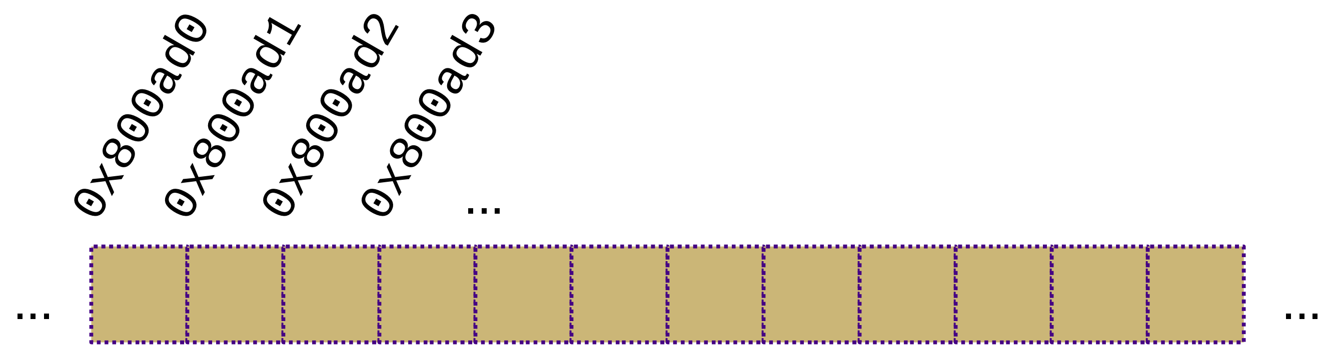
CS240
Fall 2014

Mike Lam, Professor

Dynamic Arrays

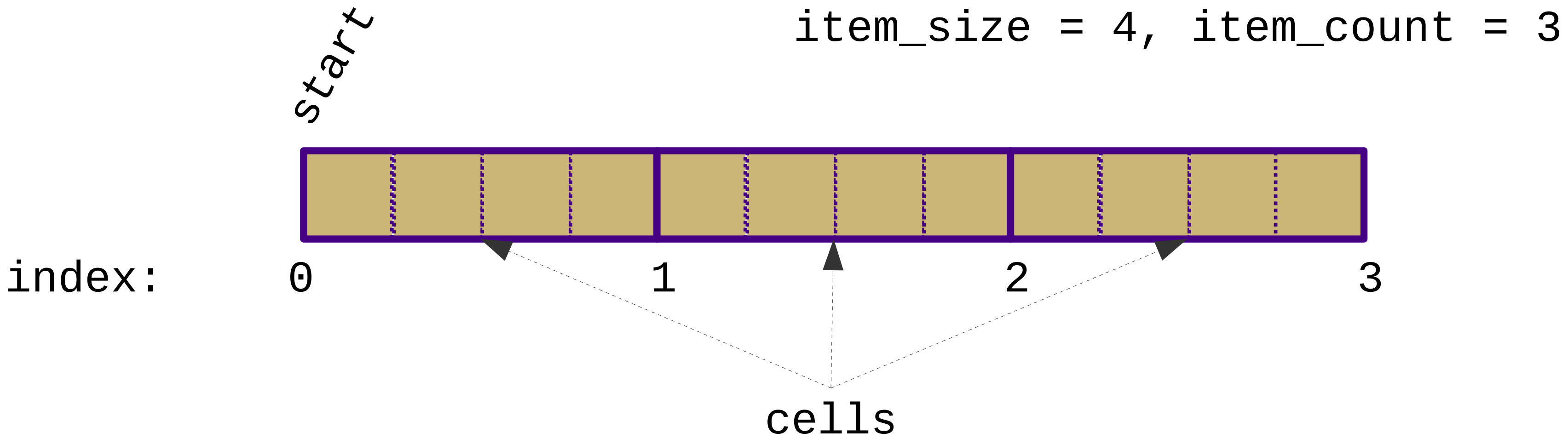
Computer Memory

- Lowest level: sequence of bytes
- Each byte has a 32-bit or 64-bit address
- Every byte is equally easy to access
  - "Random access" memory



Arrays

- Finite sequence of uniformly-sized segments
  - Starting address, item size (in bytes), item count (fixed)
- Each location is a **cell** located at a zero-based **index** offset from the start
  - Address of cell `i` is `start+(i*item_size)`



Compact Arrays

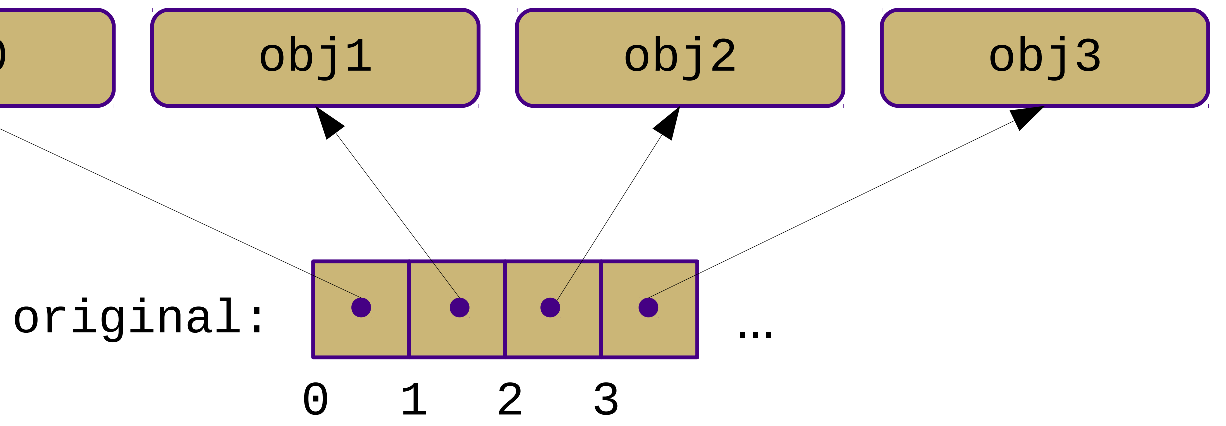
- In some languages, compact byte arrays are part of the language
  - Stack (C/C++)
    - `int my_array[n]`
  - Heap (C/C++)
    - `my_array = (int*)malloc(n*sizeof(int))`
  - Heap (C++/Java)
    - `my_array = new int[n]`

Compact Arrays

- In Python, compact byte arrays of built-in types are supported by the array module
  - `from array import array`
  - `my_array = array('i', [0]*n)`
- However, Python lists are not compact arrays!
  - Referential
  - Not fixed-length

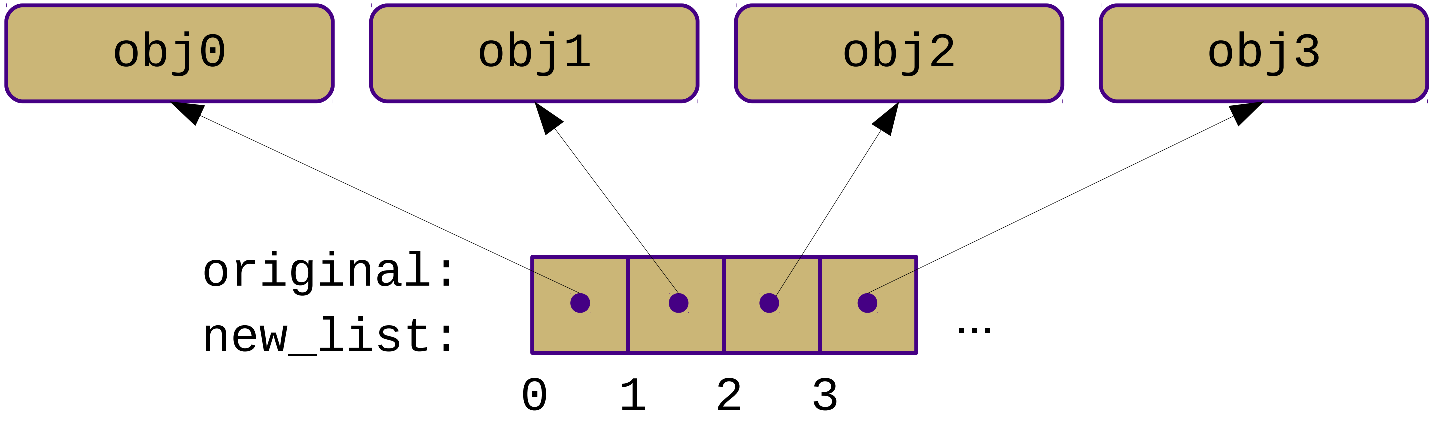
Referential Array

- Array of references
- Each cell contains a 32 or 64 bit pointer to actual objects
- This is how Python implements lists



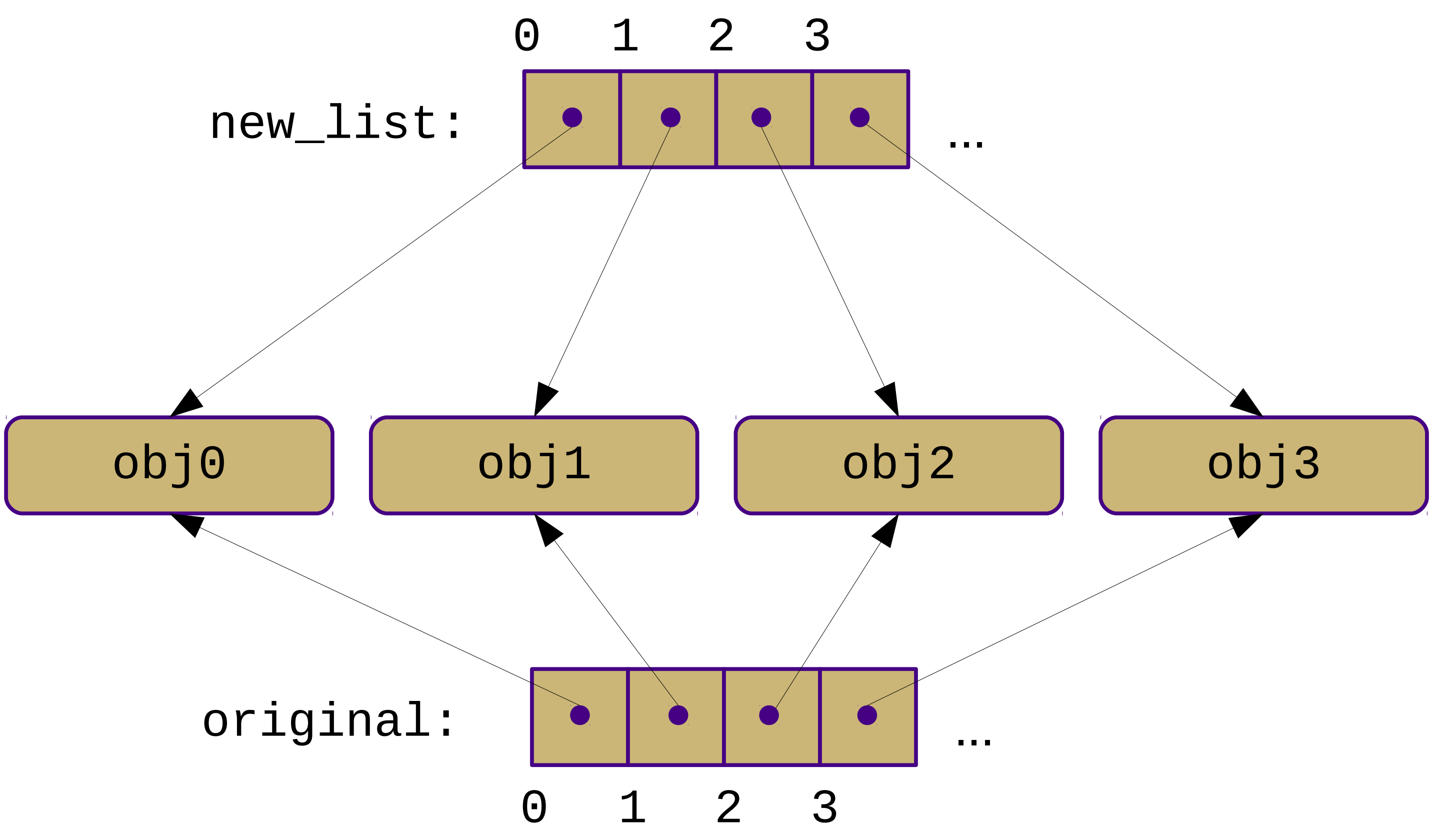
Shallow vs. Deep Copy

- Alias: `new_list = original`



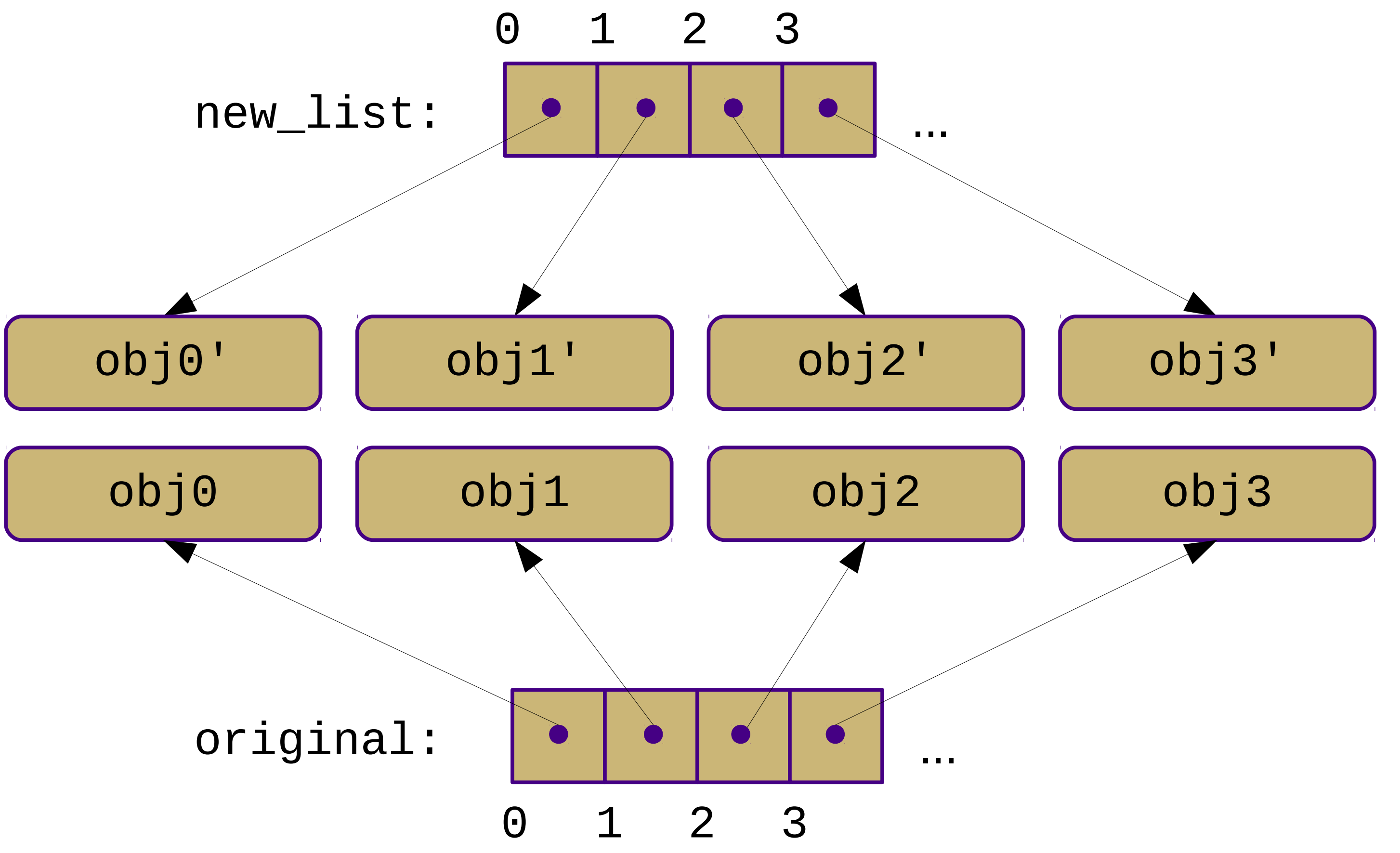
Shallow vs. Deep Copy

- Shallow copy: `new_list = list(original)`



Shallow vs. Deep Copy

- Deep copy: `new_list = copy.deepcopy(original)`



Python Lists

- How does the append operation work in Python?
  - Standard arrays are fixed-length
  - Python uses "dynamic arrays"

Dynamic Arrays

- Goal: Add items to an array
- Issue: Arrays are fixed-length

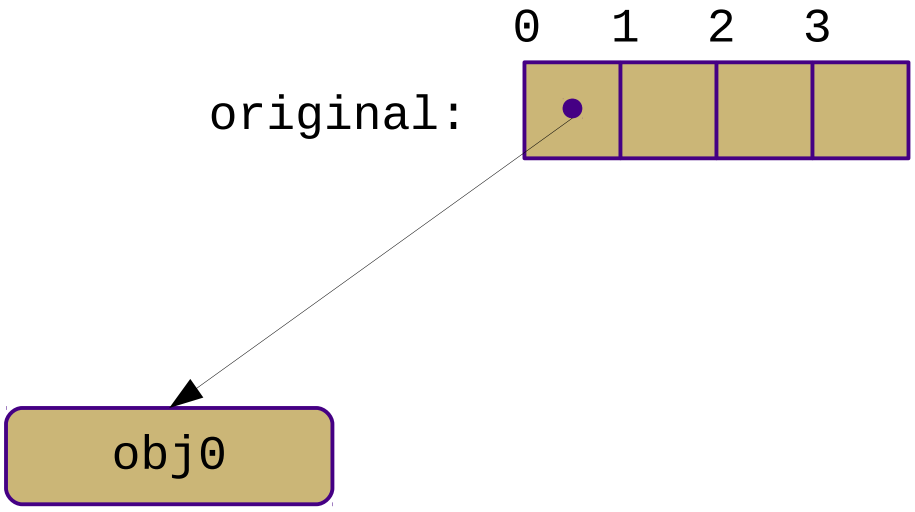
Dynamic Arrays

- Goal: Add items to an array
- Issue: Arrays are fixed-length
- Naive solution: Resize the array
  - Problem: no guarantee that we can do this!

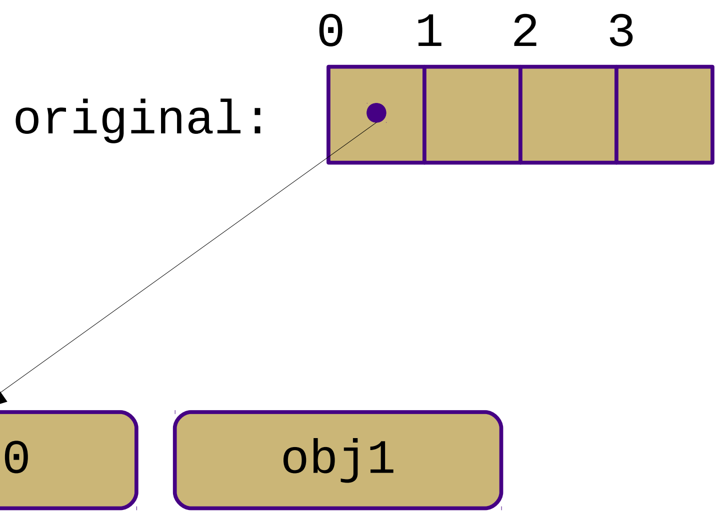
Dynamic Arrays

- Goal: Add items to an array
- Issue: Arrays are fixed-length
- Naive solution: Resize the array
  - Problem: no guarantee that we can do this!
- More robust solution: Dynamic arrays
  - Allocate more space than currently needed
  - Re-allocate and copy when the original size is exceeded

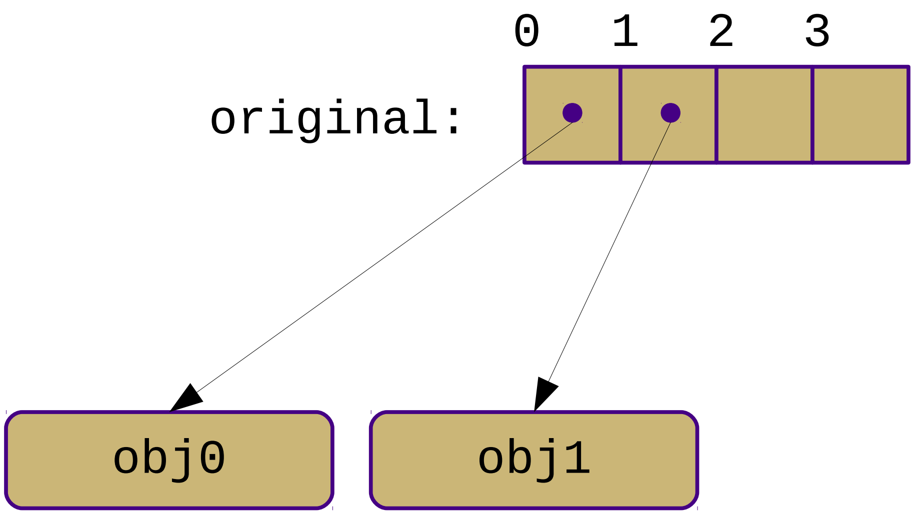
Dynamic Arrays



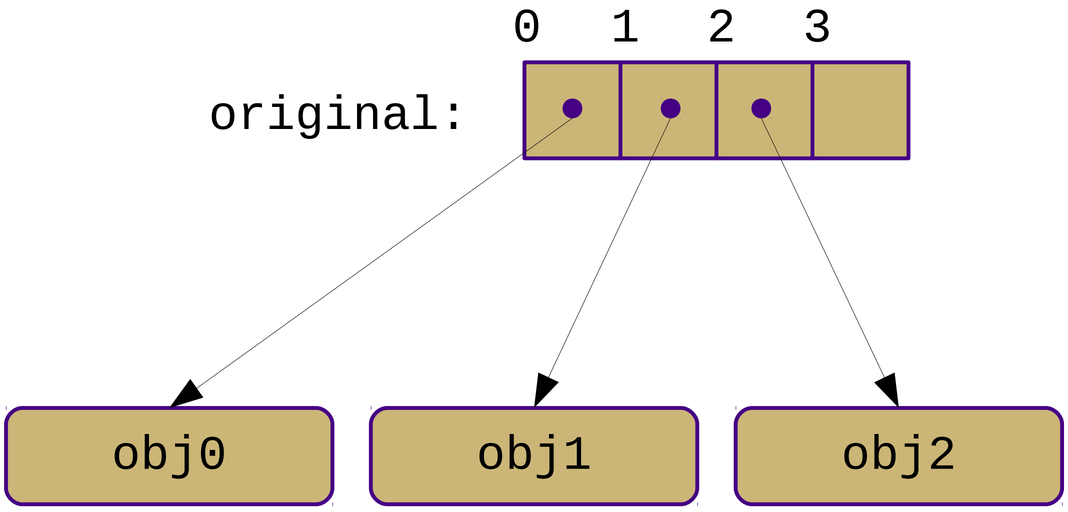
Dynamic Arrays



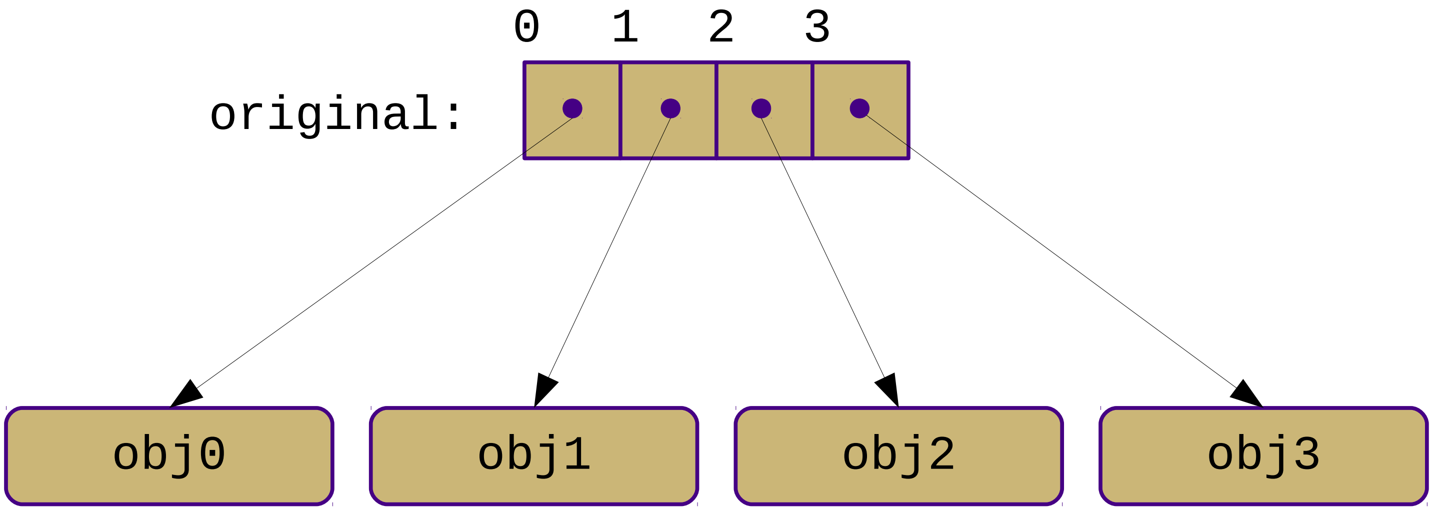
Dynamic Arrays



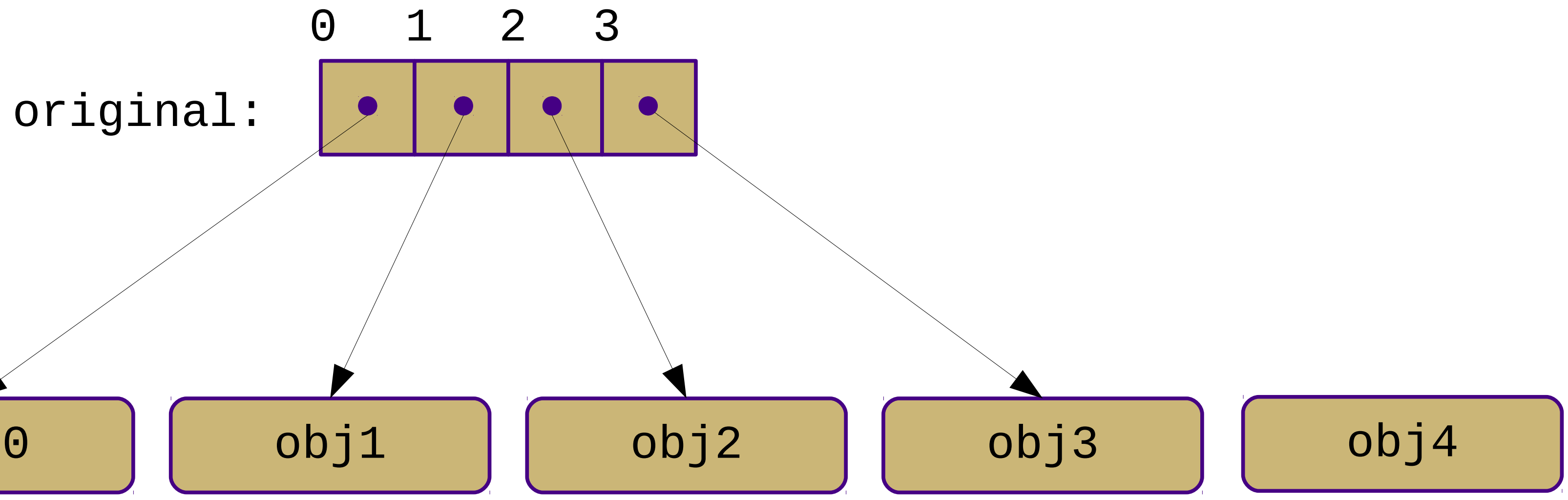
Dynamic Arrays



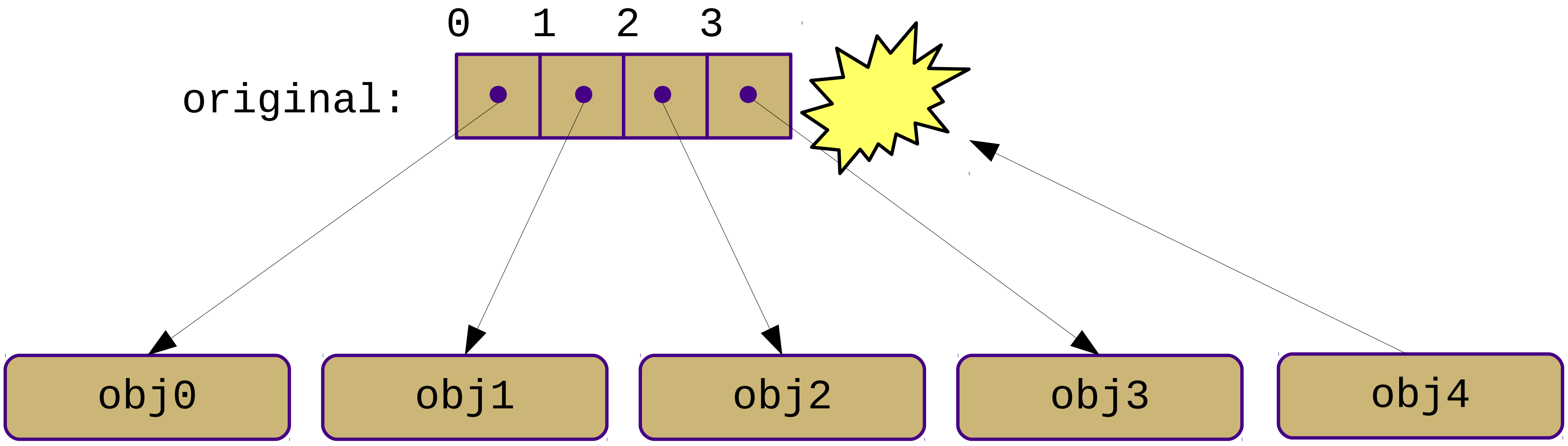
Dynamic Arrays



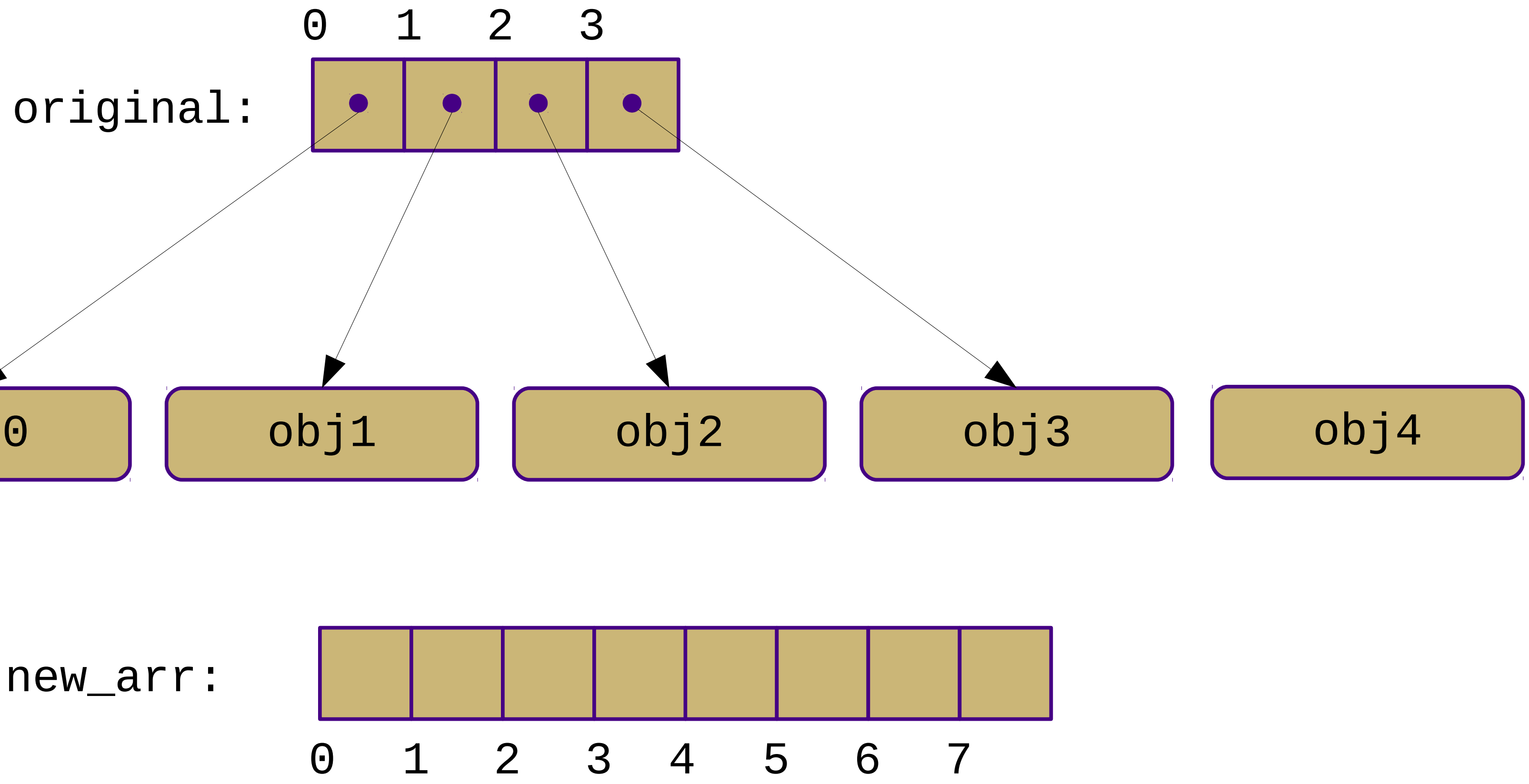
Dynamic Arrays



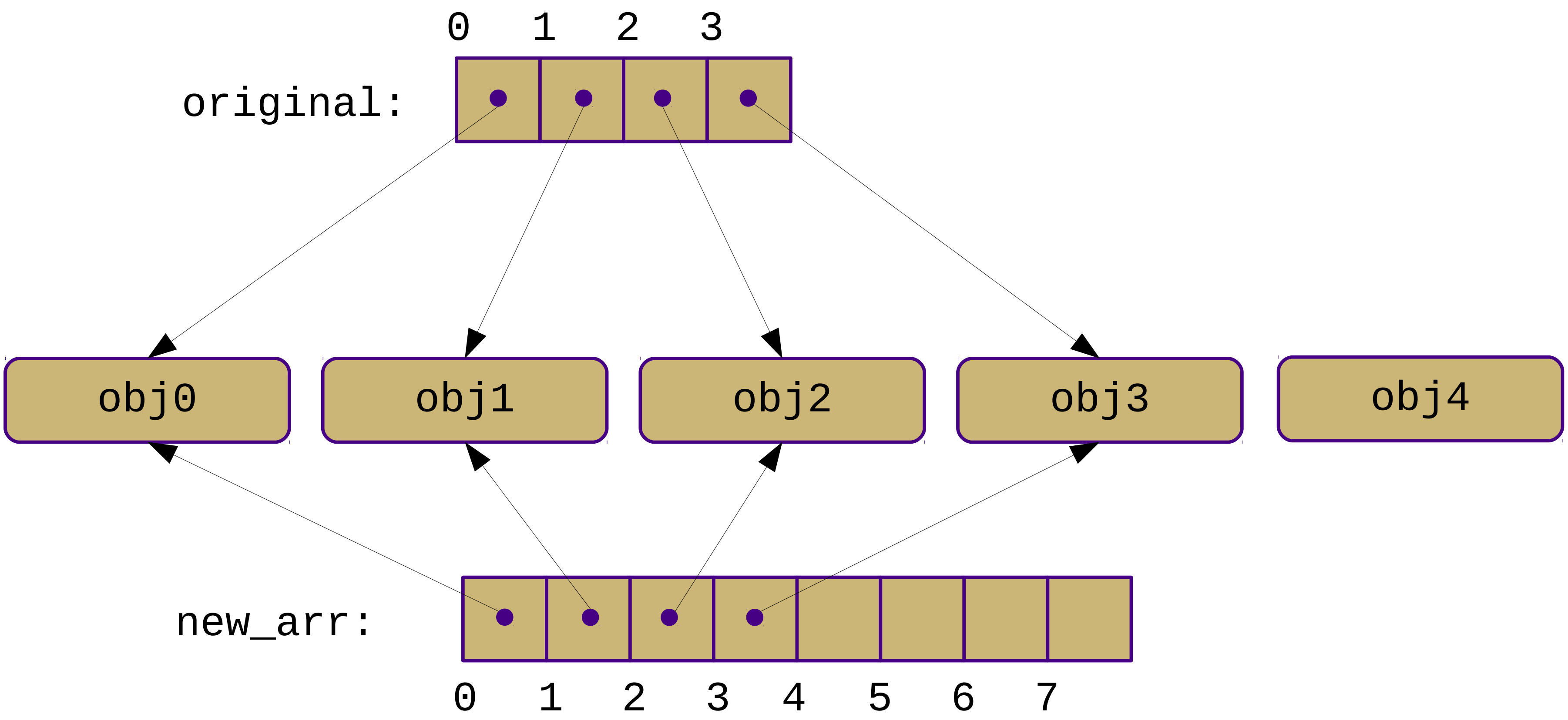
Dynamic Arrays



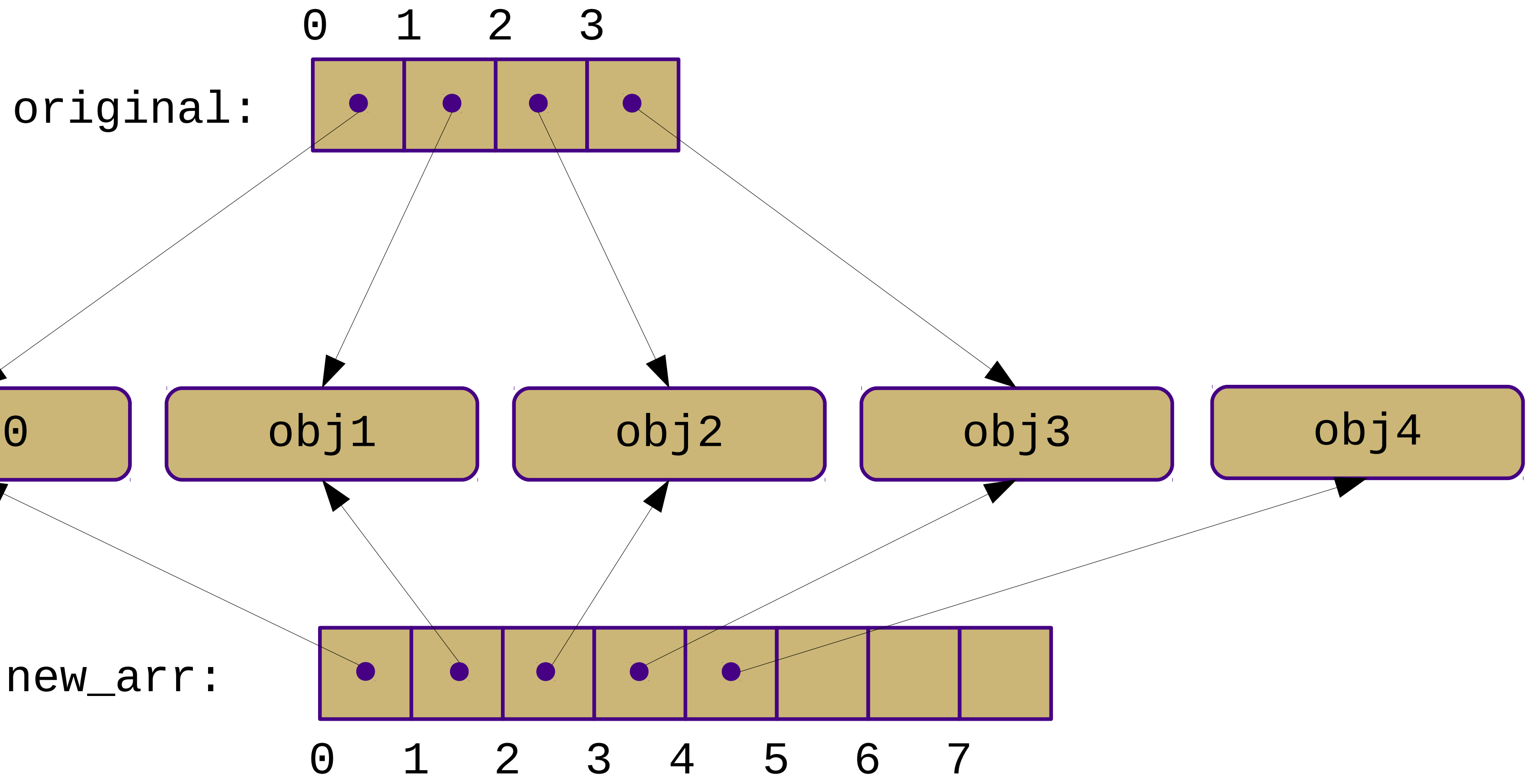
Dynamic Arrays



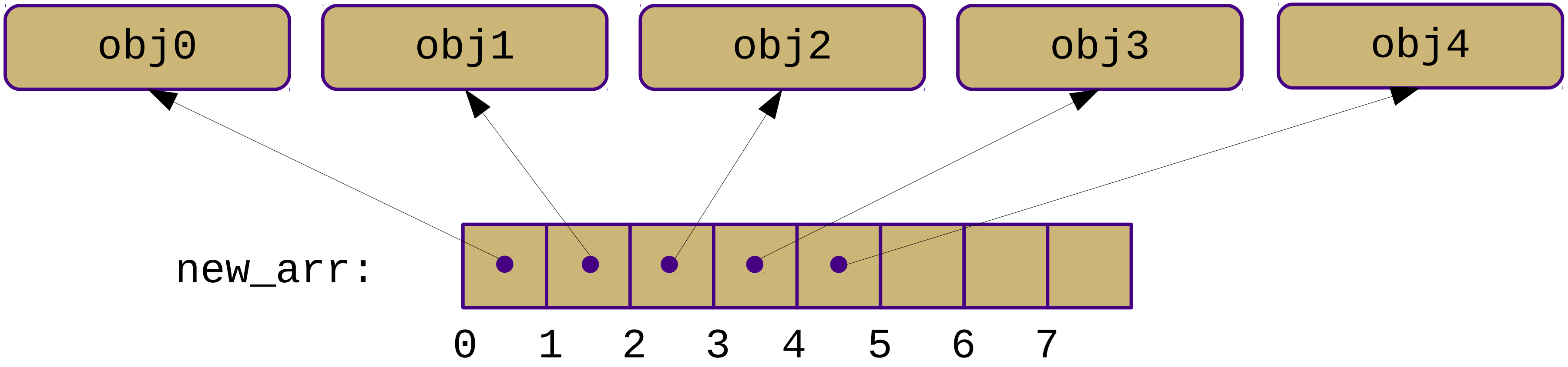
Dynamic Arrays



Dynamic Arrays



Dynamic Arrays



Dynamic Arrays

- State information:
  - **n**: current element count
  - **cap**: current maximum element count
  - **arr**: array reference
- Invariant: **cap** >= **n**

Dynamic Arrays

- How big should we initialize new arrays?
  - For now let's make it big enough for a single element
- How much extra space should we allocate when we need to resize it?
  - For now, let's assume we double the size

Dynamic Arrays

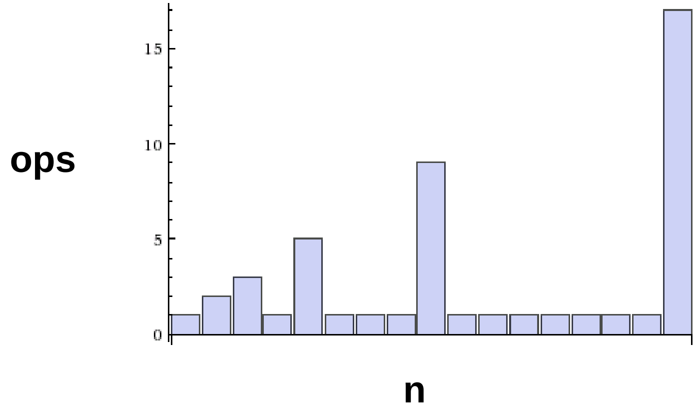
- See code example
  - (simpler than book example)
  - (uses built-in lists instead of ctypes)

Dynamic Arrays

- Big-O analysis
  - Create empty array: O(1)
  - Access element: O(1)
  - Modify element: O(1)
  - Get length: O(1)
  - Append element: ???
    - Let's measure cost in "copy operations"

Dynamic Arrays

- Big-O analysis
  - Create empty array: O(1)
  - Access element: O(1)
  - Modify element: O(1)
  - Get length: O(1)
  - Append element:
    - If **cap** > **n**: O(1)
    - If **cap** == **n**: O(n)



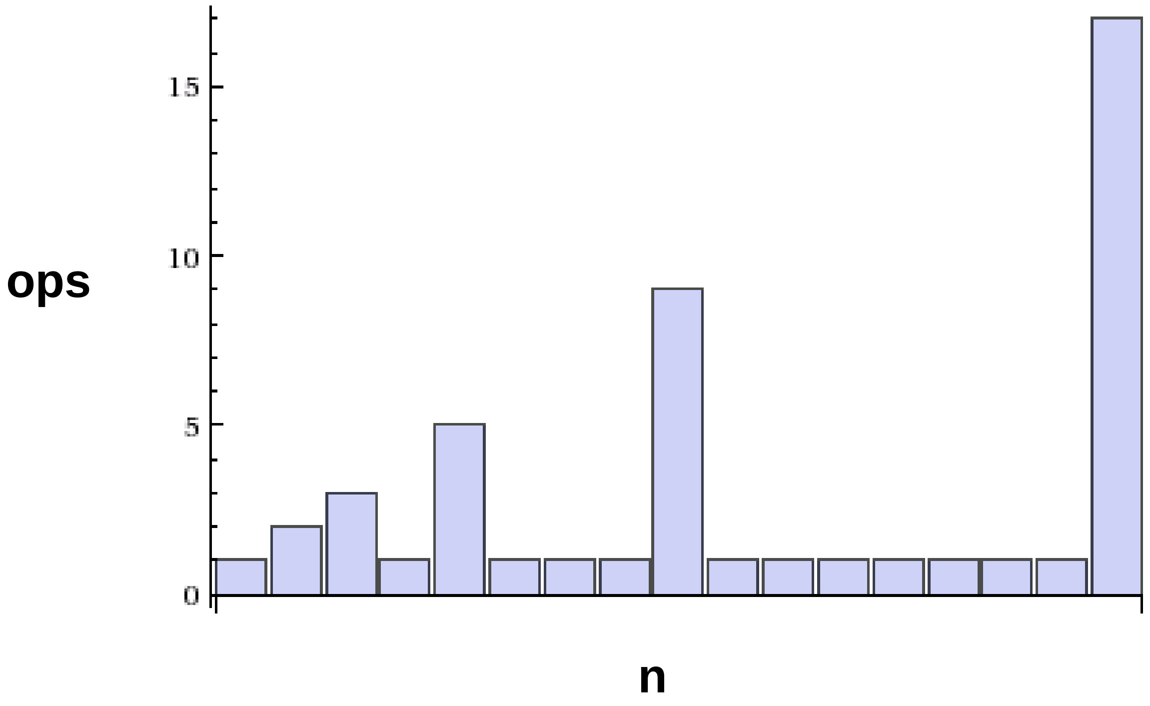
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
cap	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16	32
ops	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17

Dynamic Arrays

- Can we argue that the *average* cost of the append operation is O(1), despite its occasional O(n) cost?
- Yes! Use *amortized* analysis
- Basic idea: charge algorithm $$ to perform operations
  - Overcharge for some (inexpensive) operations
  - Use saved $$ to pay for expensive operations
  - Show that the total $$ spent is O(n) for *n* operations
  - Thus, each operation can be considered O(1)

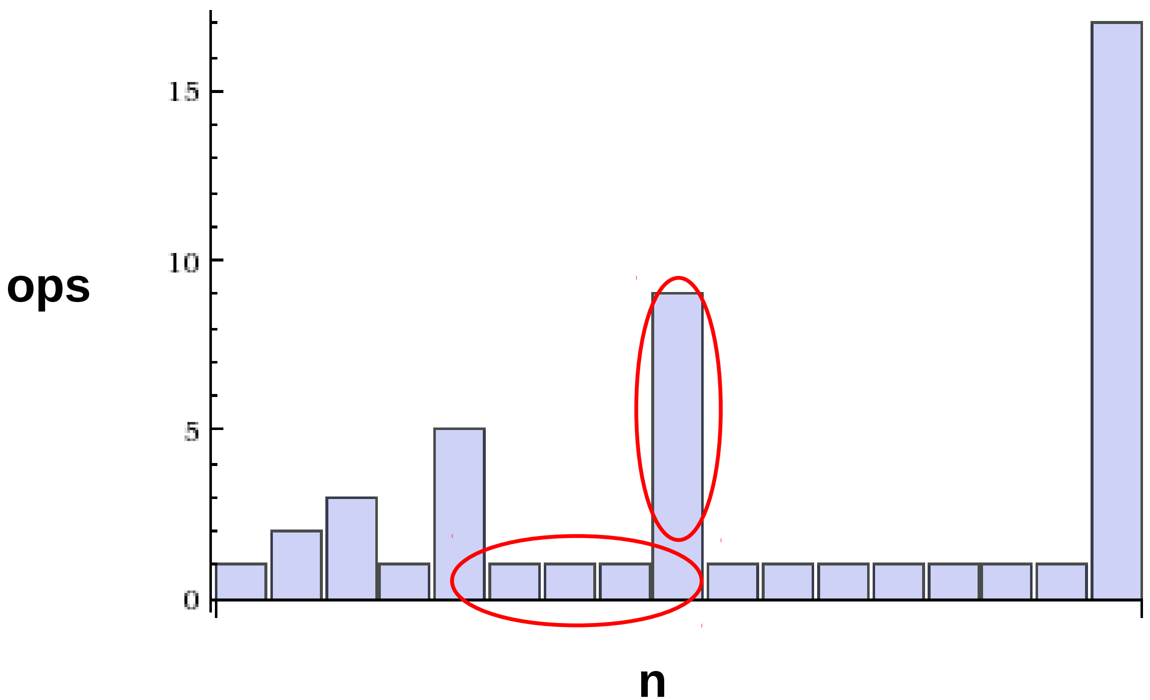
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency



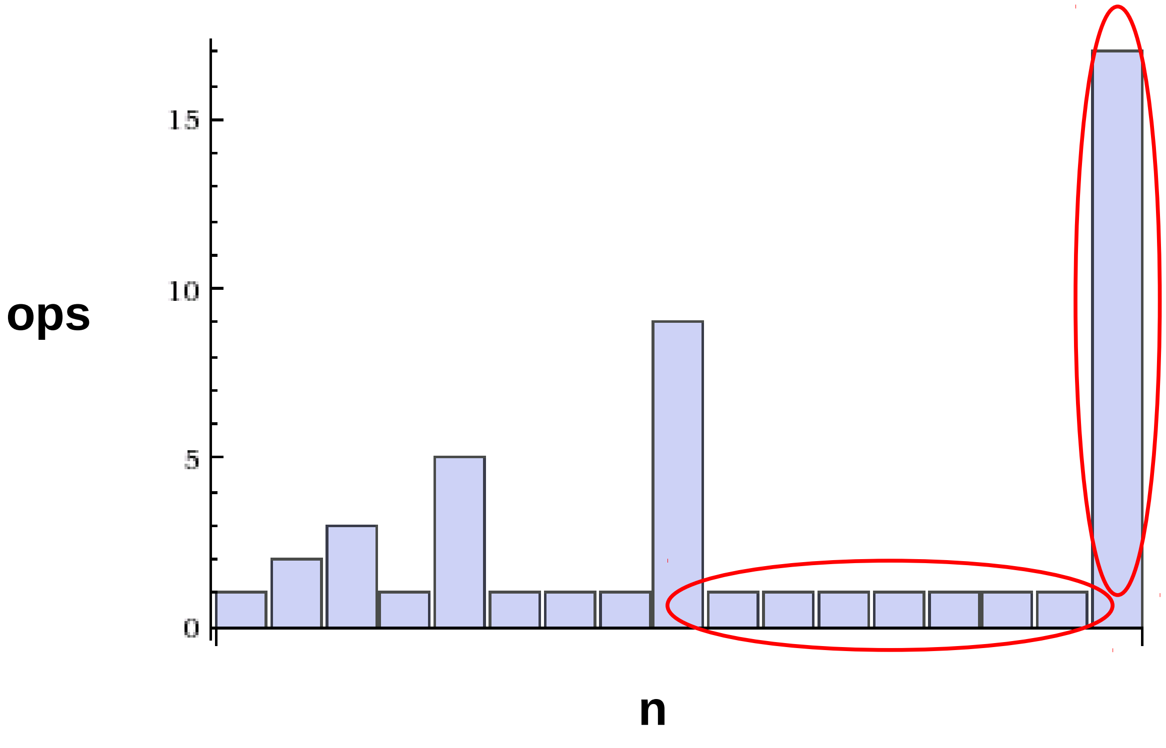
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency



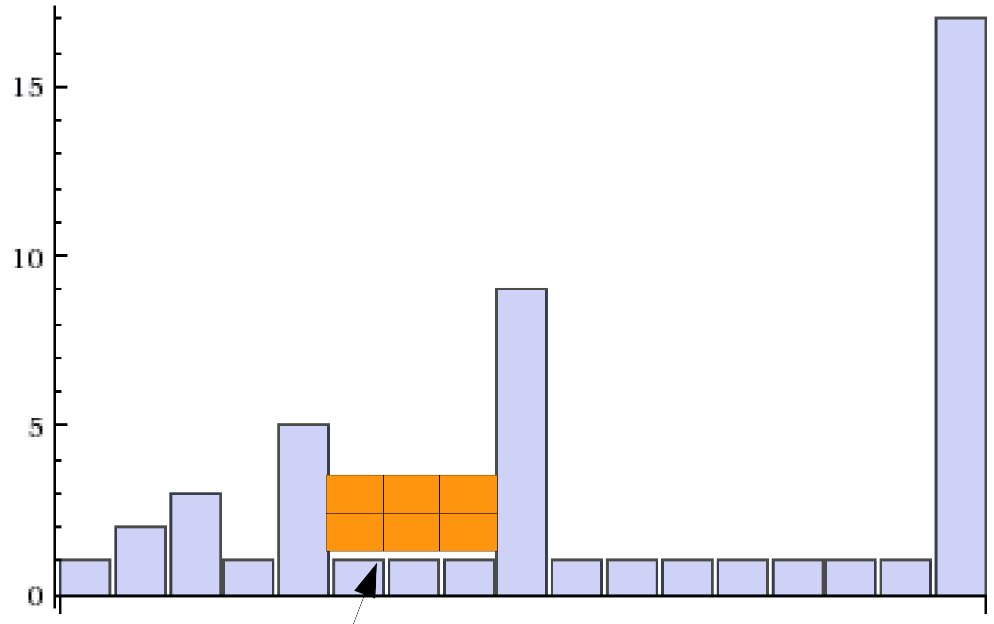
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency



Amortized Analysis

- Idea: Charge extra for O(1) insertions to “save up” and “pay for” the O(n) insertions



Charge extra here

Amortized Analysis

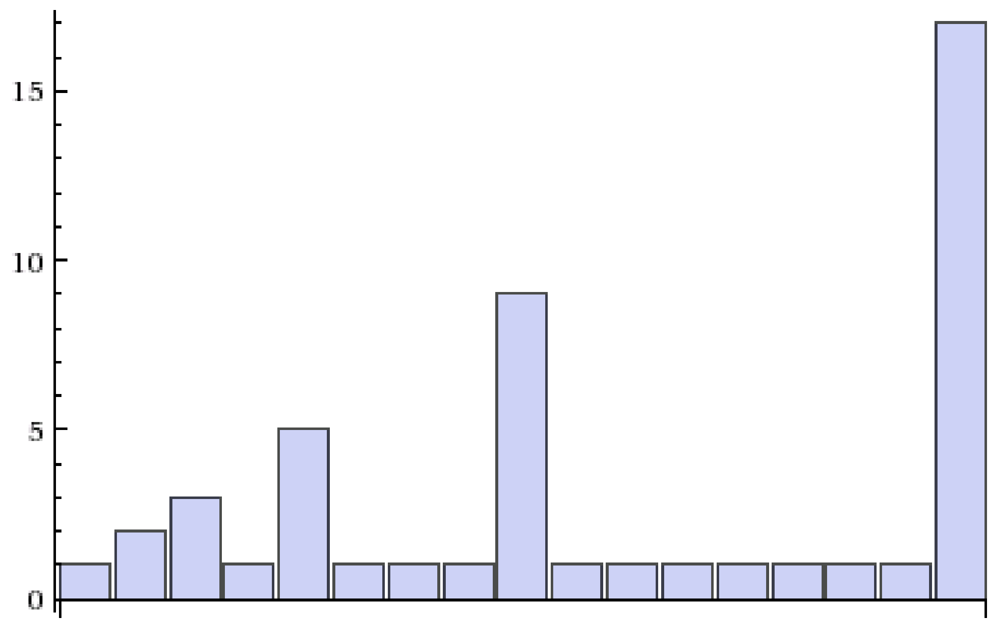
- Idea: Charge extra for O(1) insertions to “save up” and “pay for” the O(n) insertions

Amortized Analysis

- How much extra do we charge?
  - Let's try charging 1 extra operation
  - Total of 2 operations per append

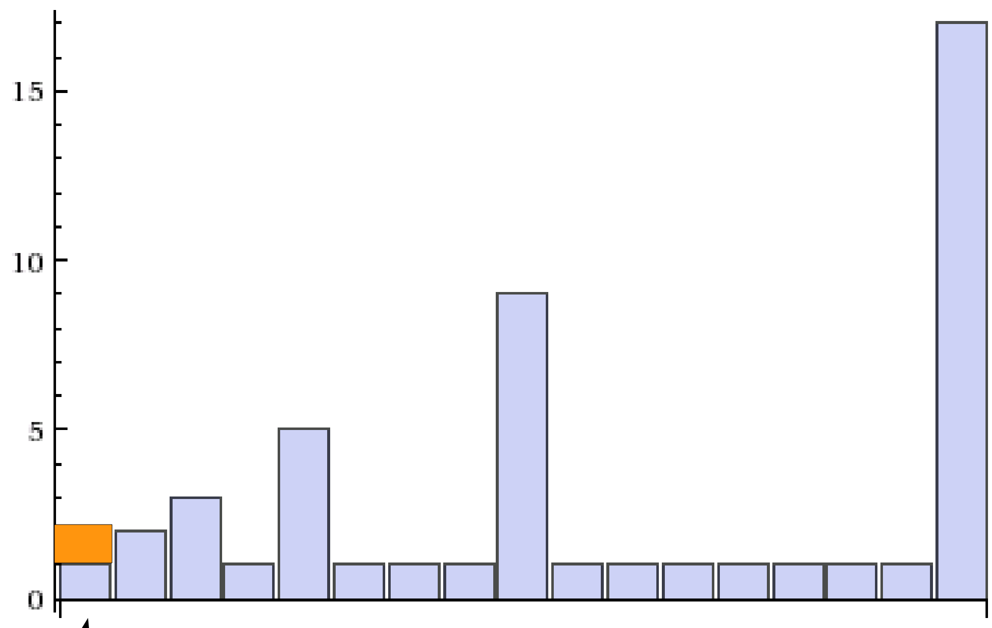
Amortized Analysis

- How much extra do we charge?



Amortized Analysis

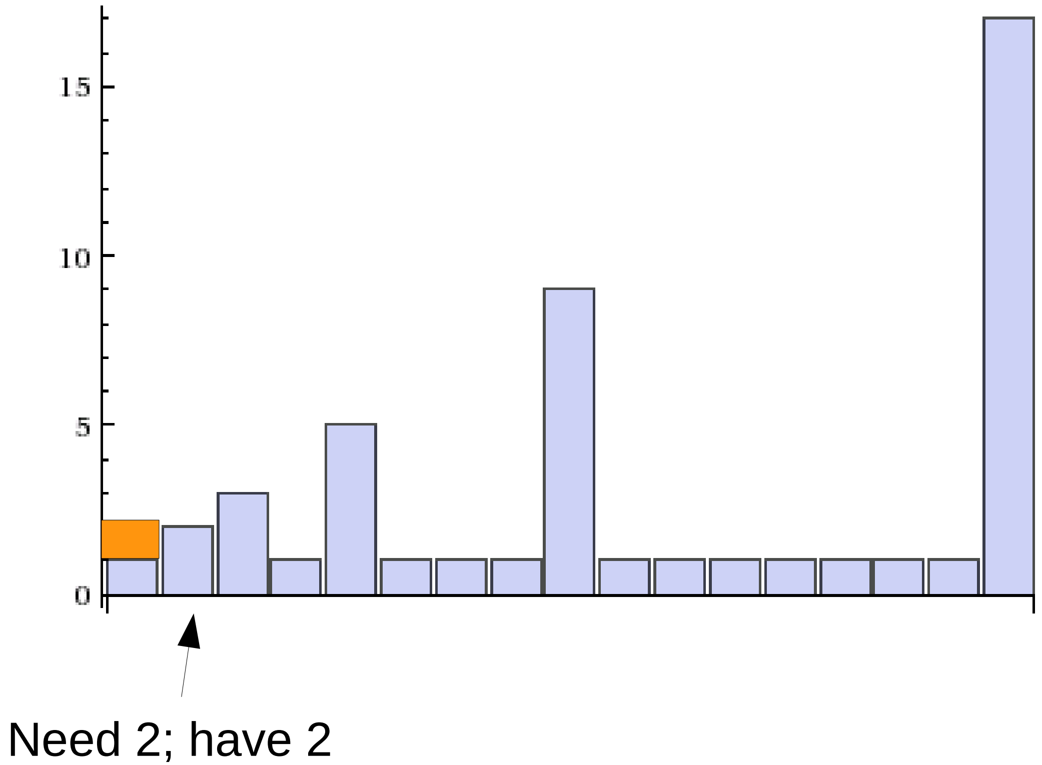
- How much extra do we charge?



Need 1; have 2; save one!

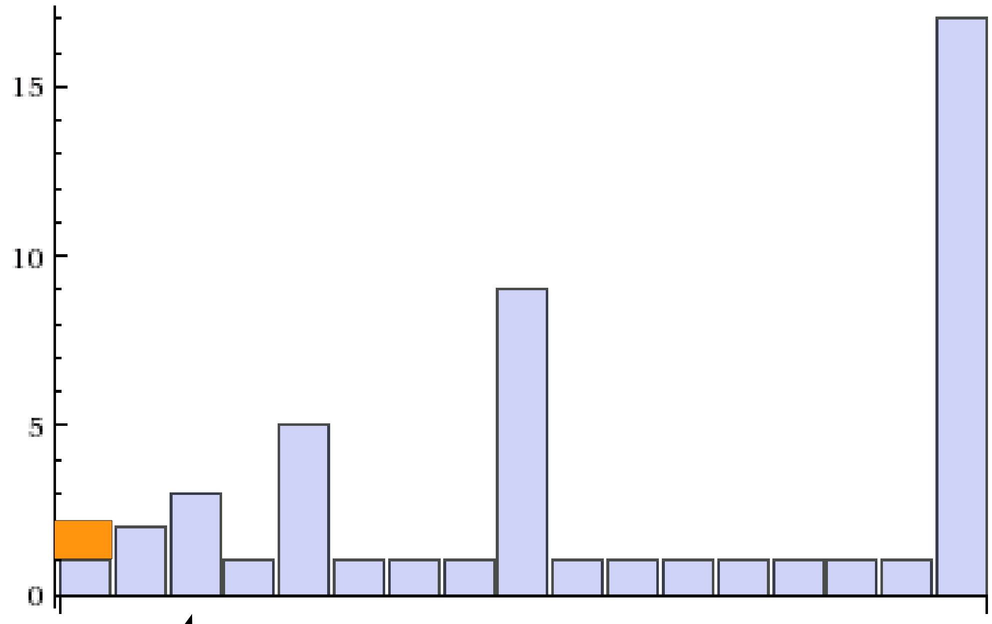
Amortized Analysis

- How much extra do we charge?



Amortized Analysis

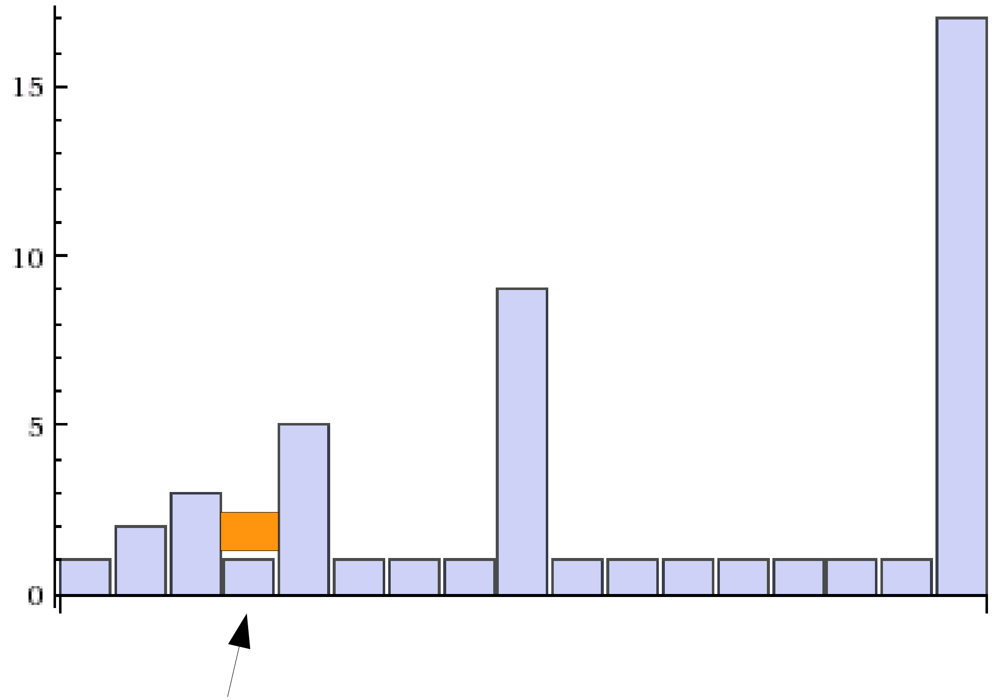
- How much extra do we charge?



Need 3; have 2+1=3

Amortized Analysis

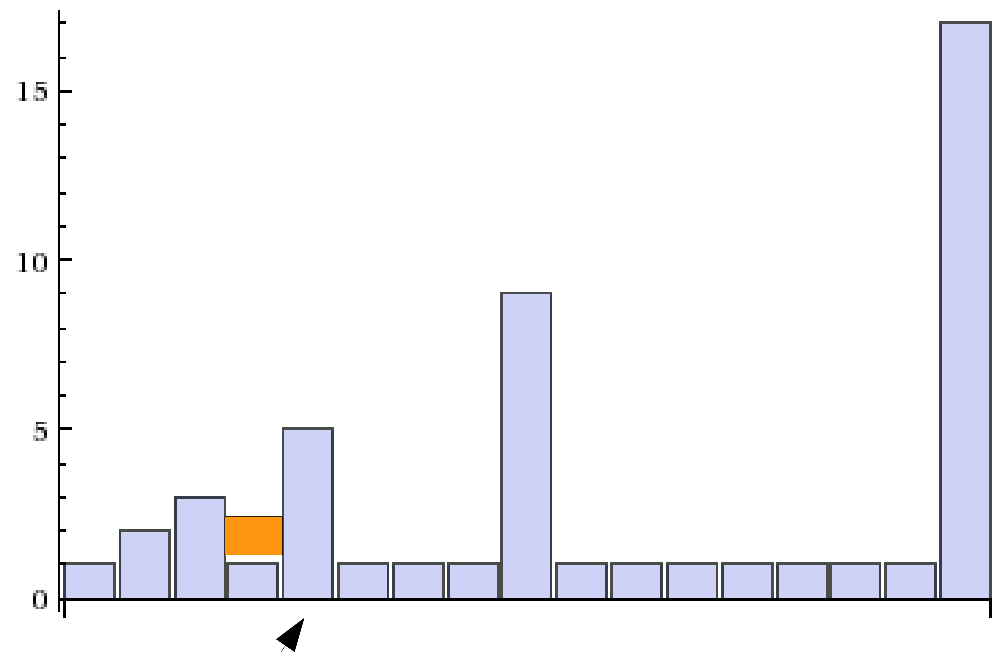
- How much extra do we charge?



Need 1; have 2; save one!

Amortized Analysis

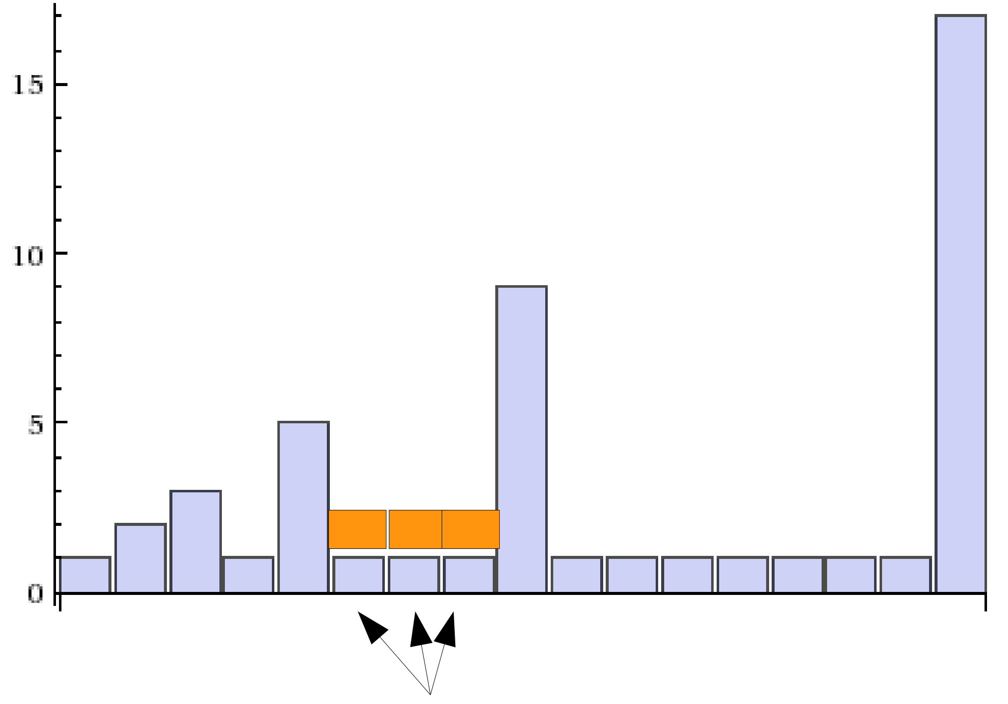
- How much extra do we charge?



Need 5; have 2+1=3 !!

Amortized Analysis

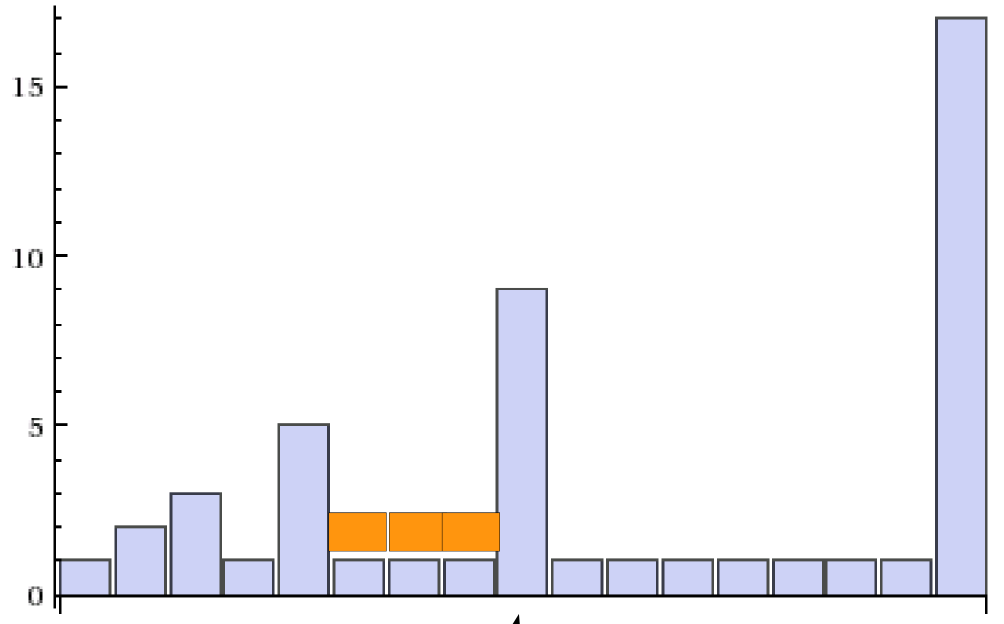
- How much extra do we charge?



Need 1, have 2; save one each!

Amortized Analysis

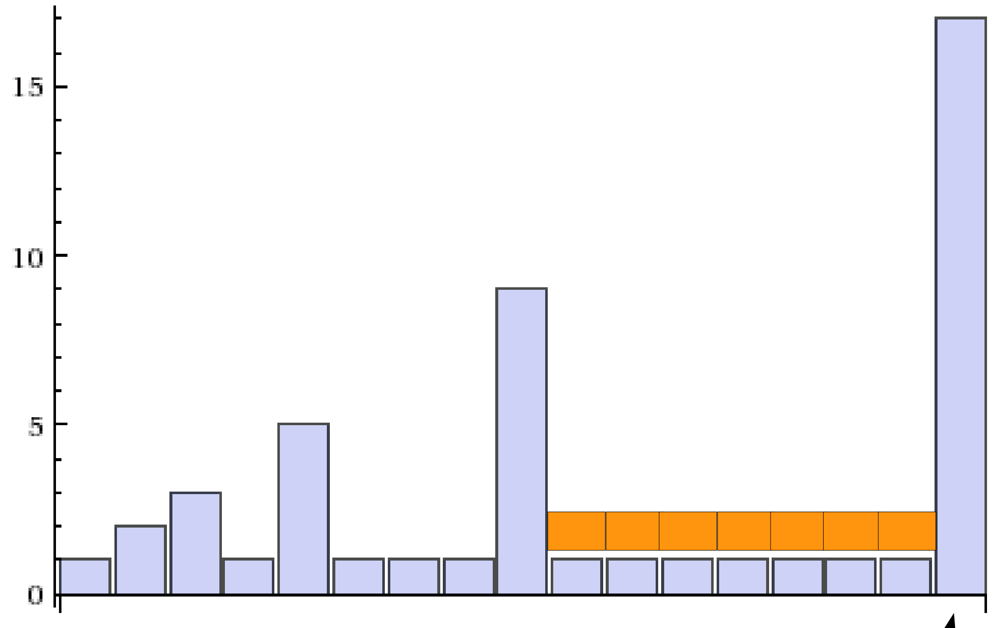
- How much extra do we charge?



Need 9, have 2+3=5 !!

Amortized Analysis

- How much extra do we charge?



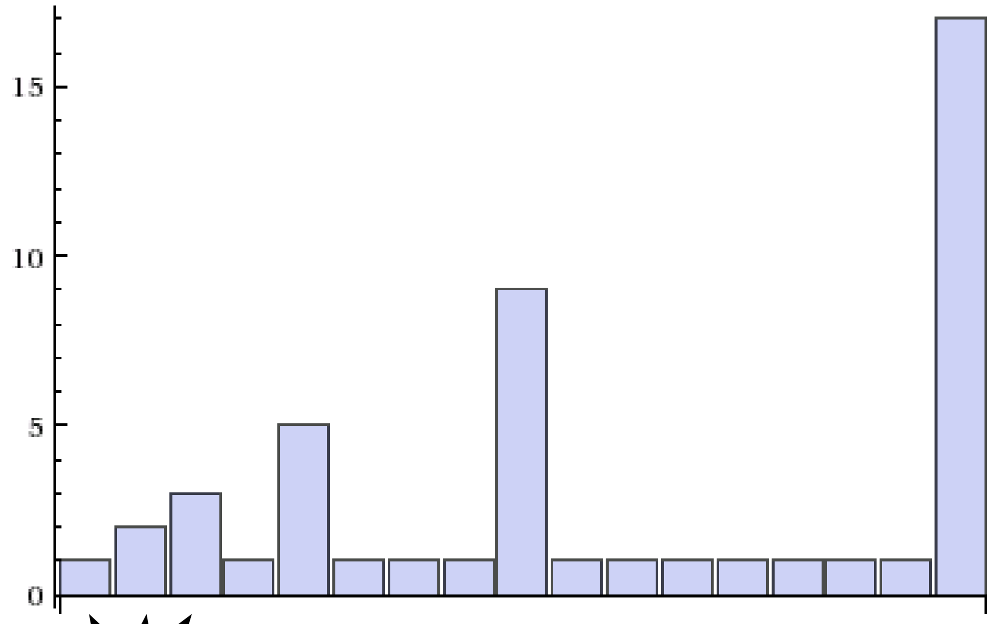
Need 17, have 2+7=9 !!

Amortized Analysis

- How much extra do we charge?
  - Let's try charging 2 extra operations
  - Total of 3 operations per append

Amortized Analysis

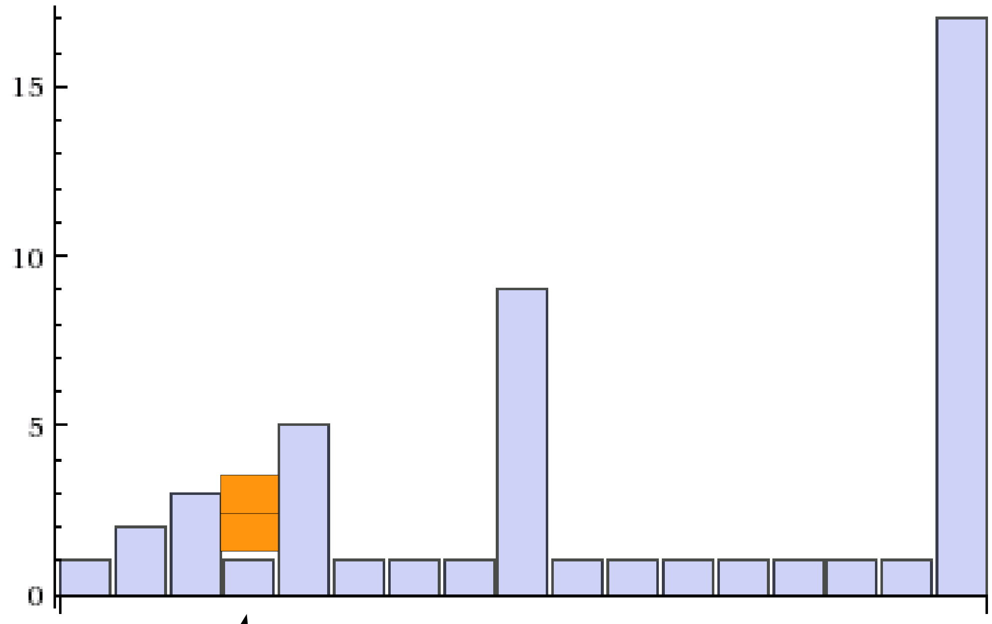
- How much extra do we charge?



Need <=3; have 3

Amortized Analysis

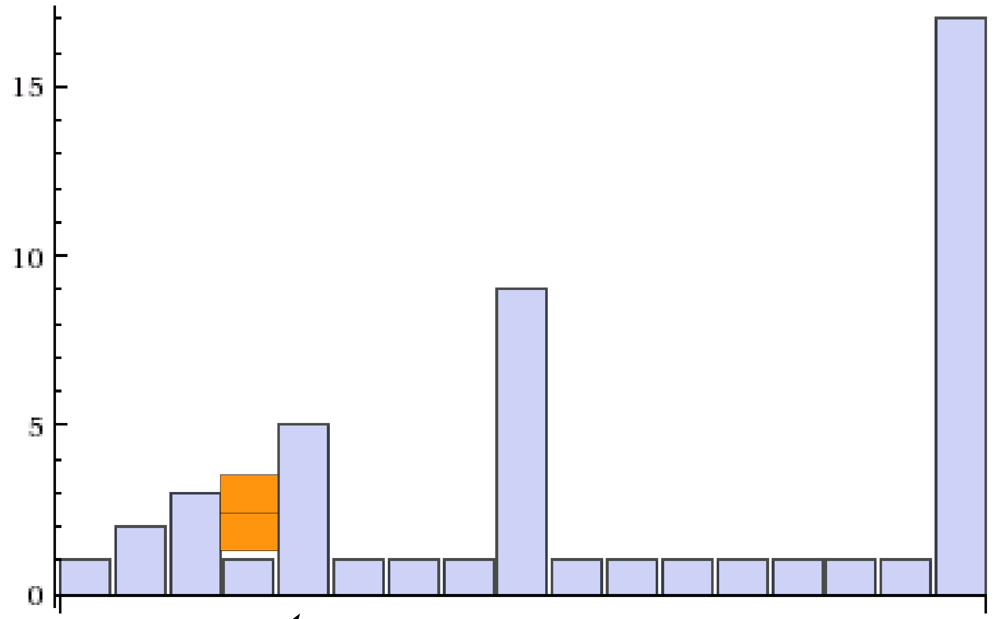
- How much extra do we charge?



Need 1; have 3; save two!

Amortized Analysis

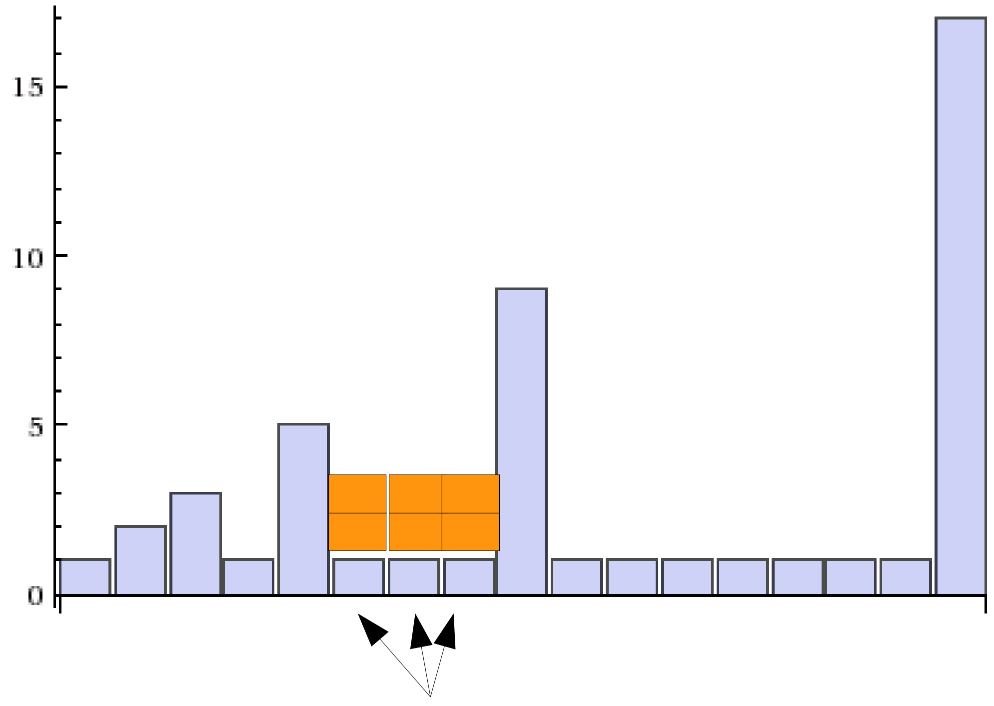
- How much extra do we charge?



Need 5; have 3+2=5

Amortized Analysis

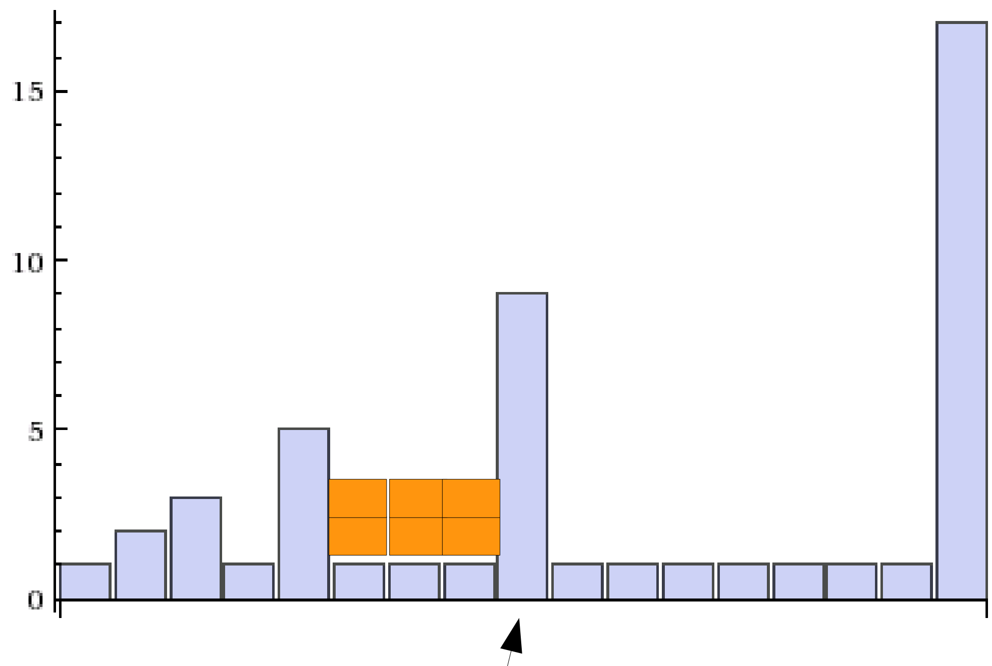
- How much extra do we charge?



Need 1, have 3; save two each!

Amortized Analysis

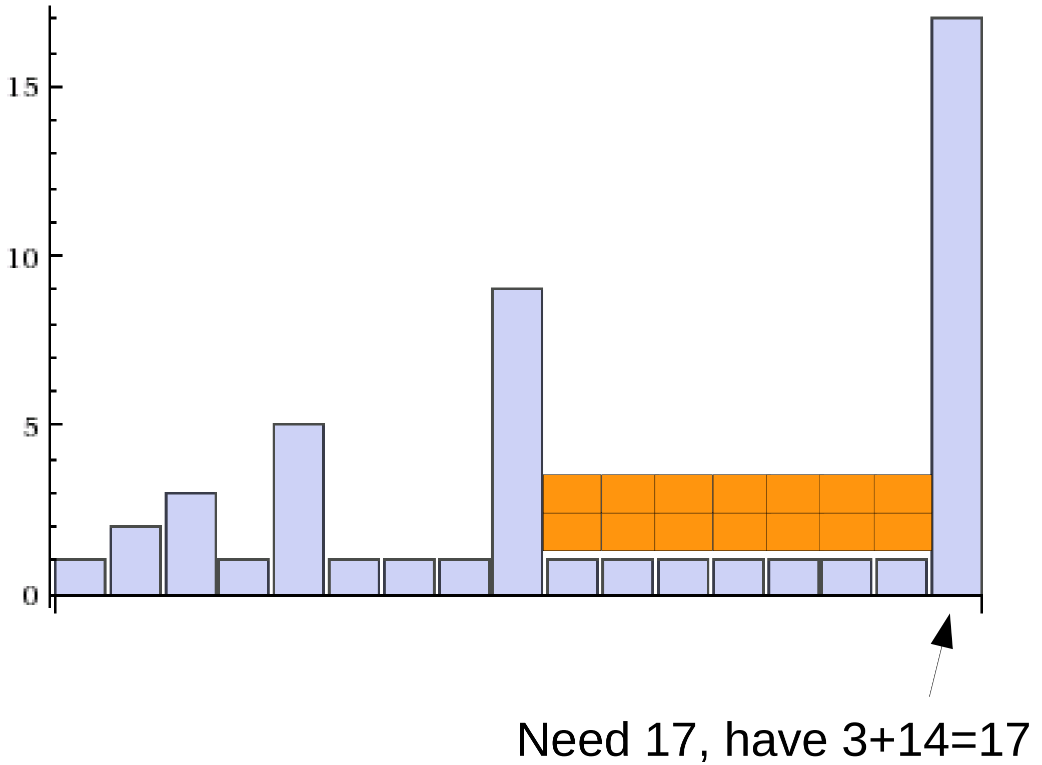
- How much extra do we charge?



Need 9, have 3+6=9

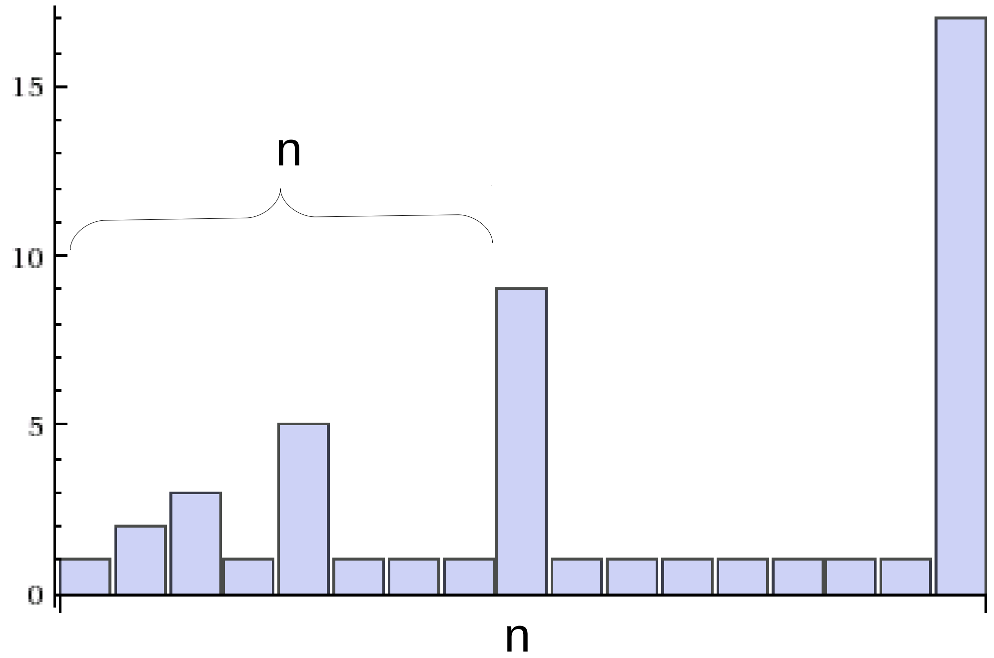
Amortized Analysis

- How much extra do we charge?



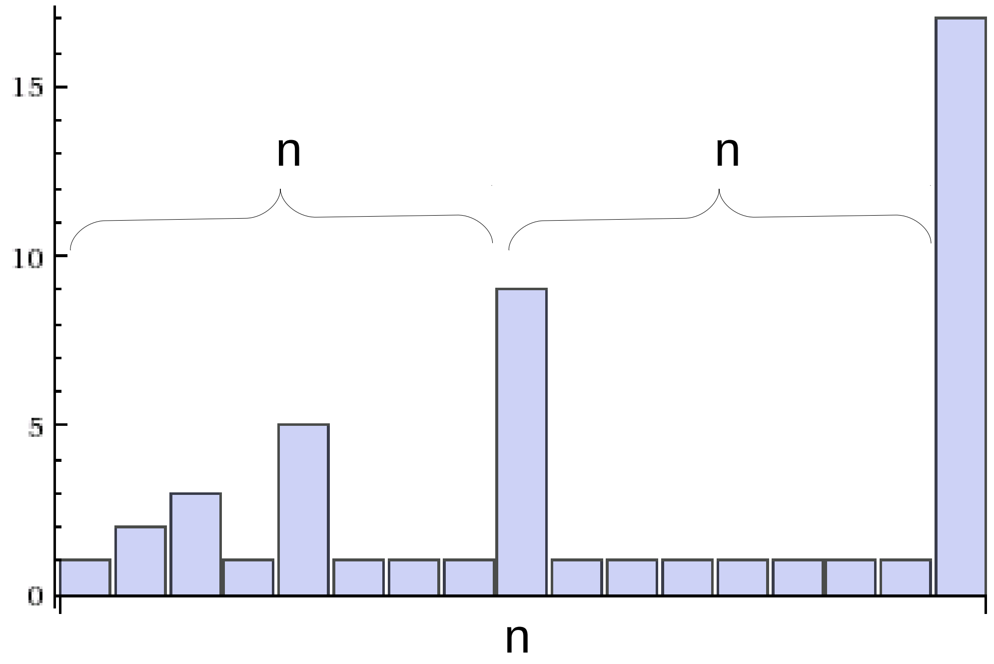
Amortized Analysis

- How much extra do we charge?



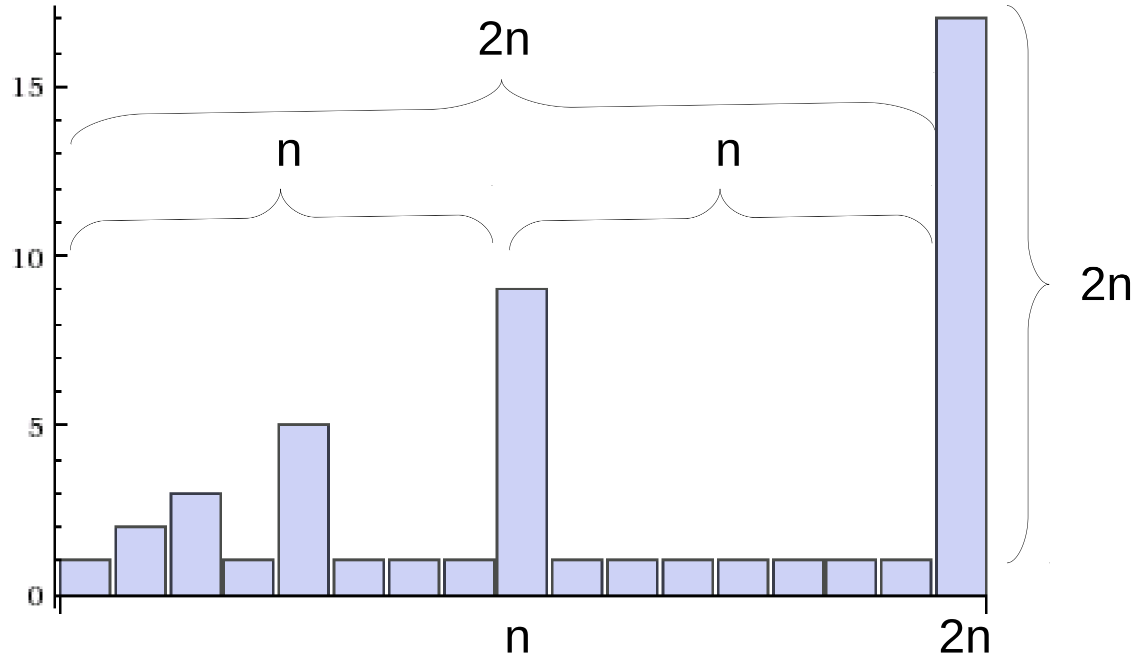
Amortized Analysis

- How much extra do we charge?



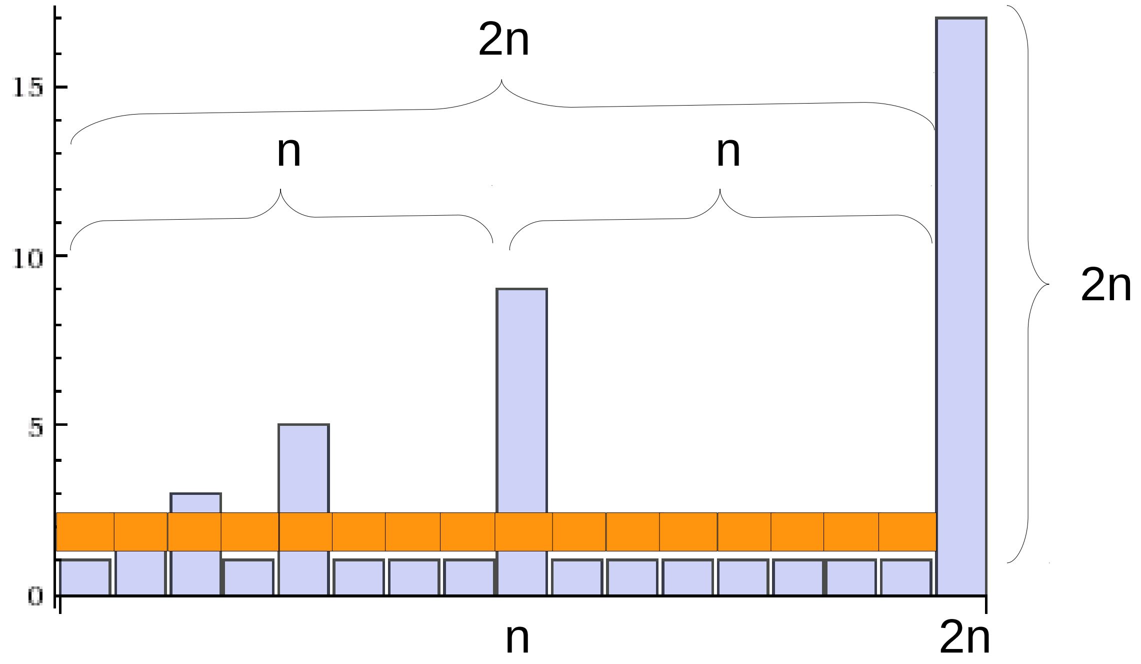
Amortized Analysis

- How much extra do we charge?



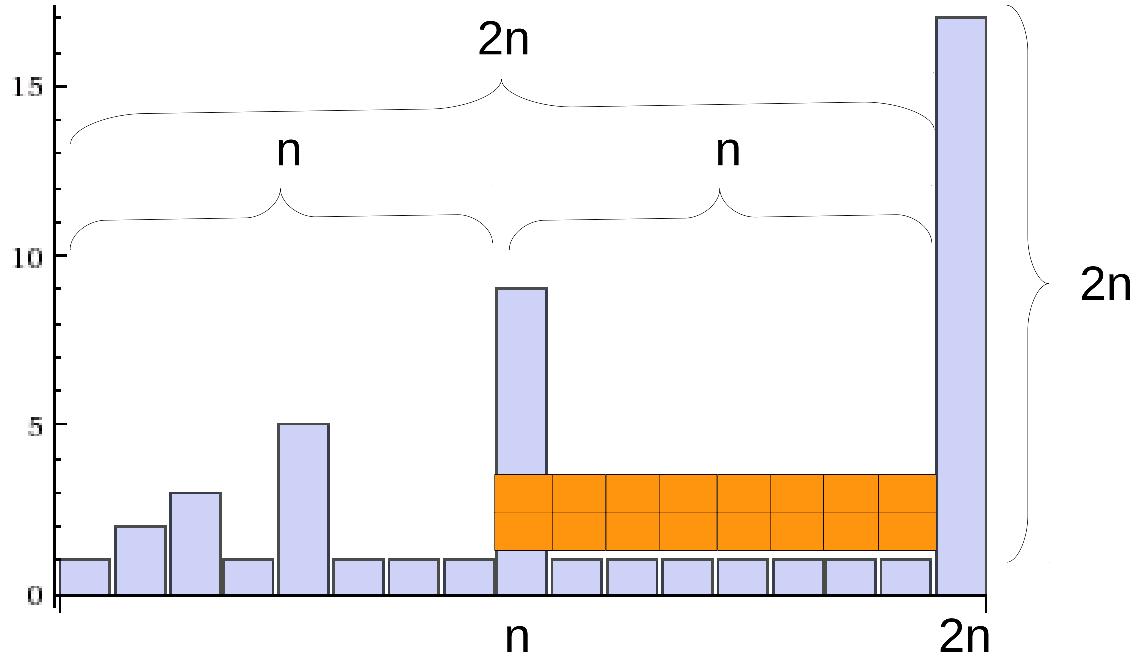
Amortized Analysis

- How much extra do we charge?



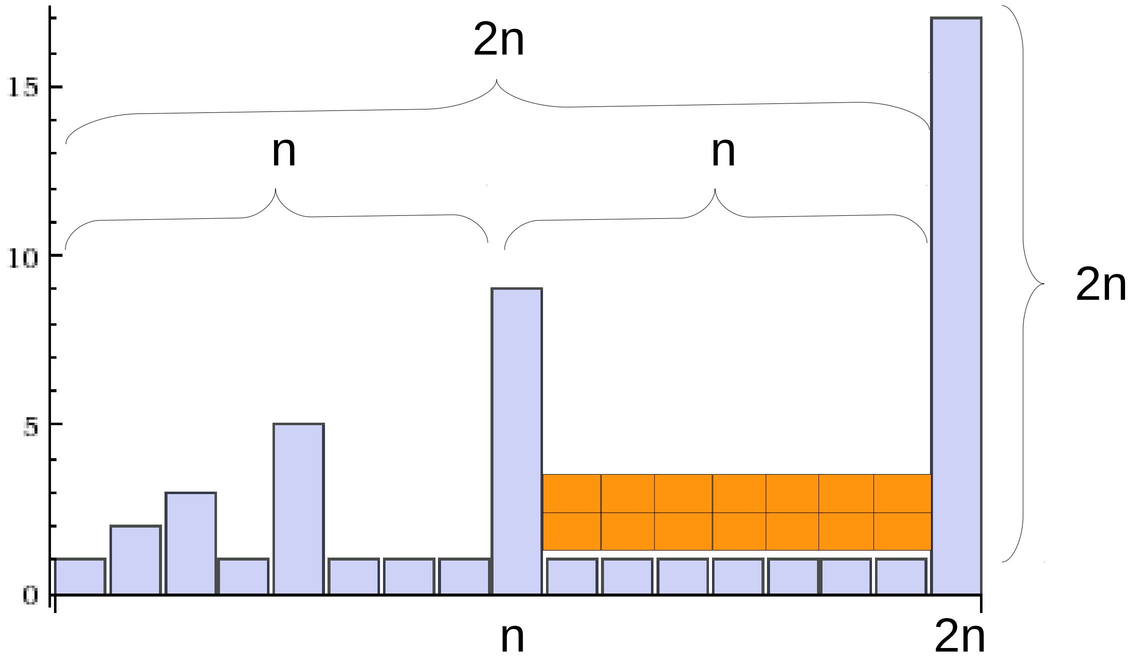
Amortized Analysis

- How much extra do we charge?



Amortized Analysis

- How much extra do we charge?



Amortized Analysis

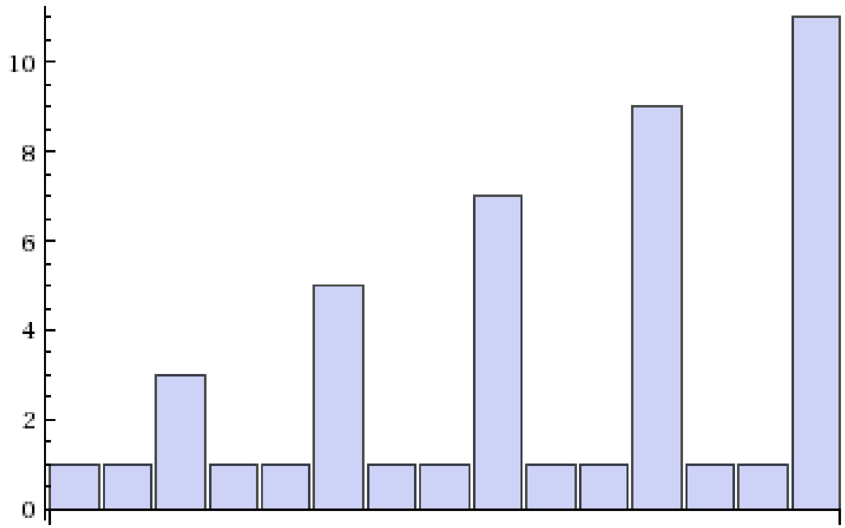
- How much extra do we charge?
  - If we're doubling the size each time…
    - We will need to make 2n copies at the next increase
    - We will have n new appends during that period
  - So we need to “save up” two extra operations per cheap append to pay for the expensive appends
  - Charge 3 total operations for each append

Amortized Analysis

- Total # of operations to add n items: 3n
  - Which is O(n)
- Average operations per append = 3n/n = 3
- More generally: the total # of operations is O(n so the amortized cost per append is O(1)

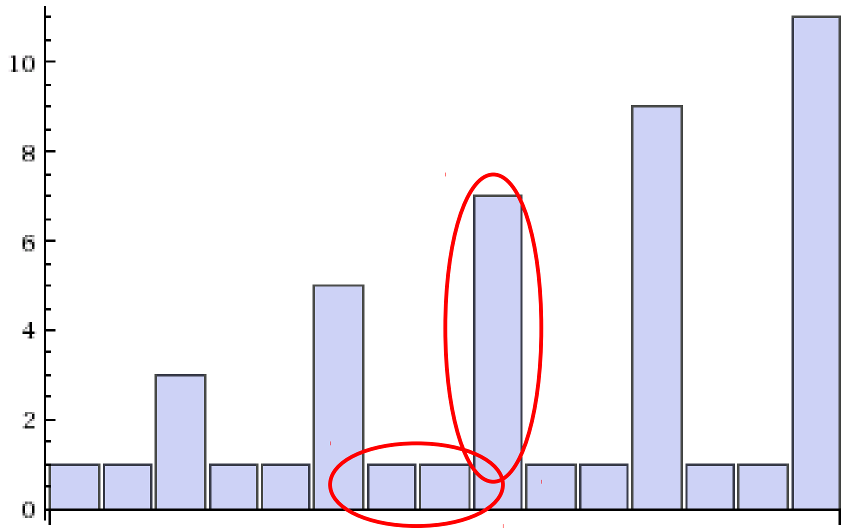
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?



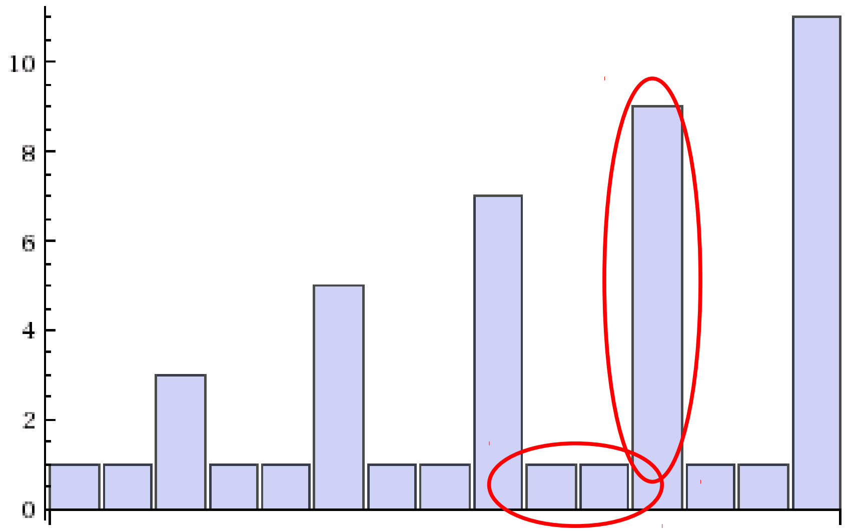
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?



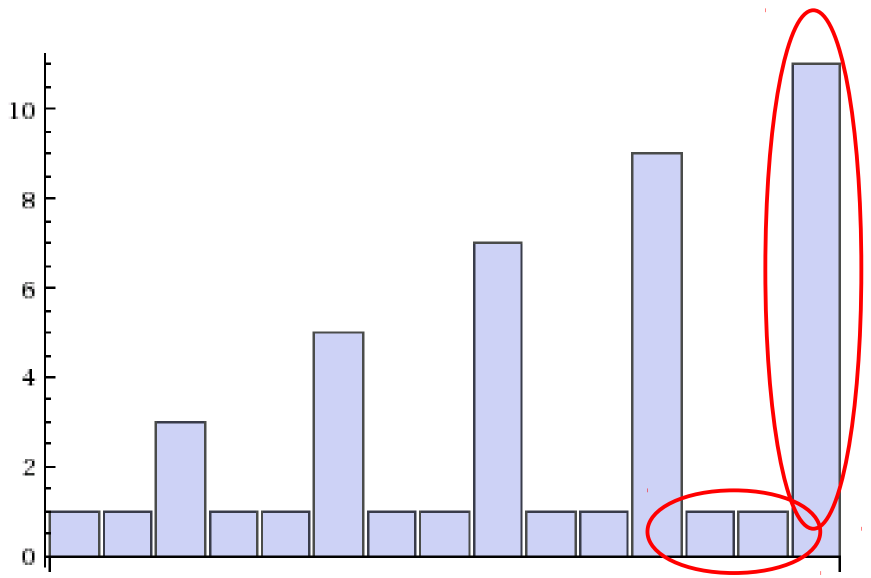
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?



Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?

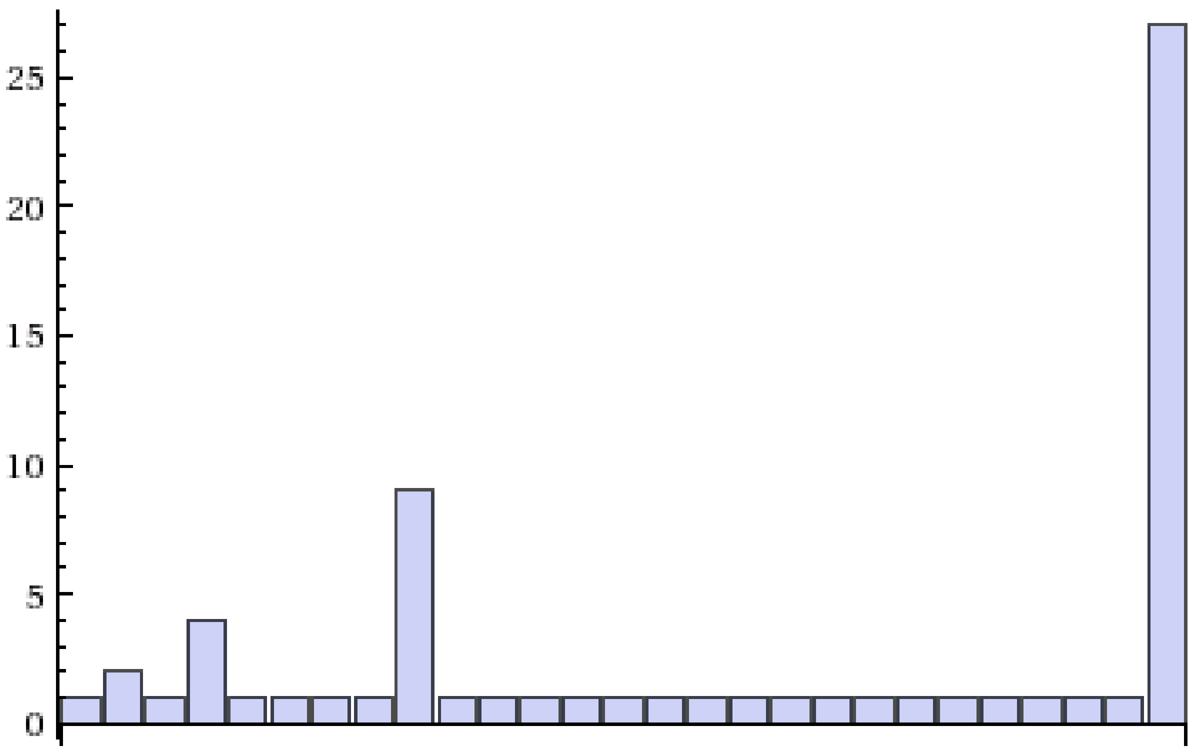


Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?
  - No! The amount of operations "saved" is always constant between increases, but the amount of work done by the capacity increases grows linearly with the size of the array.
  - This actually leads to $\Omega(n^2)$ total operations for n appends, instead of O(n) total operations

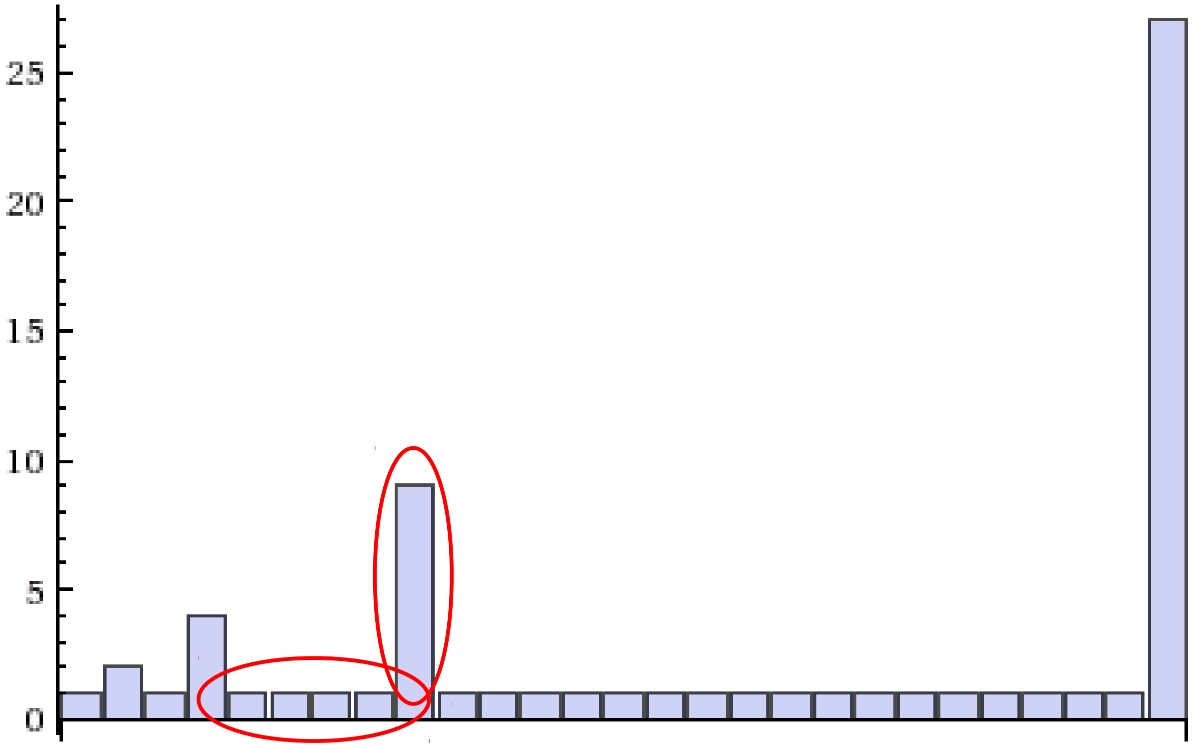
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?



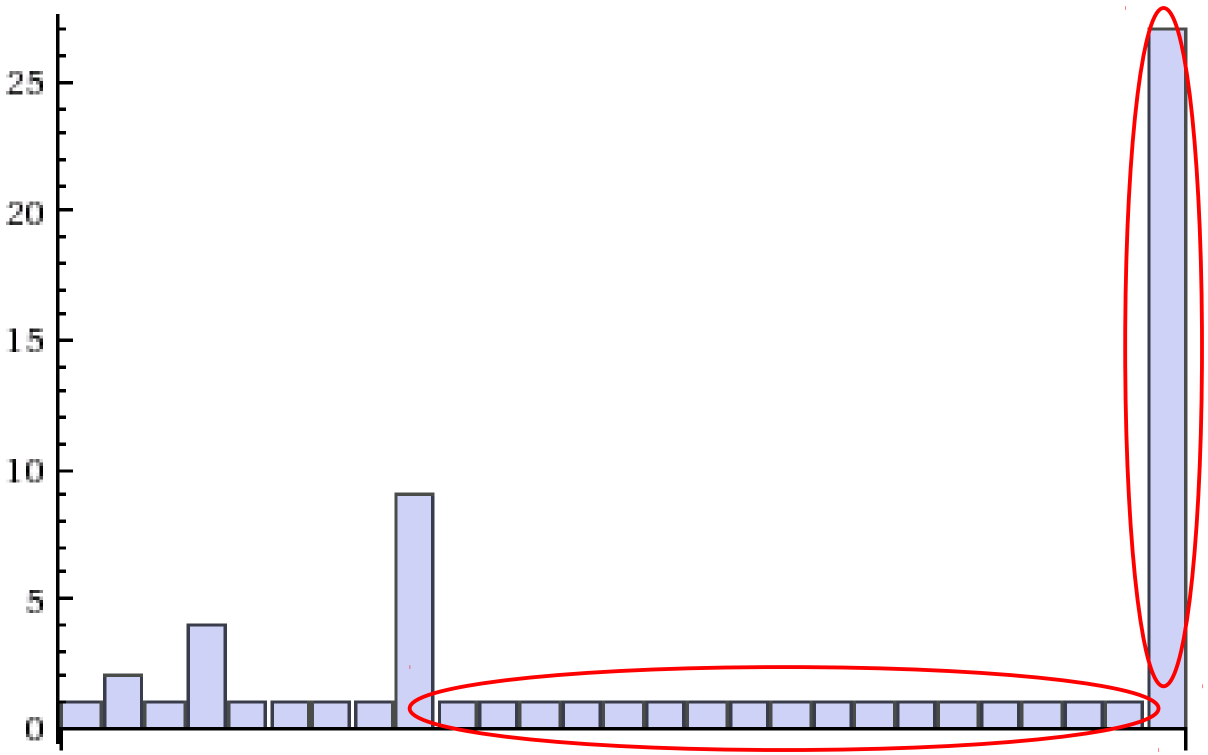
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?



Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?



Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?
  - Yes! Charge three extra operations instead of two, and then we will have saved roughly 3n operations before the next capacity increase.
  - Total operations for *n* appends: $4n \in O(n)$
    - The amortized cost for each append is still O(1)
  - In fact, the argument works for **any** geometric progression

Amortized Analysis

- Python lists don't use strict geometric progression
- But the average cost is still O(1)
- See Section 5.3.3 for evidence

Amortized Analysis

- Overcharge for cheap operations to “save up” credit for expensive operations
- If the total cost for *n* operations can be shown to be O(n), then the average cost for each individual operation is O(1)

Next Time

- Complexity classes for common operations on Python lists and strings