_i > c_i$: Save unused coins in D_i
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 is 0

larger, equal or smaller than c_i

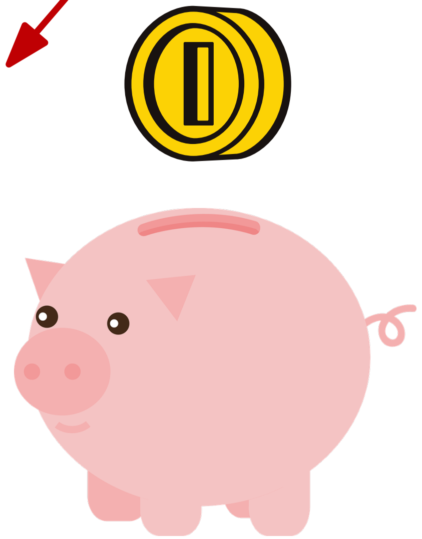


Accounting Method

Assign to each operation an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = assigned coins

larger, equal or smaller than c_i



- If $\hat{c}_i > c_i$: Save unused coins in D_i
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 is 0

cost invariant: saved coins ≥ 0

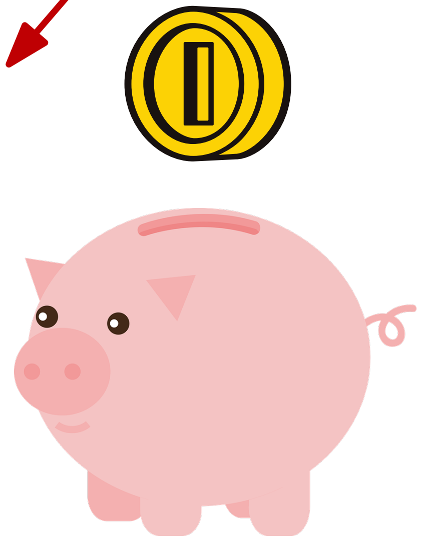
for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

Accounting Method

Assign to each operation an amortized cost ("coins")

- D_i = data structure after ith operation
- c_i = actual cost of ith operation
- $\hat{c}_i$ = amortized cost of ith operation = assigned coins

larger, equal or smaller than c_i



- If $\hat{c}_i > c_i$: Save unused coins in D_i
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 is 0

cost invariant: saved coins ≥ 0

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

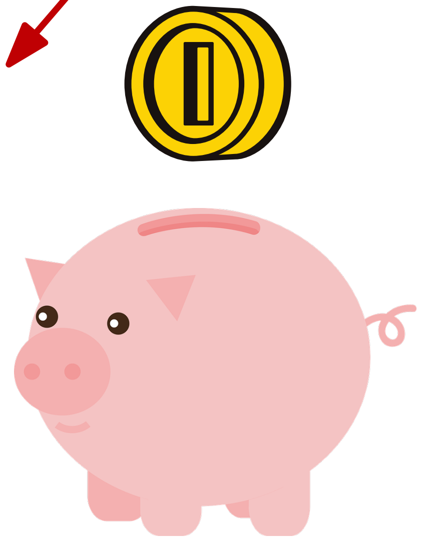
our task: choose amortized costs, such that invariant always holds

Accounting Method

Assign to each operation an amortized cost (“coins”)

- D_i $=$ data structure after ith operation
- c_i $=$ actual cost of ith operation
- $\hat{c}_i$ $=$ amortized cost of ith operation = assigned coins

larger, equal or smaller than c_i



- If $\hat{c}_i > c_i$: Save unused coins in D_i
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 is 0

cost invariant: saved coins ≥ 0

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

our task: choose amortized costs, such that invariant always holds

invariant implies: total cost of n operations $\leq$ sum of amortized costs

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

Accounting Method

Assign to each operation an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = assigned coins

larger, equal or smaller than c_i

- If $\hat{c}_i > c_i$: Save unused coins in D_i
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 is 0

cost invariant: saved coins ≥ 0

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

our task: choose amortized costs, such that invariant always holds

invariant implies: total cost of n operations $\leq$ sum of amortized costs

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

intuition: measure running time in coins („time is money")

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

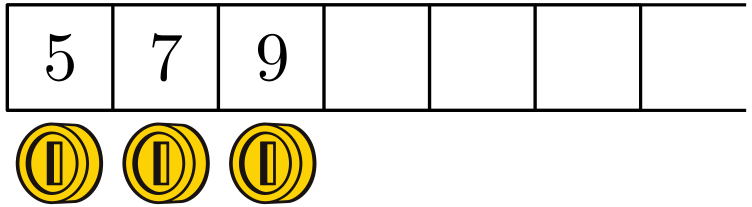
k coins per MULTI-POP(S, k) ($c_i = k$)

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin

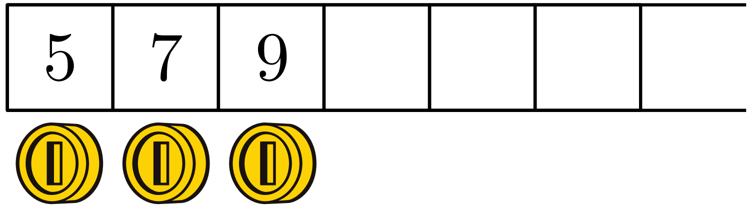


Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



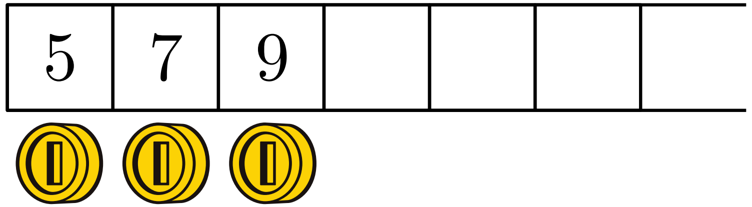
Accounting ($\hat{c}_i$):

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

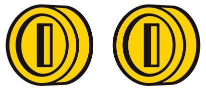
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



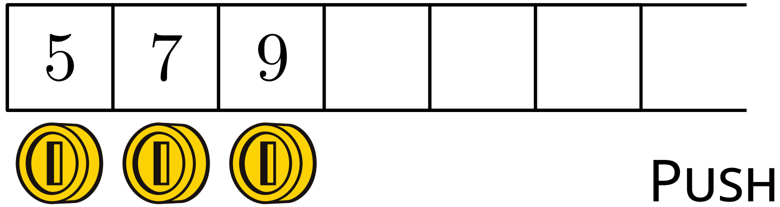
- 1 coin pays for PUSH of x
- 1 coin is “saved” with $x \rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

k coins per MULTI-POP(S, k) ($c_i = k$)

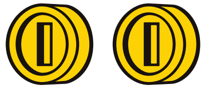
Invariant: Every element in the stack has a coin



PUSH($S, 8$)

Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



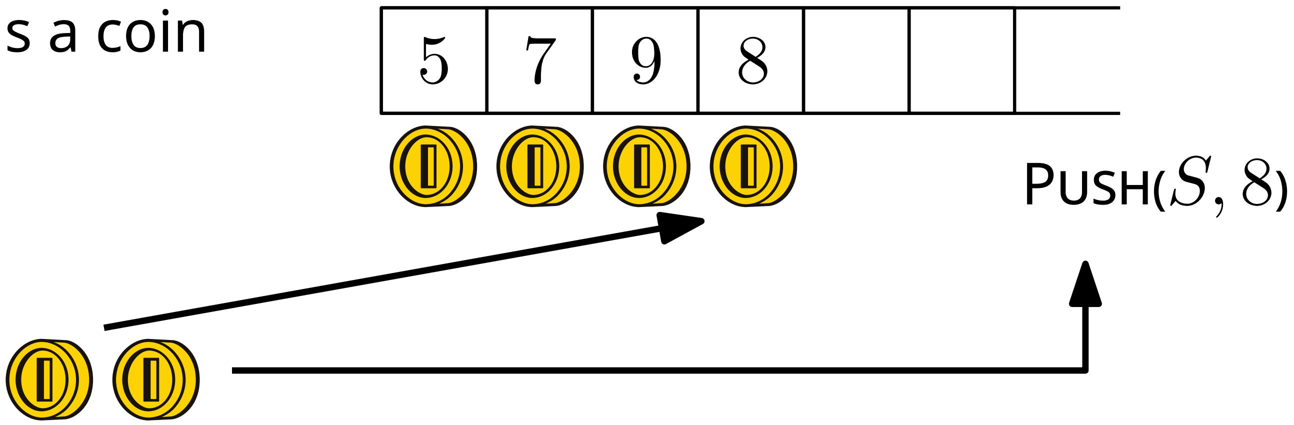
- 1 coin pays for PUSH of x
- 1 coin is “saved” with x $\rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

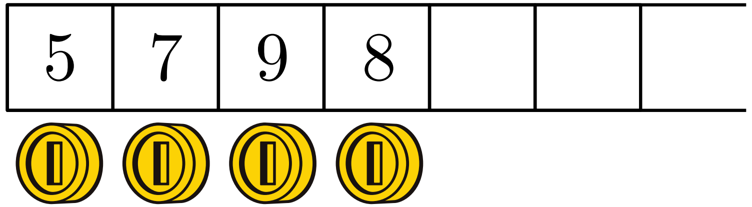
PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)

- 1 coin pays for PUSH of x
- 1 coin is “saved” with $x \to$ invariant maintained

Accounting Method: Multi-Pop Stack

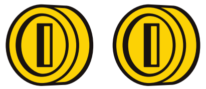
Actual costs per operation: 1 coin per Push or Pop ($c_i = 1$)
k coins per Multi-Pop(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

Push(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for Push of x
- 1 coin is “saved” with $x \rightarrow$ invariant maintained

Pop(S): Assign 0 coins ($\hat{c}_i = 0$)

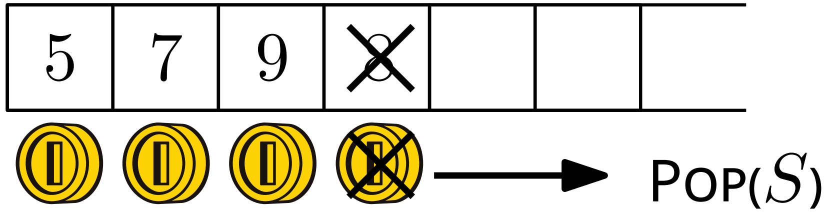
- coin saved with the removed element pays for Pop $\rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

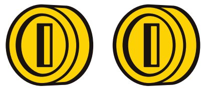
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for PUSH of x
- 1 coin is “saved” with $x \rightarrow$ invariant maintained

POP(S): Assign 0 coins ($\hat{c}_i = 0$)

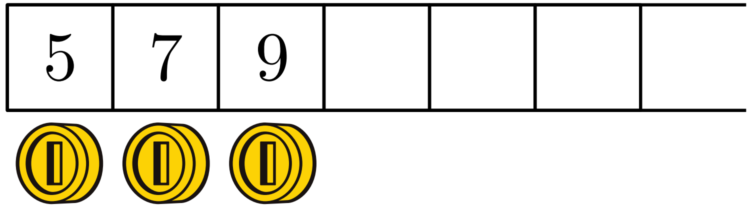
- coin saved with the removed element pays for POP $\rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per Push or Pop ($c_i = 1$)

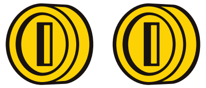
k coins per Multi-Pop(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

Push(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for Push of x
- 1 coin is "saved" with $x \rightarrow$ invariant maintained

Pop(S): Assign 0 coins ($\hat{c}_i = 0$)

- coin saved with the removed element pays for Pop $\rightarrow$ invariant maintained

How many coins do we need to assign to Multi-Pop?

A: 0

B: 1

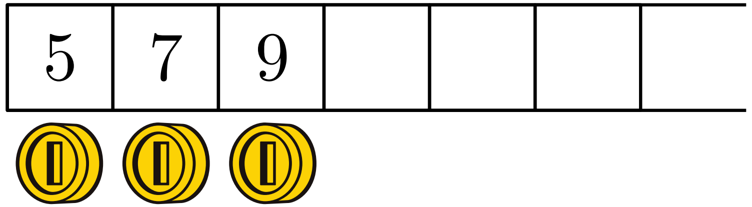
C: k

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

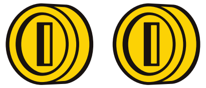
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for PUSH of x
- 1 coin is "saved" with $x \rightarrow$ invariant maintained

POP(S): Assign 0 coins ($\hat{c}_i = 0$)

- coin saved with the removed element pays for POP $\rightarrow$ invariant maintained

How many coins do we need to assign to MULTI-POP?

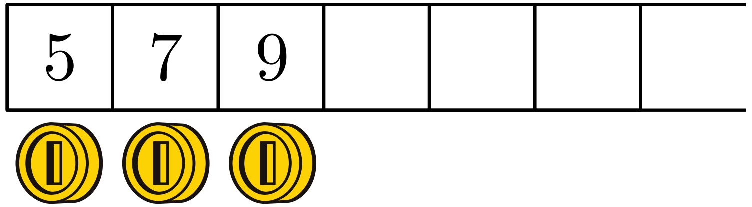
A: 0 B: 1 C: k

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

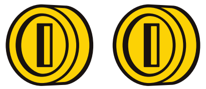
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for PUSH of x
- 1 coin is “saved” with $x \rightarrow$ invariant maintained

POP(S): Assign 0 coins ($\hat{c}_i = 0$)

- coin saved with the removed element pays for POP $\rightarrow$ invariant maintained

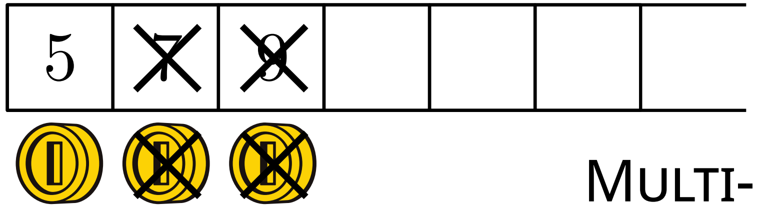
MULTI-POP(S, k): Assign 0 coins ($\hat{c}_i = 0$)

- saved coins pay for POPS $\rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per Push or Pop ($c_i = 1$)
k coins per Multi-Pop(S, k) ($c_i = k$)

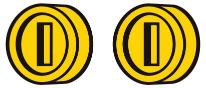
Invariant: Every element in the stack has a coin



Multi-Pop($S, 2$)

Accounting ($\hat{c}_i$):

Push(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



- 1 coin pays for Push of x
- 1 coin is “saved” with x $\rightarrow$ invariant maintained

Pop(S): Assign 0 coins ($\hat{c}_i = 0$)

- coin saved with the removed element pays for Pop $\rightarrow$ invariant maintained

Multi-Pop(S, k): Assign 0 coins ($\hat{c}_i = 0$)

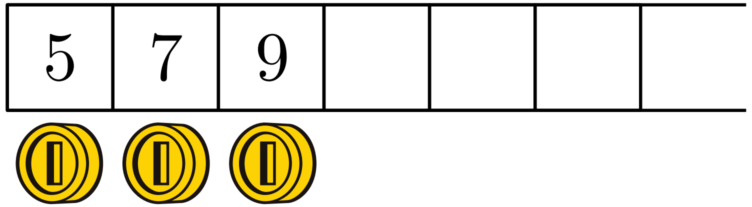
- saved coins pay for Pops $\rightarrow$ invariant maintained

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

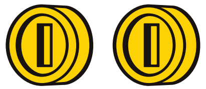
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



POP(S): Assign 0 coins ($\hat{c}_i = 0$)

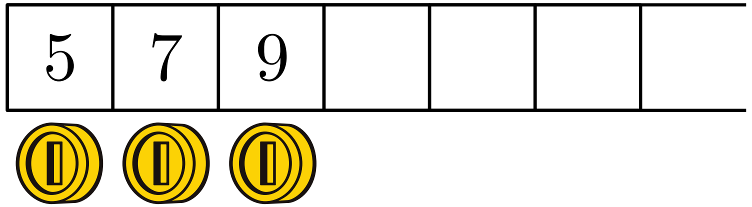
MULTI-POP(S, k): Assign 0 coins ($\hat{c}_i = 0$)

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per Push or Pop ($c_i = 1$)

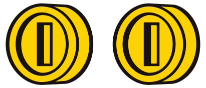
k coins per Multi-Pop(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

Push(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



Pop(S): Assign 0 coins ($\hat{c}_i = 0$)

Multi-Pop(S, k): Assign 0 coins ($\hat{c}_i = 0$)

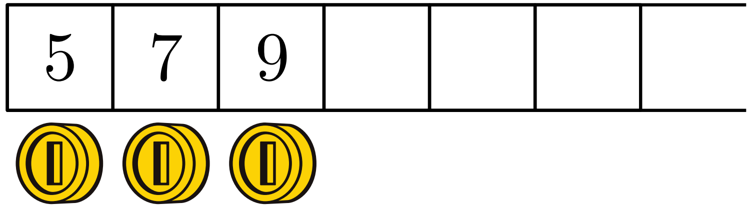
Running time of a sequence of n Push, Pop, Multi-Pop operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

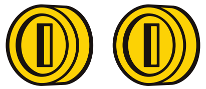
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



POP(S): Assign 0 coins ($\hat{c}_i = 0$)

MULTI-POP(S, k): Assign 0 coins ($\hat{c}_i = 0$)

Running time of a sequence of n PUSH, POP, MULTI-POP operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

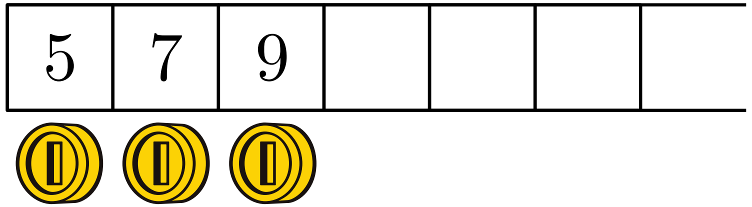
- invariant $\Rightarrow$ Number of coins in data structure $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$
- amortized cost per operation $\hat{c}_i \leq 2$

Accounting Method: Multi-Pop Stack

Actual costs per operation: 1 coin per PUSH or POP ($c_i = 1$)

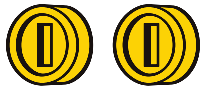
k coins per MULTI-POP(S, k) ($c_i = k$)

Invariant: Every element in the stack has a coin



Accounting ($\hat{c}_i$):

PUSH(S, x): Assign 2 coins ($\hat{c}_i$ = 2)



POP(S): Assign 0 coins ($\hat{c}_i = 0$)

MULTI-POP(S, k): Assign 0 coins ($\hat{c}_i = 0$)

Running time of a sequence of n PUSH, POP, MULTI-POP operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

- invariant $\Rightarrow$ Number of coins in data structure $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$
- amortized cost per operation $\hat{c}_i \leq 2$
- actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq \sum_{i=1}^{n} 2 = 2n$

Overview

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Overview

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis: Compute worst-case running time $T(n)$ for whole sequence
  amortized cost per operation: $T(n)/n$

Overview

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis: Compute worst-case running time $T(n)$ for whole sequence
  amortized cost per operation: $T(n)/n$
- Accounting method: coins can pay for (later) operations
  invariant coins saved in the data structure
  accounting
  - amortized cost of operations = coins assigned
  - how to maintain invariant

Overview

Amortized Analysis:

- Consider a sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis: Compute worst-case running time $T(n)$ for whole sequence
  amortized cost per operation: $T(n)/n$
- Accounting method: coins can pay for (later) operations
  invariant coins saved in the data structure
  accounting
    amortized cost of operations = coins assigned
    how to maintain invariant

Examples:

- multi-pop stack ✓
- binary counter
- dynamic array

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

example: $k = 6$

flipped bits

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0

example: $k = 6$

flipped bits

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

costs. number of bits flipped per operation ($= O(k)$)

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

costs. number of bits flipped per operation ($= O(k)$)

running time. worst-case running time of sequence of n increments is $O(kn)$.

Binary Counter

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

costs. number of bits flipped per operation ($= O(k)$)

running time. worst-case running time of sequence of n increments is $O(kn)$. ← too pessimistic!

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

How often is $A[i]$ flipped?

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

$A[2]$: $\lfloor n/4 \rfloor$ times

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

$A[2]$: $\lfloor n/4 \rfloor$ times

$\ldots$

$A[i]$: $\lfloor n/2^i \rfloor$ times

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

$A[2]$: $\lfloor n/4 \rfloor$ times

$\cdots$

$A[i]$: $\lfloor n/2^i \rfloor$ times

total cost

$$= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$$

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

$A[2]$: $\lfloor n/4 \rfloor$ times

$\cdots$

$A[i]$: $\lfloor n/2^i \rfloor$ times

total cost

$$= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$$

$$\leq \sum_{i=0}^{k-1} n/2^i$$

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

How often is $A[i]$ flipped?

$A[0]$: n times

$A[1]$: $\lfloor n/2 \rfloor$ times

$A[2]$: $\lfloor n/4 \rfloor$ times

$\cdots$

$A[i]$: $\lfloor n/2^i \rfloor$ times

total cost

$$= \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$$
$$\leq \sum_{i=0}^{k-1} n/2^i$$
$$< n \sum_{i=0}^{\infty} 1/2^i$$
$$= 2n$$

Aggregate Analysis: Binary Counter

Aggregate Analysis. Analyze worst-case running time $T(n)$ of n increment operations.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

The worst-case running time $T(n)$ of a sequence of n increments (starting from 0) is $O(n)$.
The amortized running time of one increment is $T(n)/n = O(1)$

Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: ???

accounting/amortized cost ($\hat{c}_i$): ???

Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: ???

accounting/amortized cost ($\hat{c}_i$): ???

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: ???

accounting/amortized cost ($\hat{c}_i$): ???

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

Which of the following invariants is suitable? (I.e., where do we need coins?)

Accounting Method: Binary Counter

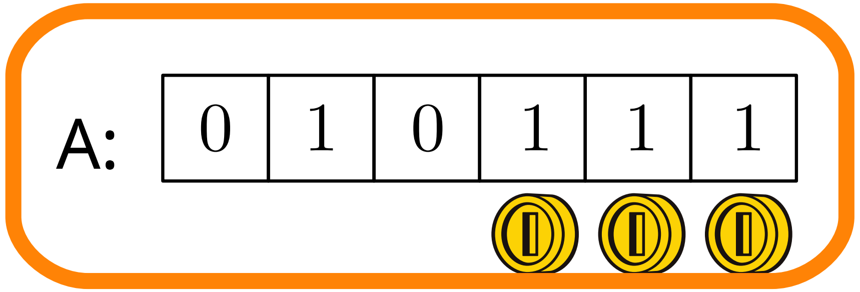
actual cost per operation: 1 coin per bit flipped

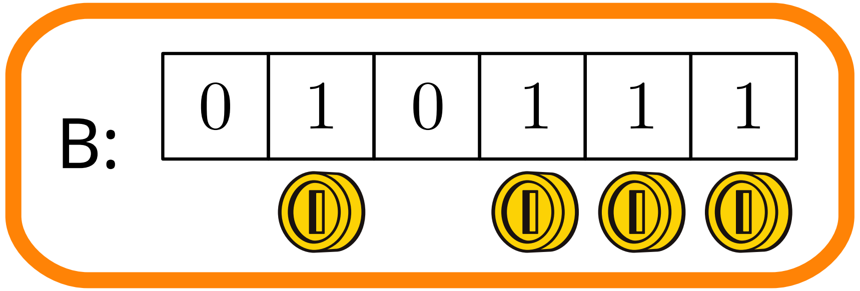
invariant: ???

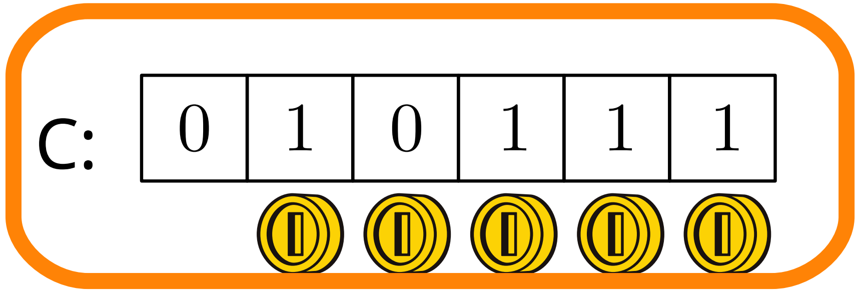
accounting/amortized cost ($\hat{c}_i$): ???

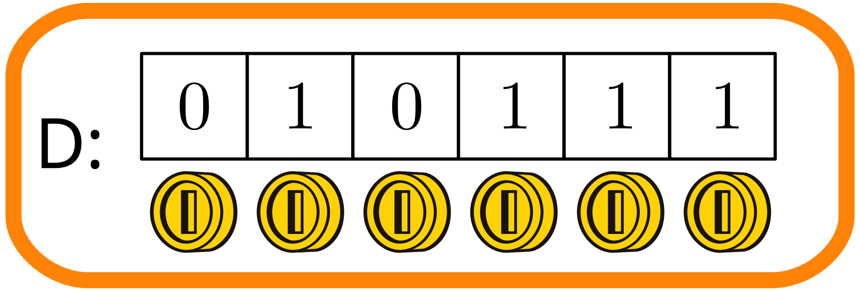
	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

Which of the following invariants is suitable? (I.e., where do we need coins?)









Accounting Method: Binary Counter

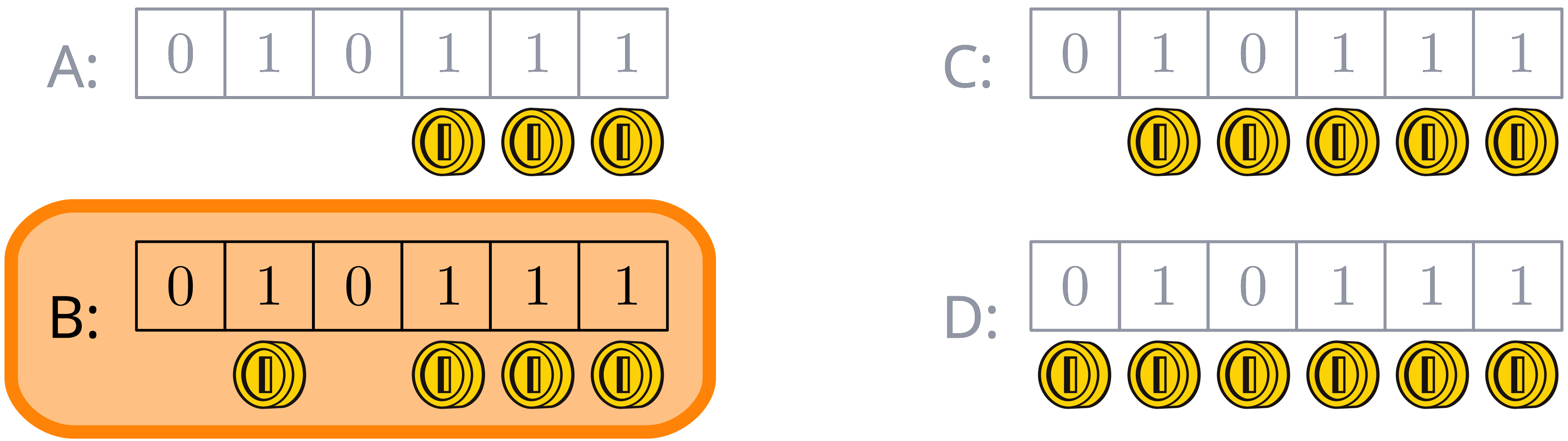
actual cost per operation: 1 coin per bit flipped

invariant: ???

accounting/amortized cost ($\hat{c}_i$): ???

	5	4	3	2	1	0
23 :	0	1	0	1	1	1

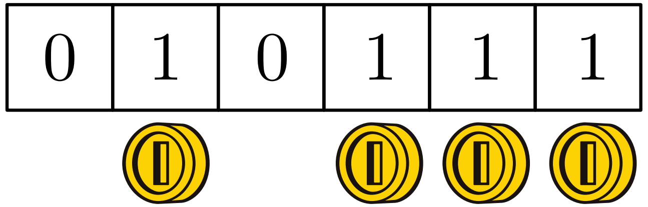
Which of the following invariants is suitable? (I.e., where do we need coins?)



Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin

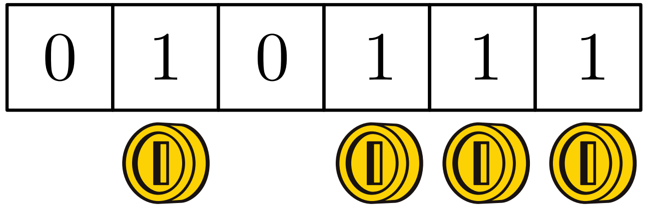


accounting/amortized cost ($\hat{c}_i$): ???

Accounting Method: Binary Counter

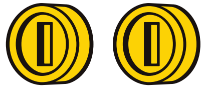
actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

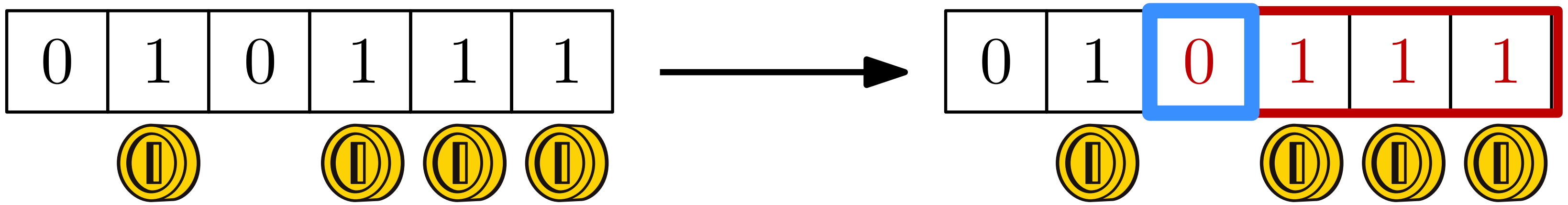
increment: assign 2 coins ($\hat{c}_i$ = 2)



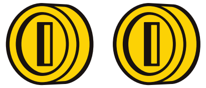
Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

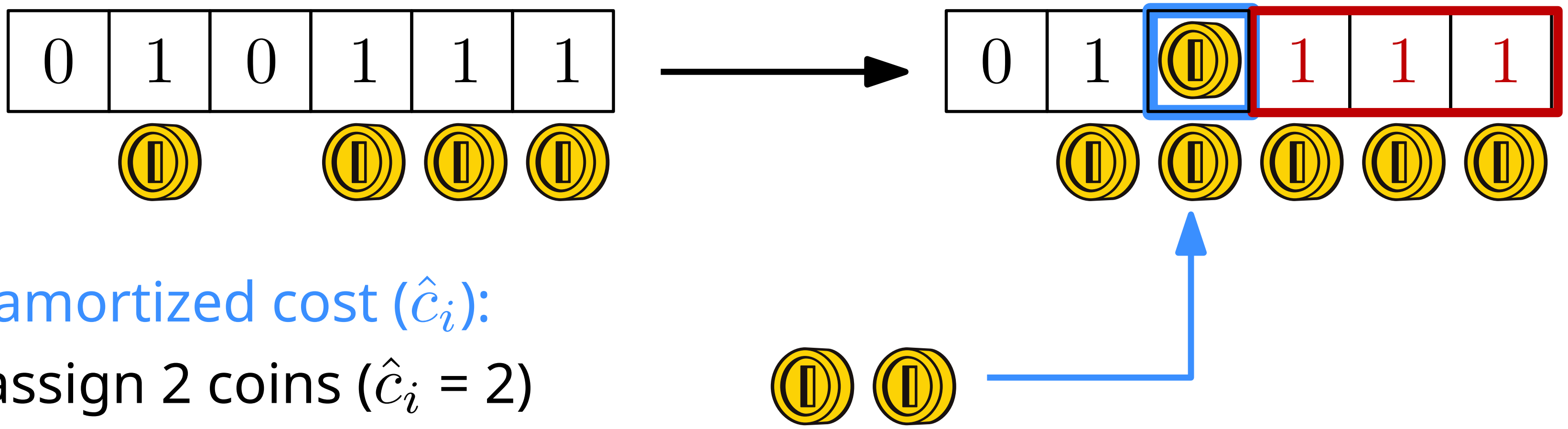
increment: assign 2 coins ($\hat{c}_i$ = 2) 

- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1

Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

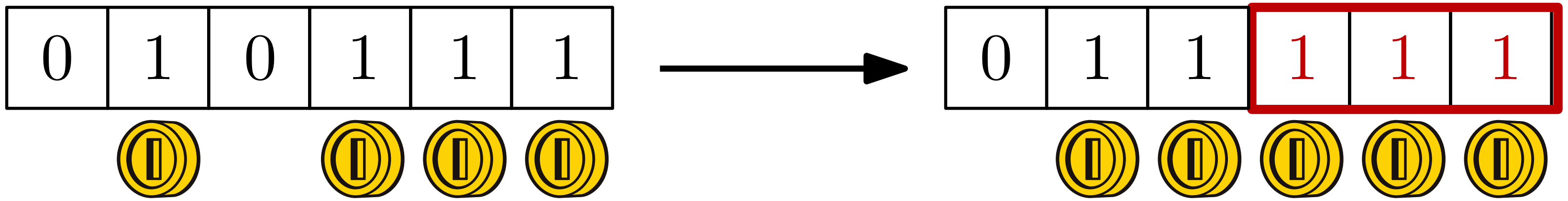
increment: assign 2 coins ($\hat{c}_i$ = 2)

- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1

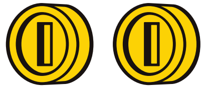
Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

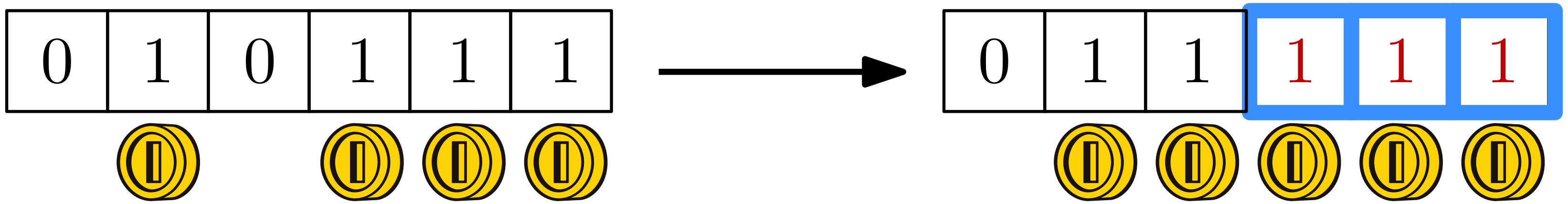
increment: assign 2 coins ($\hat{c}_i$ = 2) 

- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1

Accounting Method: Binary Counter

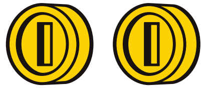
actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2)

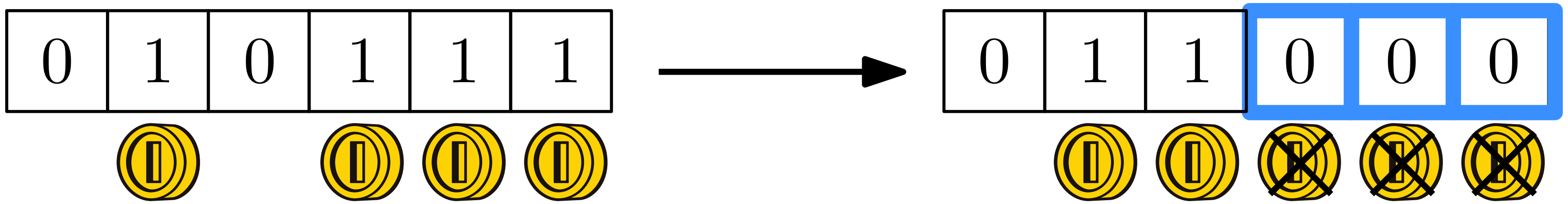


- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1
- to flip a 1 to 0: use saved coin

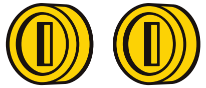
Accounting Method: Binary Counter

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

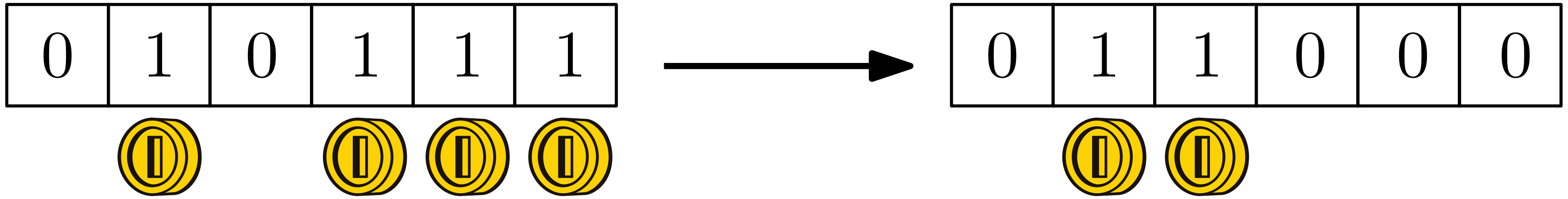
increment: assign 2 coins ($\hat{c}_i$ = 2) 

- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1
- to flip a 1 to 0: use saved coin

Accounting Method: Binary Counter

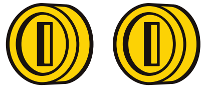
actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2)



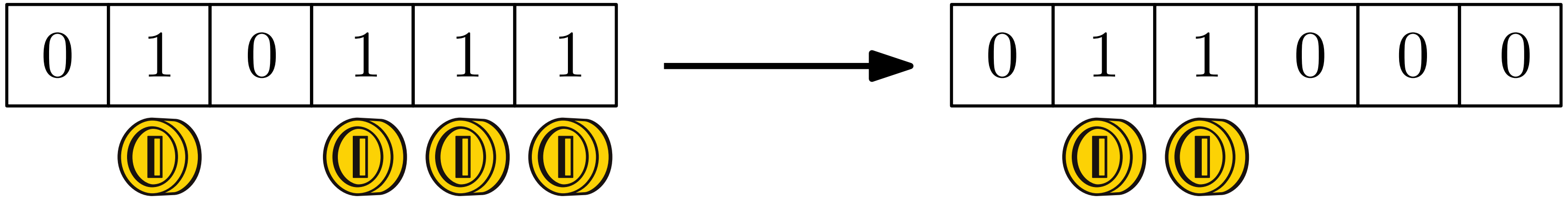
- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1
- to flip a 1 to 0: use saved coin

The worst-case running time $T(n)$ of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

Accounting Method: Binary Counter

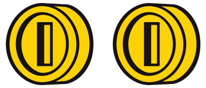
actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2)



- exactly 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with the new 1
- to flip a 1 to 0: use saved coin

The worst-case running time $T(n)$ of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

- invariant $\Rightarrow$ number of coins in data structure $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$
- amortized cost per operation $\hat{c}_i \leq 2$
- actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq \sum_{i=1}^{n} 2 = 2n$

Overview

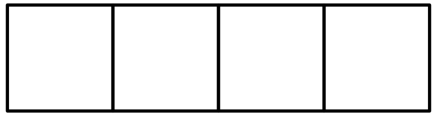
Methods:

- Aggregate analysis
- Accounting method

Examples:

- Multi-pop stack ✓
- Binary counter ✓
- dynamic array: insert, accounting method

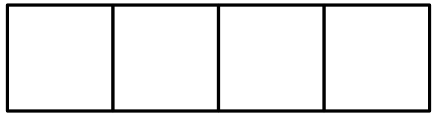
Dynamic Array: Insert

(static) array 

+ random-access, compact

- fixed size: problem for priority queues (heaps), hash tables, . . .

Dynamic Array: Insert

(static) array 

+ random-access, compact

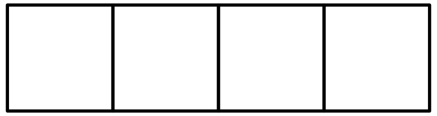
- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

Dynamic Array: Insert

(static) array



+ random-access, compact

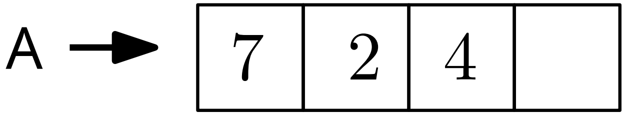
- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

implementation

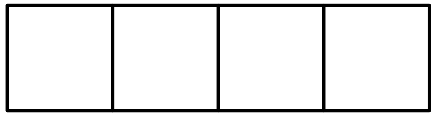
- use static array



append: $9, 3, \ldots$

Dynamic Array: Insert

(static) array



+ random-access, compact

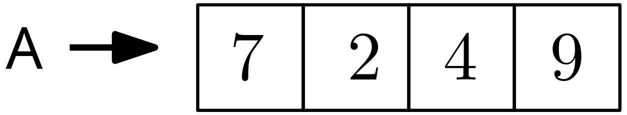
- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

implementation

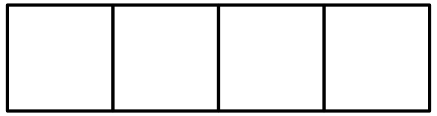
- use static array



append: $9, 3, \ldots$

Dynamic Array: Insert

(static) array



+ random-access, compact

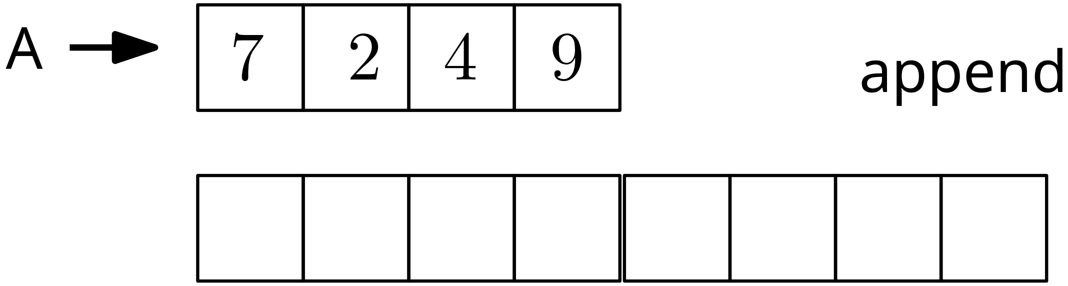
- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

implementation

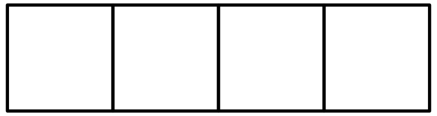
- use static array
- if full: create array of twice the size



append: $9, 3, \ldots$

Dynamic Array: Insert

(static) array



+ random-access, compact

- fixed size: problem for priority queues (heaps), hash tables, . . .

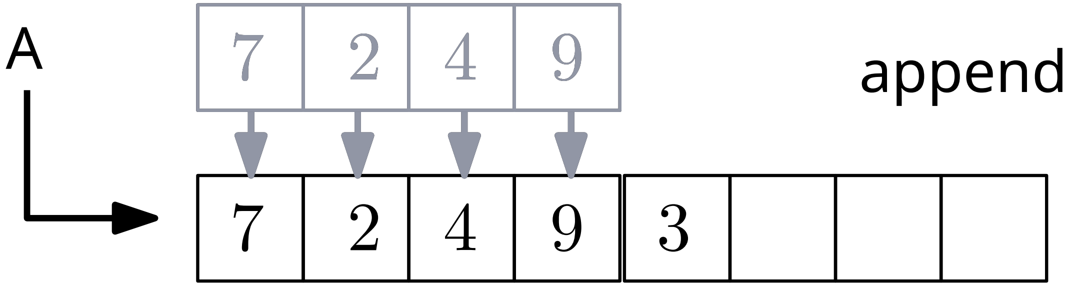
dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

implementation

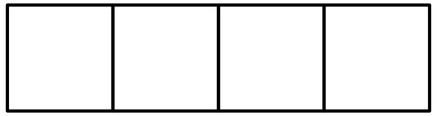
- use static array
- if full: create array of twice the size

  copy elements to new array (and delete old one)



append: $9, 3, \ldots$

Dynamic Array: Insert

(static) array 

+ random-access, compact

- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

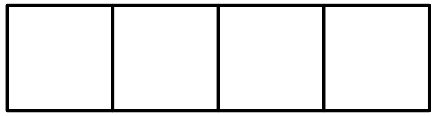
implementation

- use static array
- if full: create array of twice the size

  copy elements to new array (and delete old one)

cost per insertion: $O(n)$ if $n = 1+$ power of 2, otherwise $O(1)$

Dynamic Array: Insert

(static) array 

+ random-access, compact

- fixed size: problem for priority queues (heaps), hash tables, . . .

dynamic array with append (insert at end)

frequently used: java ArrayList, C++ vector, Python List, . . .

implementation

- use static array
- if full: create array of twice the size

  copy elements to new array (and delete old one)

too pessimistic!

cost per insertion: $O(n)$ if $n = 1+$ power of 2, otherwise $O(1)$

simple analysis: the running time of n insert operations is $O(n^2)$.

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

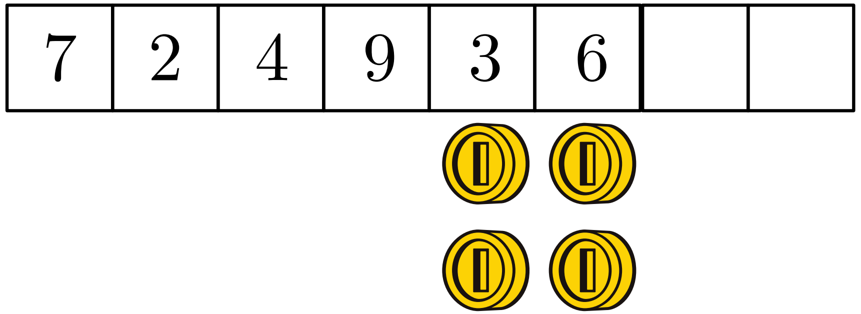
n coins to copy n elements to a new array

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins

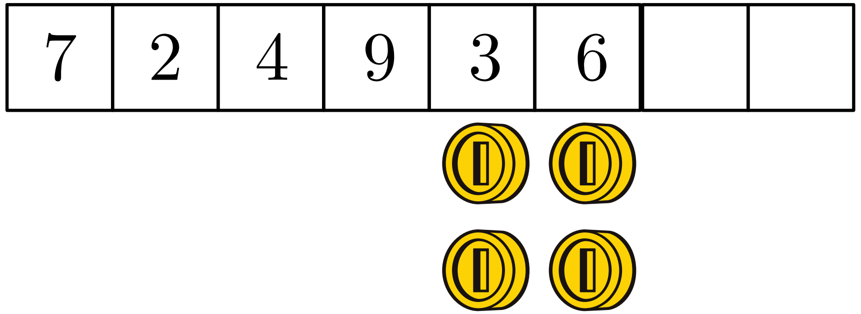


Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

How high do the amortized cost $\hat{c}_i$ of one append/insert need to be?

A: 1

B: 2

C: 3

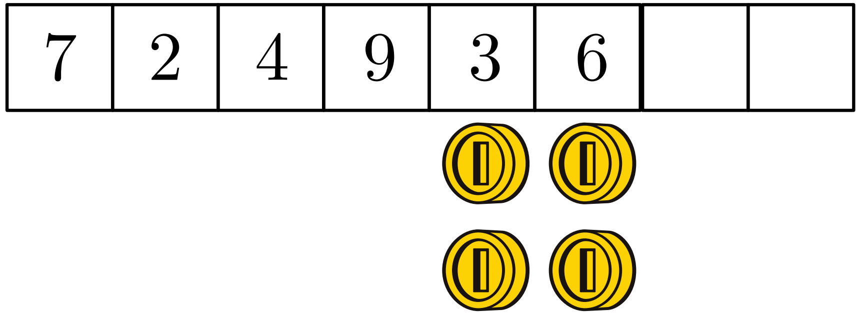
D: $n/2$

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

How high do the amortized cost $\hat{c}_i$ of one append/insert need to be?

A: 1

B: 2

C: 3

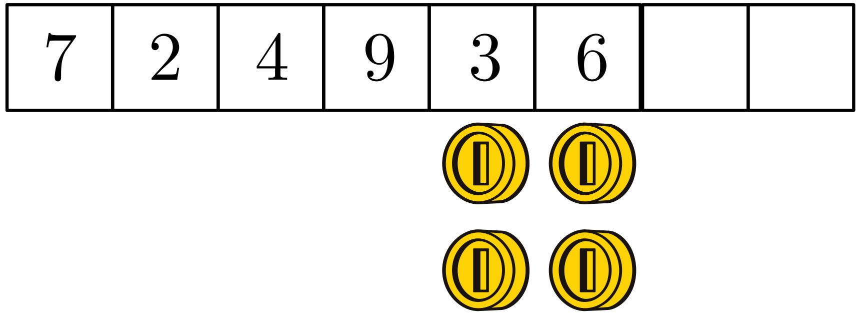
D: $n/2$

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

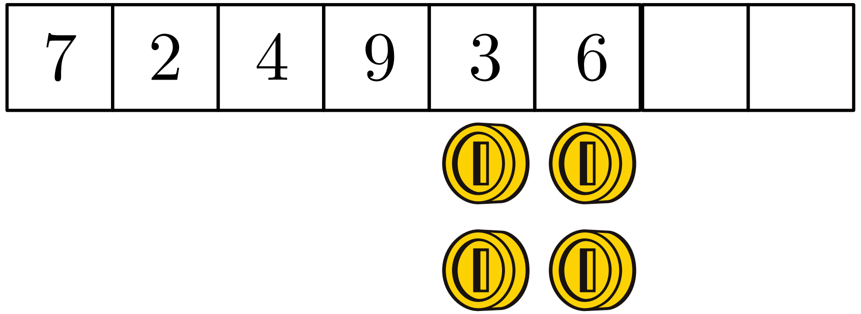
append: assign 3 coins ($\hat{c}_i$ = 3)

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

append: assign 3 coins ($\hat{c}_i$ = 3)

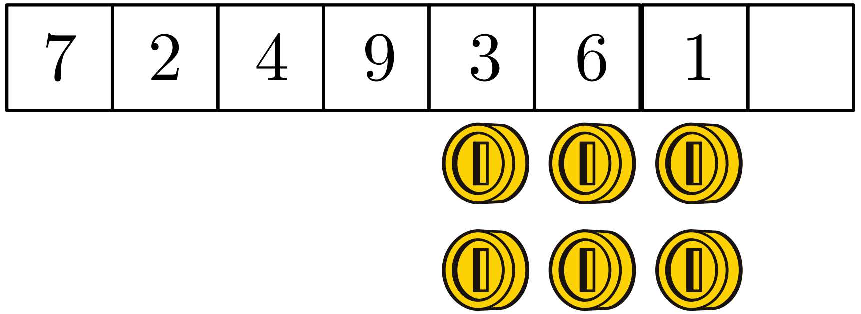
- 1 coin pays for the append. 2 coins are saved at the new element

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

append: assign 3 coins ($\hat{c}_i$ = 3)

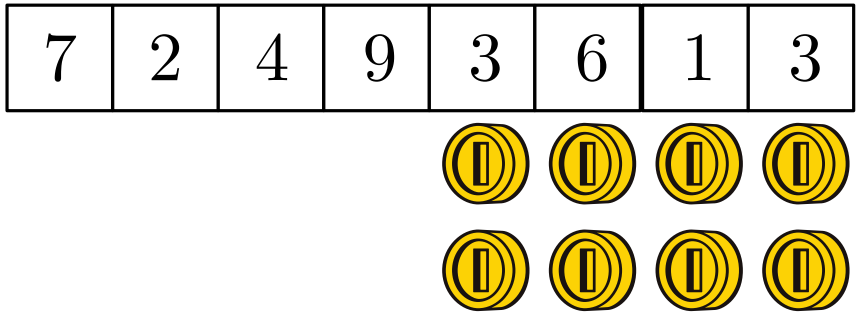
- 1 coin pays for the append. 2 coins are saved at the new element

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

append: assign 3 coins ($\hat{c}_i$ = 3)

- 1 coin pays for the append. 2 coins are saved at the new element

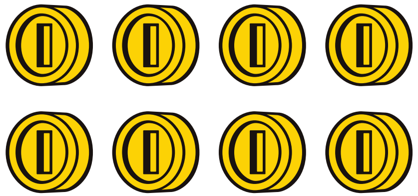
Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element

n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins

7	2	4	9	3	6	1	3



accounting/amortized costs($\hat{c}_i$):

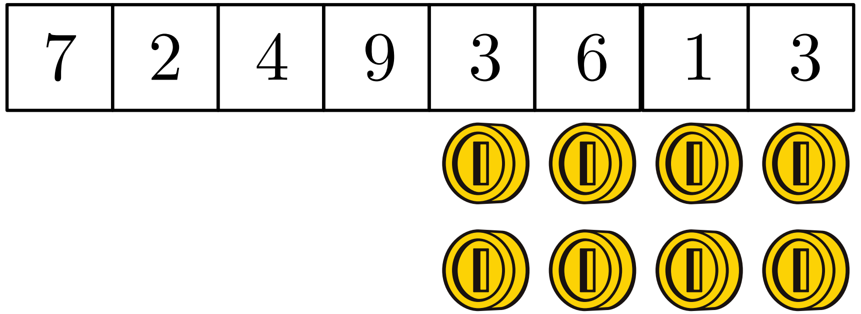
append: assign 3 coins ($\hat{c}_i$ = 3)

- 1 coin pays for the append. 2 coins are saved at the new element
- If full: the whole right half ($= n/2$ elements) has 2 coins per element
  $\rightarrow$ overall there are n coins which can pay for the copying

Accounting Method: Inserting into Dynamic Array

actual cost of operation: 1 coin to insert 1 element
n coins to copy n elements to a new array

invariant: every element in the right half of the array has 2 coins



accounting/amortized costs($\hat{c}_i$):

append: assign 3 coins ($\hat{c}_i$ = 3)

- 1 coin pays for the append. 2 coins are saved at the new element
- If full: the whole right half ($= n/2$ elements) has 2 coins per element
  $\rightarrow$ overall there are n coins which can pay for the copying

The worst-case running time $T(n)$ of a sequence of n append operations is $O(n)$. The amortized running time of one append is $O(\hat{c}_i) = O(1)$.

Overview

Amortized analysis:

- Consider sequence of n operations
- Determine the average worst-cost of operations by averaging over the whole sequence

Methods:

- Aggregate analysis
- Accounting method

Examples:

- Multi-pop stack
- Binary counter
- Dynamic arrays: insert, accounting method

Overview

Amortized analysis:

- Consider sequence of n operations
- Determine the average worst-cost of operations by averaging over the whole sequence

Methods:

- Aggregate analysis
- Accounting method

Examples:

- Multi-pop stack
- Binary counter
- Dynamic arrays: insert, accounting method

now: potential method

General Setup: Accounting and Potential method

Consider sequence of data structures D_i and operations Op_i

$$D_0 \overset{\text{Op}_1}{\rightsquigarrow} D_1 \overset{\text{Op}_2}{\rightsquigarrow} D_2 \overset{\text{Op}_3}{\rightsquigarrow} D_3 \overset{\text{Op}_4}{\rightsquigarrow} \cdots \overset{\text{Op}_m}{\rightsquigarrow} D_m$$

General Setup: Accounting and Potential method

Consider sequence of data structures D_i and operations Op_i

$$D_0 \overset{\mathsf{Op}_1}{\rightsquigarrow} D_1 \overset{\mathsf{Op}_2}{\rightsquigarrow} D_2 \overset{\mathsf{Op}_3}{\rightsquigarrow} D_3 \overset{\mathsf{Op}_4}{\rightsquigarrow} \cdots \overset{\mathsf{Op}_m}{\rightsquigarrow} D_m$$

Each operation has an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = coins

General Setup: Accounting and Potential method

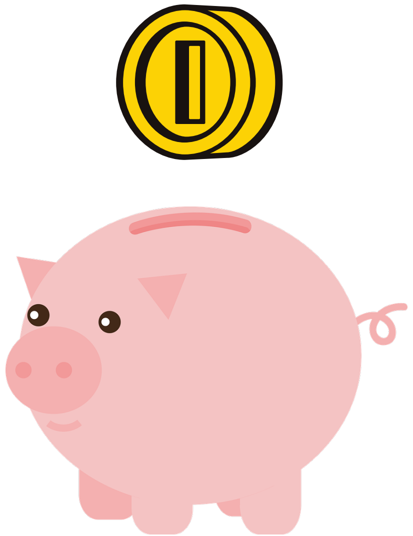
Consider sequence of data structures D_i and operations Op_i

$$D_0 \overset{\mathrm{Op}_1}{\rightsquigarrow} D_1 \overset{\mathrm{Op}_2}{\rightsquigarrow} D_2 \overset{\mathrm{Op}_3}{\rightsquigarrow} D_3 \overset{\mathrm{Op}_4}{\rightsquigarrow} \cdots \overset{\mathrm{Op}_m}{\rightsquigarrow} D_m$$

Each operation has an amortized cost (“coins”)

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = coins

- If $\hat{c}_i > c_i$: Save unused coins in D_i: $\hat{c}_i - c_i$
- If $\hat{c}_i < c_i$: Pay with saved coins



General Setup: Accounting and Potential method

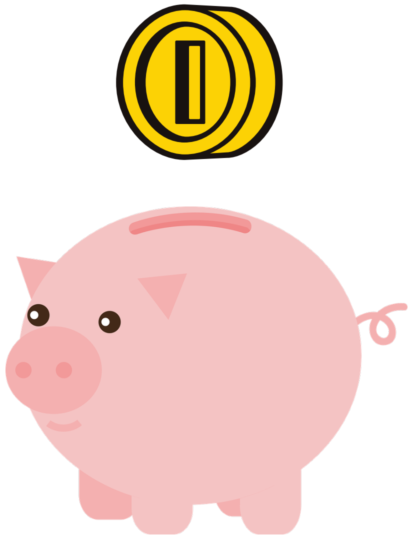
Consider sequence of data structures D_i and operations Op_i

$$D_0 \overset{\text{Op}_1}{\rightsquigarrow} D_1 \overset{\text{Op}_2}{\rightsquigarrow} D_2 \overset{\text{Op}_3}{\rightsquigarrow} D_3 \overset{\text{Op}_4}{\rightsquigarrow} \cdots \overset{\text{Op}_m}{\rightsquigarrow} D_m$$

Each operation has an amortized cost (“coins”)

- D_i = data structure after ith operation
- c_i = actual cost of ith operation
- $\hat{c}_i$ = amortized cost of ith operation = coins

- If $\hat{c}_i > c_i$: Save unused coins in D_i: $\hat{c}_i - c_i$
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 typically 0



General Setup: Accounting and Potential method

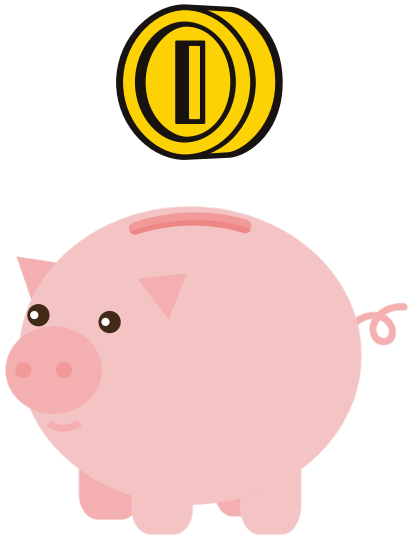
Consider sequence of data structures D_i and operations Op_i

$$D_0 \overset{\mathsf{Op}_1}{\rightsquigarrow} D_1 \overset{\mathsf{Op}_2}{\rightsquigarrow} D_2 \overset{\mathsf{Op}_3}{\rightsquigarrow} D_3 \overset{\mathsf{Op}_4}{\rightsquigarrow} \cdots \overset{\mathsf{Op}_m}{\rightsquigarrow} D_m$$

Each operation has an amortized cost ("coins")

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = coins

- If $\hat{c}_i > c_i$: Save unused coins in D_i: $\hat{c}_i - c_i$
- If $\hat{c}_i < c_i$: Pay with saved coins
- initial credit in D_0 typically 0



cost invariant: saved coins ≥ 0

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$\hat{c}_i = c_i$ + coins saved in step i

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$\hat{c}_i = c_i$ + coins saved in step i

$= c_i$ + change in potential in step i

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\quad \sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$\hat{c}_i = c_i$ + coins saved in step i

$= c_i$ + change in potential in step i

$= c_i + \Phi(D_i) - \Phi(D_{i-1})$

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$$\begin{aligned} \hat{c}_i &= c_i + \text{coins saved in step } i \\ &= c_i + \text{change in potential in step } i \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \end{aligned}$$

first step in potential method:

define potential function $\Phi: D \to \mathbb{R}_0^+$

Accounting Method vs Potential Method

accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\quad \sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$

(coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\quad \Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$\hat{c}_i = c_i$ + coins saved in step i

$= c_i$ + change in potential in step i

$= c_i + \Phi(D_i) - \Phi(D_{i-1})$

first step in potential method:

define potential function $\Phi :\ D \to \mathbb{R}_0^+$

with $\Phi(D_i) \geq \Phi(D_0)$ for all i $\quad$ (and typically $\Phi(D_0) = 0$)

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

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

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i \quad + \quad \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1})\end{aligned}$$

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

$$
\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \Phi(D_1) - \Phi(D_0) + \\
&\qquad\qquad\quad \Phi(D_2) - \Phi(D_1) + \\
&\qquad\qquad\quad \Phi(D_3) - \Phi(D_2) + \\
&\qquad\qquad\qquad \dots \qquad + \\
&\qquad\qquad\quad \Phi(D_n) - \Phi(D_{n-1})
\end{aligned}
$$

(telescopic sum)

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

$$
\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \Phi(D_1) - \Phi(D_0) + \\
&\qquad\qquad\qquad \Phi(D_2) - \Phi(D_1) + \\
&\qquad\qquad\qquad \Phi(D_3) - \Phi(D_2) + \\
&\qquad\qquad\qquad \quad \ldots \quad + \\
&\qquad\qquad\qquad \Phi(D_n) - \Phi(D_{n-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \Phi(D_n) - \Phi(D_0)
\end{aligned}
$$

(telescopic sum)

Why $\Phi(D_i) \geq \Phi(D_0)$?

We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

$$
\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \Phi(D_1) - \Phi(D_0) + \\
&\qquad\qquad\qquad \Phi(D_2) - \Phi(D_1) + \\
&\qquad\qquad\qquad \Phi(D_3) - \Phi(D_2) + \\
&\qquad\qquad\qquad \qquad \dots \qquad + \\
&\qquad\qquad\qquad \Phi(D_n) - \Phi(D_{n-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}
\end{aligned}
$$

(telescopic sum)

Why $\Phi(D_i) \geq \Phi(D_0)$?

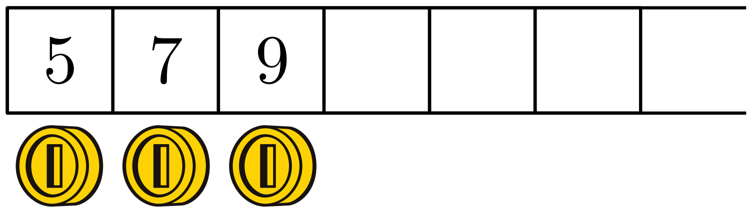
We want to show $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ (actual costs $\leq$ amortized costs)

$$
\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \Phi(D_1) - \Phi(D_0) + \\
&\qquad\qquad\quad \Phi(D_2) - \Phi(D_1) + \\
&\qquad\qquad\quad \Phi(D_3) - \Phi(D_2) + \\
&\qquad\qquad\qquad \ldots \qquad + \\
&\qquad\qquad\quad \Phi(D_n) - \Phi(D_{n-1}) \\
&= \sum_{i=1}^{n} c_i \quad + \quad \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0} \\
&\geq \sum_{i=1}^{n} c_i
\end{aligned}
$$

(telescopic sum)

Potential Method: Multi-Pop Stack

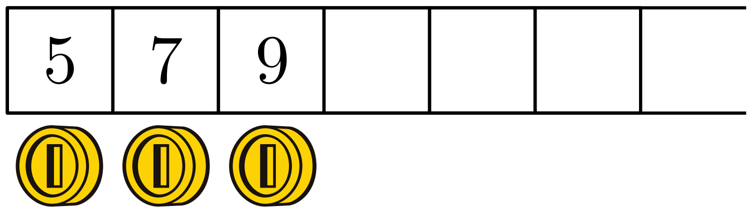
Question: How many coins should we save with a multi-pop stack?



Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i

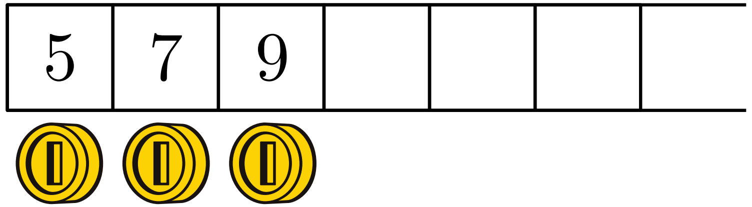


Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i

$\Phi(D_i) \geq 0 = \Phi(D_0)$

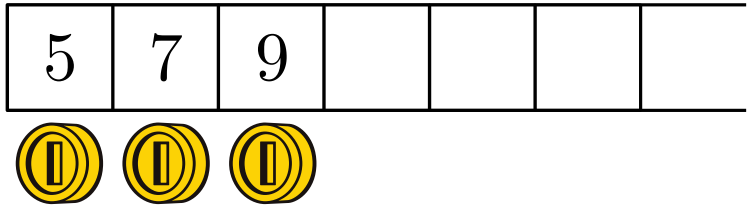


Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i

$\Phi(D_i) \geq 0 = \Phi(D_0)$

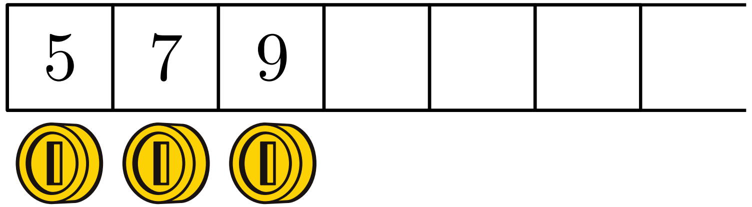


amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

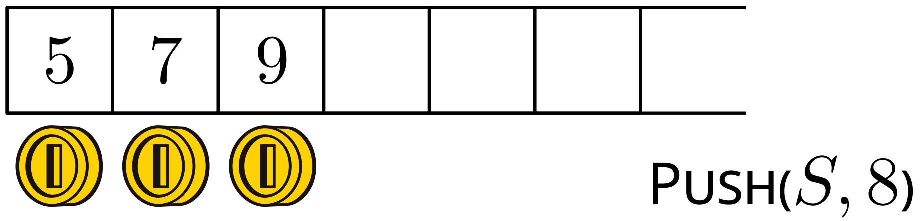
PUSH(S, x):

- $c_i = 1$ (constant-time operation)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

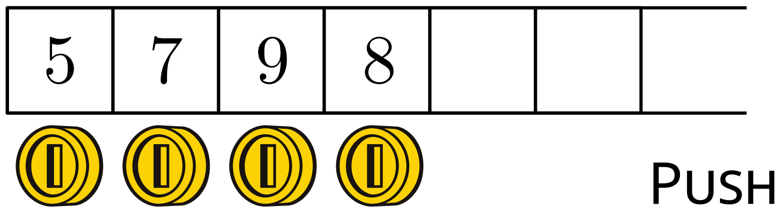
PUSH(S, x):

- $c_i = 1$ (constant-time operation)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



PUSH($S, 8$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

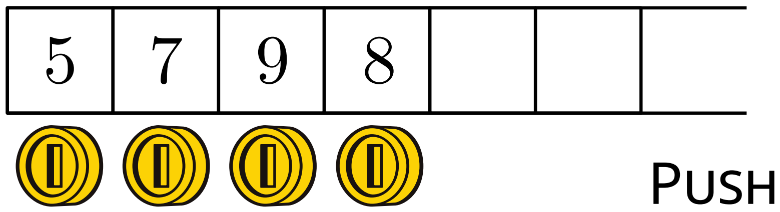
PUSH(S, x):

- $c_i = 1$ (constant-time operation)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



PUSH($S, 8$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

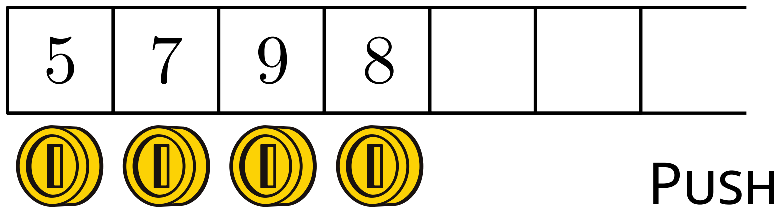
PUSH(S, x):

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



PUSH($S, 8$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

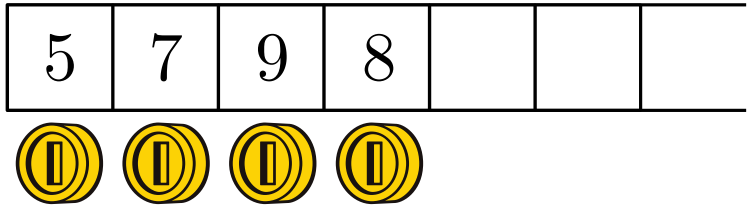
PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Push(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

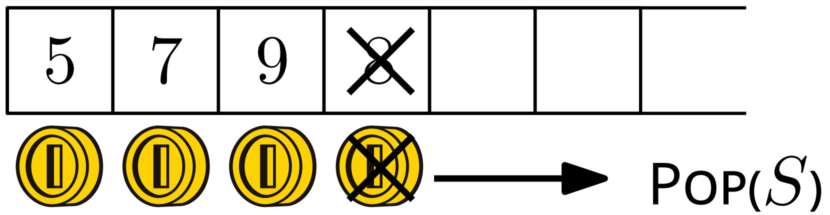
Pop(S):

- $c_i = 1$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Push(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

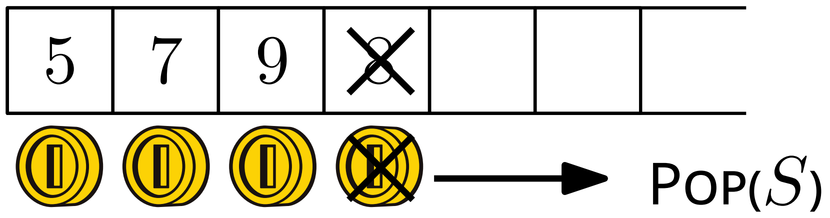
Pop(S):

- $c_i = 1$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

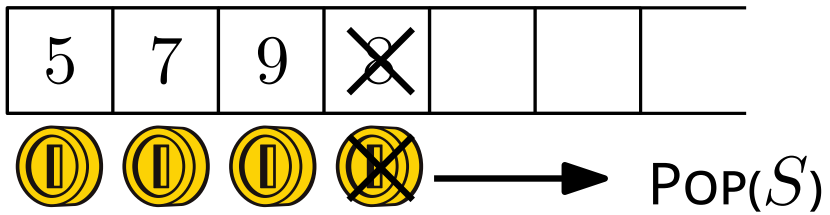
- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

POP(S):

- $c_i = 1$
- change in potential: -1 (one element removed)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?



$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

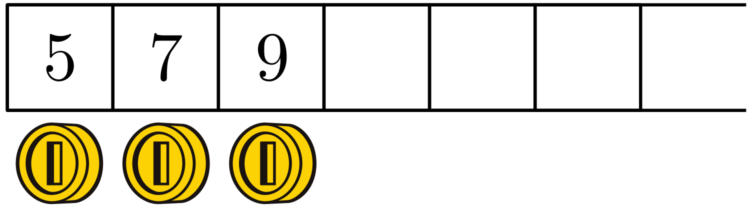
- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

POP(S): $\hat{c}_i = 1 - 1 = 0$

- $c_i = 1$
- change in potential: -1 (one element removed)

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?



$\Phi(D_i) :=$ number of elements stored in D_i

$\Phi(D_i) \geq 0 = \Phi(D_0)$

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Push(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

Pop(S): $\hat{c}_i = 1 - 1 = 0$

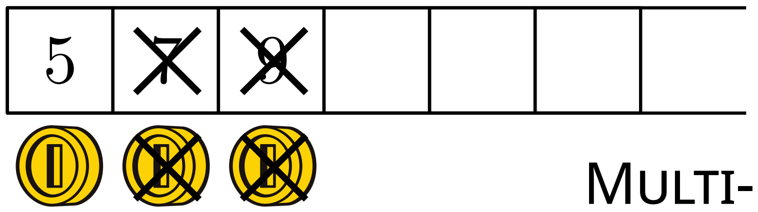
- $c_i = 1$
- change in potential: -1 (one element removed)

Multi-Pop(S, k):

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



MULTI-POP($S, 2$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

POP(S): $\hat{c}_i = 1 - 1 = 0$

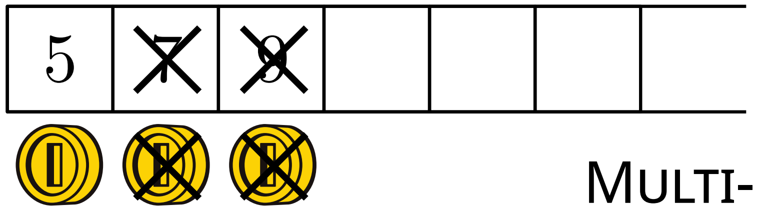
- $c_i = 1$
- change in potential: -1 (one element removed)

MULTI-POP(S, k):

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



Multi-Pop($S, 2$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Push(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

Pop(S): $\hat{c}_i = 1 - 1 = 0$

- $c_i = 1$
- change in potential: -1 (one element removed)

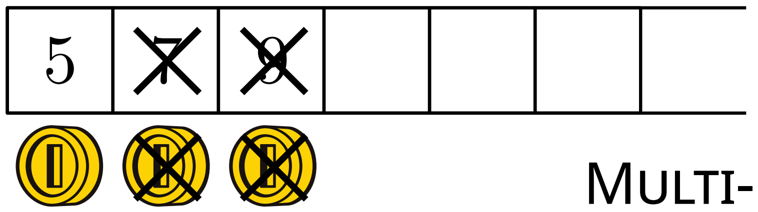
Multi-Pop(S, k):

- $c_i =$ number of elements removed

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



MULTI-POP($S, 2$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

POP(S): $\hat{c}_i = 1 - 1 = 0$

- $c_i = 1$
- change in potential: -1 (one element removed)

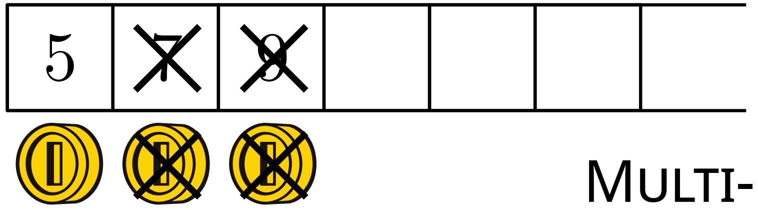
MULTI-POP(S, k):

- $c_i =$ number of elements removed
- change in potential $= -$ number of elements removed

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



MULTI-POP($S, 2$)

amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 1 + 1 = 2$

- $c_i = 1$ (constant-time operation)
- change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (one new element)

POP(S): $\hat{c}_i = 1 - 1 = 0$

- $c_i = 1$
- change in potential: -1 (one element removed)

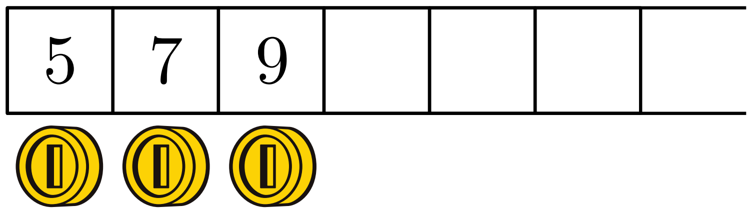
MULTI-POP(S, k): $\hat{c}_i = 0$

- $c_i =$ number of elements removed
- change in potential $= -$ number of elements removed

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



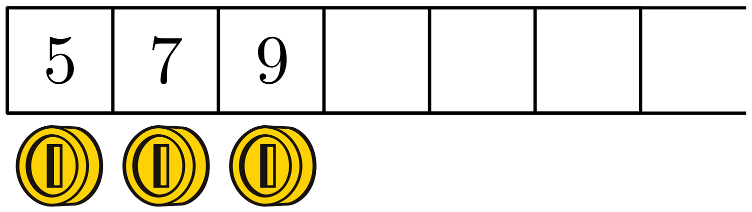
amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 2$
POP(S): $\hat{c}_i = 0$
MULTI-POP(S, k): $\hat{c}_i = 0$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 2$
POP(S): $\hat{c}_i = 0$
MULTI-POP(S, k): $\hat{c}_i = 0$

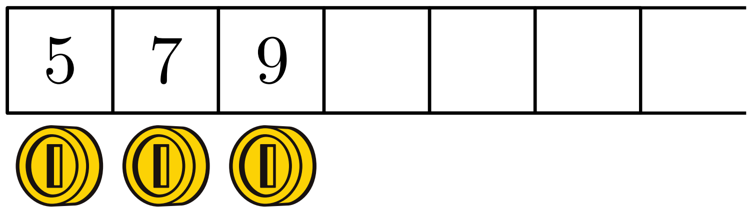
Running time of a sequence of n PUSH, POP, MULTI-POP operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i

$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

Push(S, x): $\hat{c}_i = 2$

Pop(S): $\hat{c}_i = 0$

Multi-Pop(S, k): $\hat{c}_i = 0$

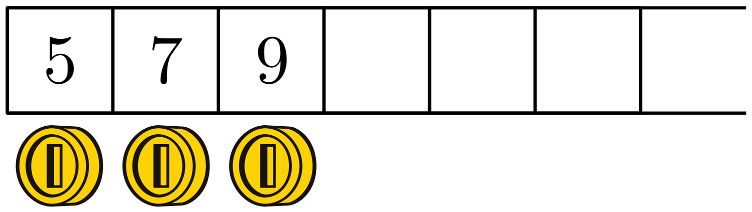
Running time of a sequence of n Push, Pop, Multi-Pop operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

- amortized cost per operation $\hat{c}_i \leq 2$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 2$
POP(S): $\hat{c}_i = 0$
MULTI-POP(S, k): $\hat{c}_i = 0$

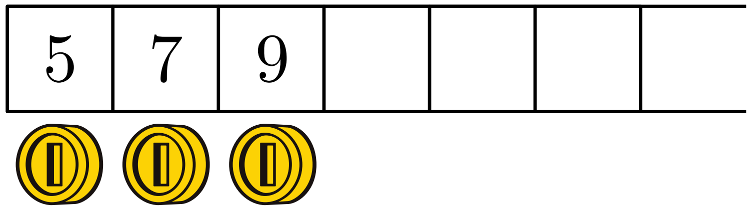
Running time of a sequence of n PUSH, POP, MULTI-POP operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

- amortized cost per operation $\hat{c}_i \leq 2$
- actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$

Potential Method: Multi-Pop Stack

Question: How many coins should we save with a multi-pop stack?

$\Phi(D_i) :=$ number of elements stored in D_i
$\Phi(D_i) \geq 0 = \Phi(D_0)$



amortized costs ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

PUSH(S, x): $\hat{c}_i = 2$
POP(S): $\hat{c}_i = 0$
MULTI-POP(S, k): $\hat{c}_i = 0$

Running time of a sequence of n PUSH, POP, MULTI-POP operations (starting from an empty stack) is in $O(n)$. The amortized cost per operation is $O(\hat{c}_i) = O(1)$

- amortized cost per operation $\hat{c}_i \leq 2$
- actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$, because $\Phi(D_n) \geq \Phi(D_0) = 0$

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

example: $k = 6$

flipped bits

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1

example: $k = 6$

flipped bits

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

costs. number of bits flipped per operation ($= O(k)$)

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

costs. number of bits flipped per operation ($= O(k)$)

running time. worst-case running time of sequence of n increments is $O(kn)$.

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

costs. number of bits flipped per operation ($= O(k)$)

running time. worst-case running time of sequence of n increments is $O(kn)$.

too pessimistic!

Want to show: $O(n)$

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

What makes an increment expensive?

Binary Counter (reminder)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

What makes an increment expensive?

What happens during such an increment?

Potential Method: Binary Counter

$\Phi(D_i) =$???

Potential Method: Binary Counter

$\Phi(D_i) =$???

$$
\begin{array}{r c}
 & \begin{array}{cccccc} 5 & 4 & 3 & 2 & 1 & 0 \end{array} \\
23: & \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array}
\end{array}
$$

Potential Method: Binary Counter

$\Phi(D_i) =$???

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

What is a good choice for $\Phi(D_{23})$?

Potential Method: Binary Counter

$\Phi(D_i) =$???

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

What is a good choice for $\Phi(D_{23})$?

A: 1

B: 3

C: 4

D: 5

Potential Method: Binary Counter

$\Phi(D_i) =$???

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

What is a good choice for $\Phi(D_{23})$?

A: 1

B: 3

C: 4

D: 5

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

What is a good choice for $\Phi(D_{23})$?

A: 1

B: 3

C: 4

D: 5

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

	5	4	3	2	1	0
$23:$	0	1	0	1	1	1

What is a good choice for $\Phi(D_{23})$?

A: 1

B: 3

C: 4

D: 5

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

- c_i = number of bits flipped

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$): $0\ 1\ \boxed{0\ 1\ 1\ 1}$

- c_i = number of bits flipped

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	0	1	1	1

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	1	1	1

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	1	1	1

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s

  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0

  = 1 $-$ number of bits flipped from 1 to 0

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s

  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0

  = 1 $-$ number of bits flipped from 1 to 0

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$=$ ($1 +$ number of bits flipped from 1 to 0) $+$

($1 -$ number of bits flipped from 1 to 0)

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= ($ $1 +$ number of bits flipped from 1 to 0$) +$

$($ $1 -$ number of bits flipped from 1 to 0 $)$

$= 2$

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

- change in potential = increase in number of 1s

  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0

  = 1 $-$ number of bits flipped from 1 to 0

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \quad = 2$$

The worst-case running time $T(n)$ of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

Potential Method: Binary Counter

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$, where $A[j]$ is the jth least-significant bit

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0
- change in potential = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \quad = 2$$

The worst-case running time $T(n)$ of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

- $\hat{c}_i \leq 2 = O(1)$, and actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = O(n)$

Summary

Amortized analysis:

- Consider sequence of n operations
- Take the average of the worst-case running time of the operations over the sequence

Methods:

- Aggregate analysis
- Accounting method
- Potential method

Examples:

- Multi-pop stack
- Binary counter
- Dynamic arrays: insert, accounting method