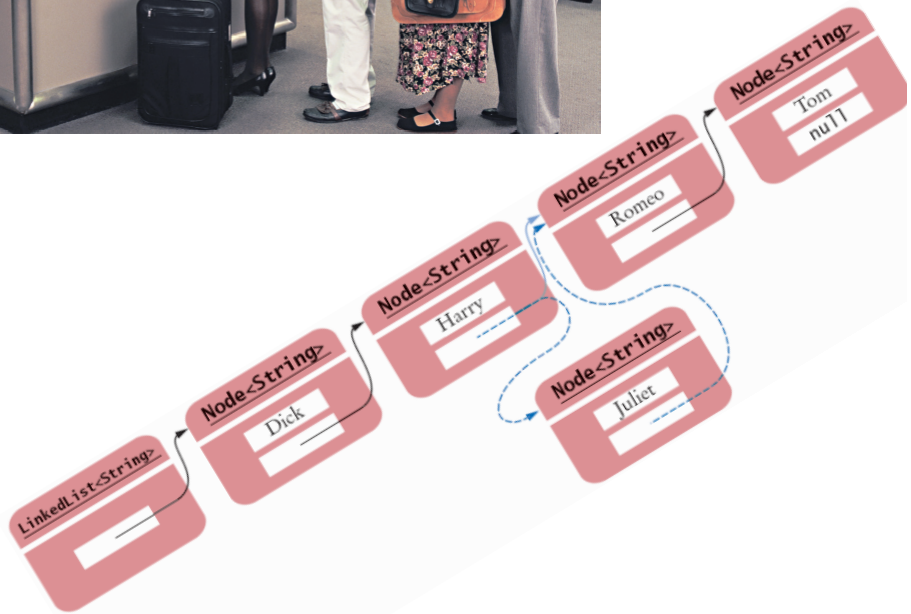
Basic Data Structures

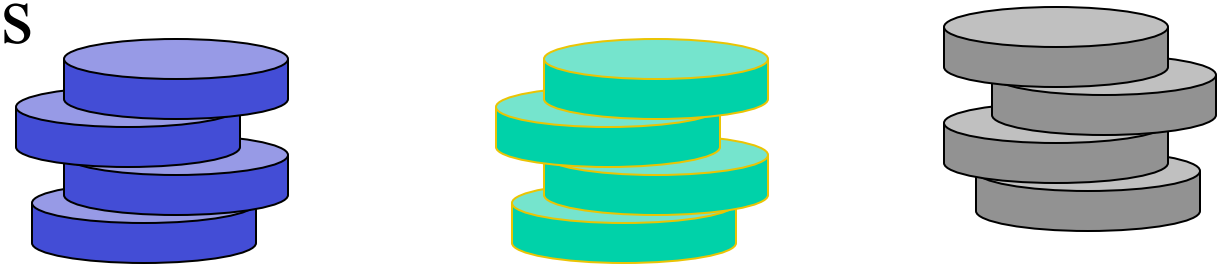
- Stacks
- Queues
- Lists

X	12	3	7	24	4	1	1
A	12	7.5	7.3	11.5	10	8.5	7.4



Abstract Data Type (ADT)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

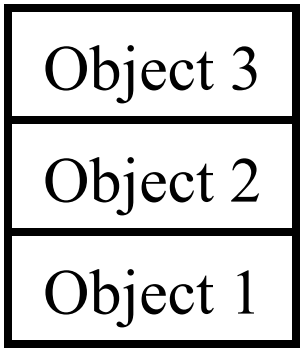


The Notion of a Stack

- Container of items
- Items are returned in reverse order of being added (LIFO)
- **Push** and **pop** items from the top of the stack
  - Stack of plates in cafeteria
  - Candy dispenser
- Examples
  - Solving a problem by completely solving every smaller problem that comes up (e.g., Quicksort, Divide and conquer algorithm)
  - Keeping track of the url's when browsing the web
  - "Undo" function of most applications that have a user interface
  - Runtime environment's handling of nested method calls
  - Recursive and nested method calls

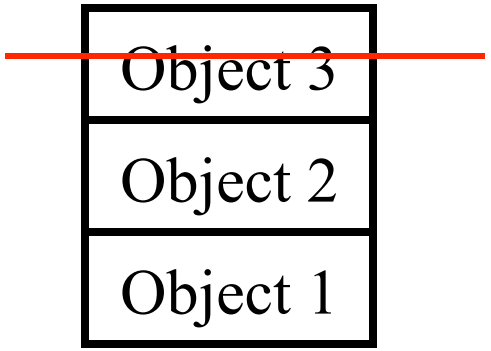
Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out



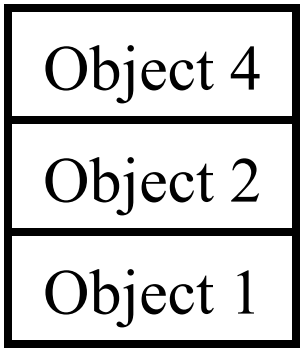
Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out



Stacks

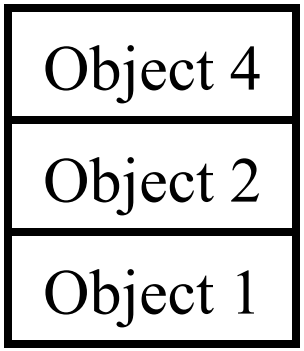
- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out



Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out

Can we remove Object 2 at this moment?



Removing an object from a stack

- Only the most recently inserted object can be removed at *any* time.
- Earlier inserted objects can only be removed if all objects that are inserted at a later time are already removed from the stack.

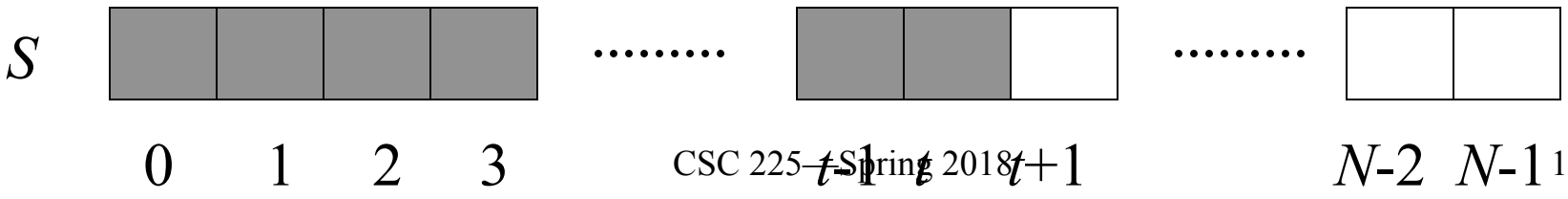
The Stack Abstract Data Type

A stack *S* is an abstract data type (ADT) supporting the following methods.

- **push(*o*):** Insert object *o* at the top of the stack
- **pop():** Remove from the stack and return the top object on the stack (that is, the most recently inserted element still in the stack); an error occurs if the stack is empty.
- **isEmpty():** Return a Boolean indicating if the stack is empty.
- t**op():** Return the top object on the stack without removing it; an error occurs if the stack is empty.
- **size():** Return the number of objects in the stack.

An Efficient Implementation of a Stack: The Simple Array-Based Stack

- S: N-element array, with elements stored from $S[0]$ to $S[t]$
- t: stack pointer; integer that gives the index of the top element in S
- N: specified max stack size (e.g., N=1500)



Algorithm push(*object*):

```
if size() = N then
    "indicate that the
    stack is full"
    return
end
t ← t + 1
S[t] ← object
return
```

Algorithm pop():

```
if isEmpty() then
    “indicate that the
    stack is empty”
    return
end
object ← S[t]
t ← t - 1
return object
```

What is the Running Time of push()?

```
Algorithm push(object):
  if size() = N then
     "indicate that the
     stack is full"
  return
  end
  t ← t + 1
  S[t] ← object
 return
```

What is the Running Time of pop()?

```
Algorithm pop():
  if isEmpty() then
     "indicate that the
     stack is empty"
  return
  end
  object ← S[t]
  t ← t - 1
  return object
```

Array-Based Implementations of a Stack: Advantages and Disadvantages

- Simple
- Efficient: O(1) per operation
- The stack *must* assume a fixed upper bound *N*
- Memory might be wasted or a stack-full error can occur!
- If good estimate for stack size is known: Array is the best choice!!

Run-time Stack

- The run-time environment for most programming languages uses a uses a stack to keep track of method invocations
- Each method call has an *activation record* or *stack frame* associated with it
- Whenever a call is made, a new activation record is allocated and *pushed* onto the stack
- When a call returns, its record is *popped* from the call stack
- Each activation record (frame) contains
  - Program counter for the current line of code (return code)
  - Space to hold all method parameters
  - Space to hold all method local variables
  - Space to hold the return value

Recursion

- A recursive method calls itself
  - `void a() { … a() … }`
- Indirect recursion
  - `void a() { … b() … }`
  - `void b() { … a() … }`
- Recursive calls of course are also realized with the run-time stack
- "Infinite Recursion" leads to stack overflow (out-of-memory error)

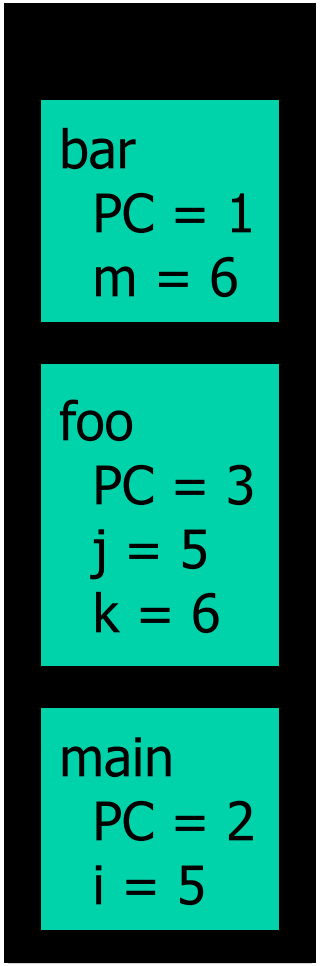
Run-time Stack

When a method terminates, its frame is popped off the stack and control is passed to the method on top of the stack (i.e., the calling method)

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



Postfix Notation

- The "normal" way to write arithmetic expressions is *infix notation*
  - because the operators are *between* the operands
- Expressions written in *postfix notation* are easier to evaluate
  - the operators are *after* the operands
  - there is no need for parenthesis
  - there is no need for operator precedence rules

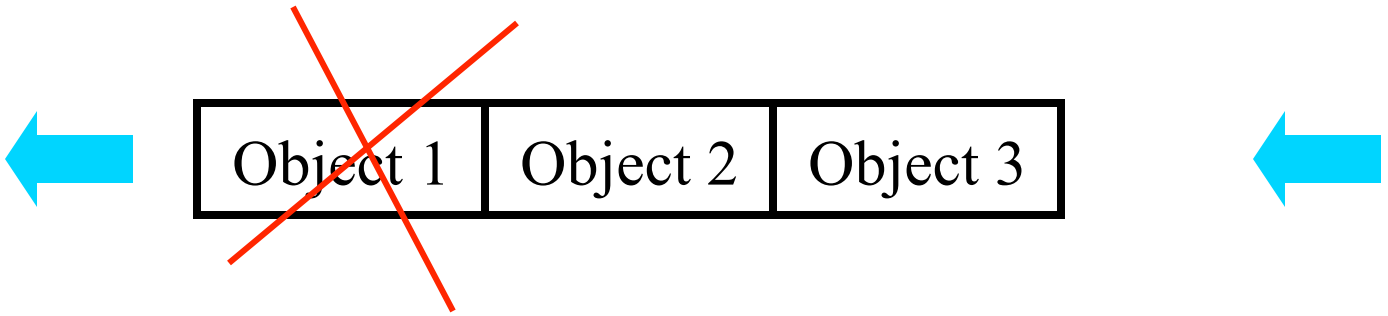
Infix Form	Postfix Form	Value
34	34	34
34 + 22	34 22 +	56
34 + 22 * 2	34 22 2 * +	78
34 * 22 + 2	34 22 * 2 +	750
(34 + 22) * 2	34 22 + 2 *	112

Queues

- Container of items that are inserted and removed following the FIFO principle FIFO = first-in first-out
- Next up is always the item that has been in the queue the longest
- Examples:
  - people waiting for a carnival ride
  - multi-user operating system's time-sharing
  - customer number systems at the bakery
  - waitlists for classes
  - Priority queues

Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time



Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Can we remove Object 2 at this moment?

Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Can we remove Object 3 at this moment?

Removing an object from a queue

- Only the element that has been in the queue the longest can be removed at any time
- Later inserted objects can only be removed if all objects that are inserted at an earlier are already removed from the queue.

The Queue Abstract Data Type

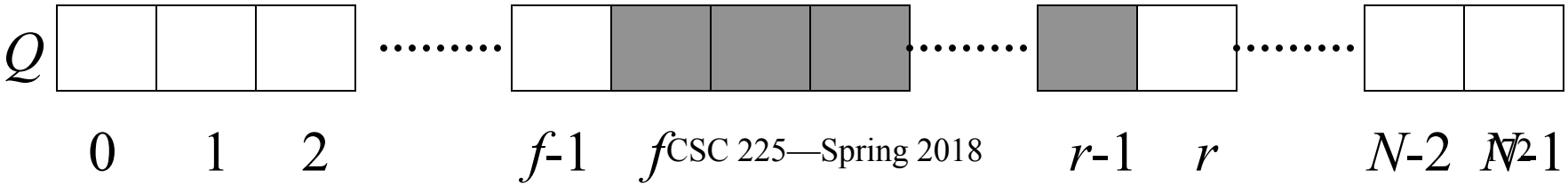
A queue *Q* is an abstract data type (ADT) supporting the following methods:

- **enqueue(*o*):** Insert object *o* at the rear of the queue
- **dequeue():** Remove and return from the queue the object at the front; an error occurs if the queue is empty
- **isEmpty():** Return a Boolean indicating if the queue is empty
- **front():** Return, but not remove, the front object in the queue; an error occurs if the queue is empty
- **size():** Return the number of objects in the queue

An Efficient Implementation of a Queue: The Simple Array-Based Queue

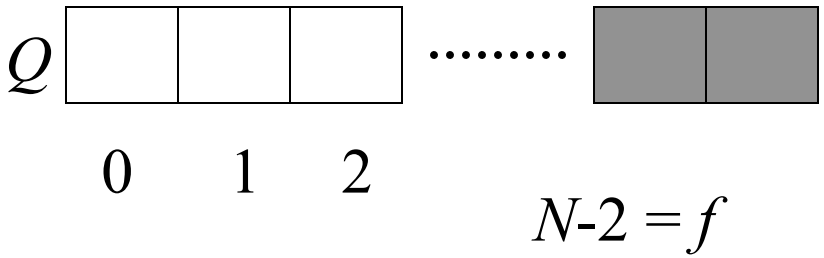
- Q: N-element array
- f: index to the cell of Q storing the first element of Q (init is f=0), unless the queue is empty ($f = r$)
- r: index to the next available array cell in Q (init is r=0)
- N: specified maximum queue size (e.g., N=1500)

$f = r$ indicates Q is empty



The Simple Array-Based Queue: "wrap around"

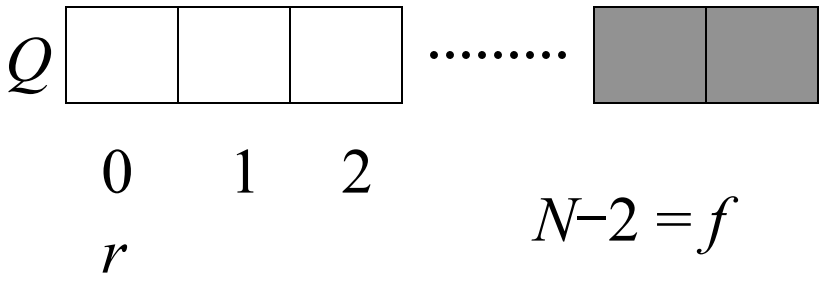
- What if for example $f = N-2$? How many elements can be stored in Q?



$r = ?$

The Simple Array-Based Queue: "wrap around"

- What if for example $f = N-2$? How many elements can be stored in Q?



Count modulo N!

$$x \bmod y = x - \lfloor x / y \rfloor y, y \neq 0$$

Another problem

- What happens if we enqueue N objects without any dequeuing?

We obtain $f = r$! (Which implies that the queue is empty)

Algorithm enqueue(o):

```
if size() = N-1 then
    throw a QueueFullException
Q[r] ← o
r ← (r + 1) mod N
```

Algorithm dequeue():

```
if isEmpty() then
    throw a QueueEmptyException
temp ← Q[f]
f ← (f + 1) mod N
return temp
```

What is the running time?

```
Algorithm enqueue(o):
  if size() = N-1 then
     throw a QueueFullException
  Q[r] ← o
  r ← (r + 1) mod N
  return
```

What is the running time?

```
Algorithm dequeue():
    if isEmpty() then
        throw a QueueEmptyException
    temp ← Q[f]
    f ← (f + 1) mod N
    return temp
```

Array-Based Implementations of a Queue: Advantages and Disadvantages

- Simple
- Efficient: O(1) per operation
- The queue has a fixed upper bound N (for $N-1$ elements in a full queue)
- If a good estimate for the size of the queue is known: an array is the best choice!

The List ADT

Supported methods for a list *S*

- **first():** Return position of 1st element of *S* (error occurs if *S* empty)
- **last():** Return position of last element of *S* (error occurs if *S* empty)
- **isFirst(*p*):** Return a Boolean value (true for *p* is 1st position, false otherwise)
- **isLast(*p*):** Return a Boolean value (true for *p* is last position, false otherwise)
- **before(*p*):** Return position of the element of *S* preceding the one at position *p* (error occurs if *p* is 1st element)
- **after(*p*):** Return position of the element of *S* following the one at position *p* (error occurs if *p* is last element)

The List ADT …

Supported methods for a list *S*

- **replaceElement(*p*, *e*):** Replace the element at position *p* with *e*, the element that was at position *p* first is returned
- **swapElements(*p*, *q*):** Swap elements stored at positions *p* and *q*
- **insertFirst(*e*):** Insert a new element *e* into *S* as the first element
- **insertLast(*e*):** Insert a new element *e* into *S* as the last element

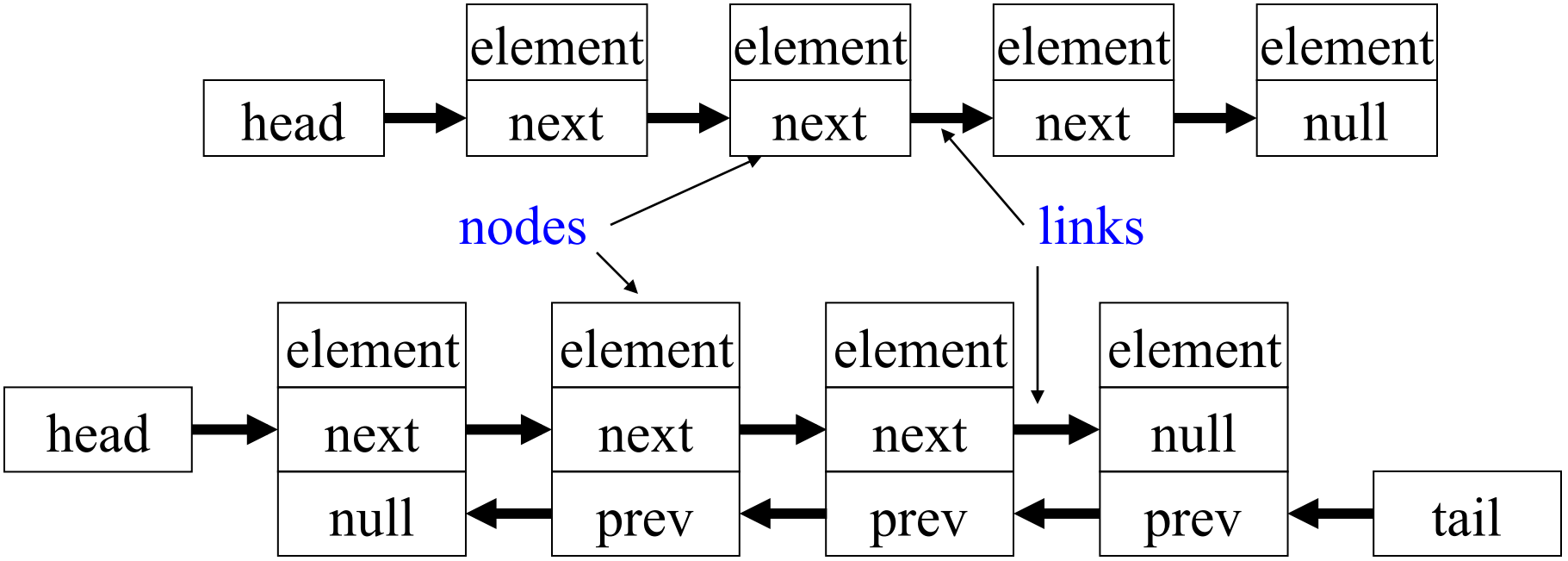
The List ADT …

Supported methods for a list *S*

- **insertBefore(*p*, *e*):** Insert a new element *e* into *S* before position *p* (error occurs if *p* is 1st element)
- **insertAfter(*p*, *e*):** Insert a new element *e* into *S* after position *p* (error occurs if *p* is last element)
- **remove(*p*):** Remove from *S* the element at position *p*

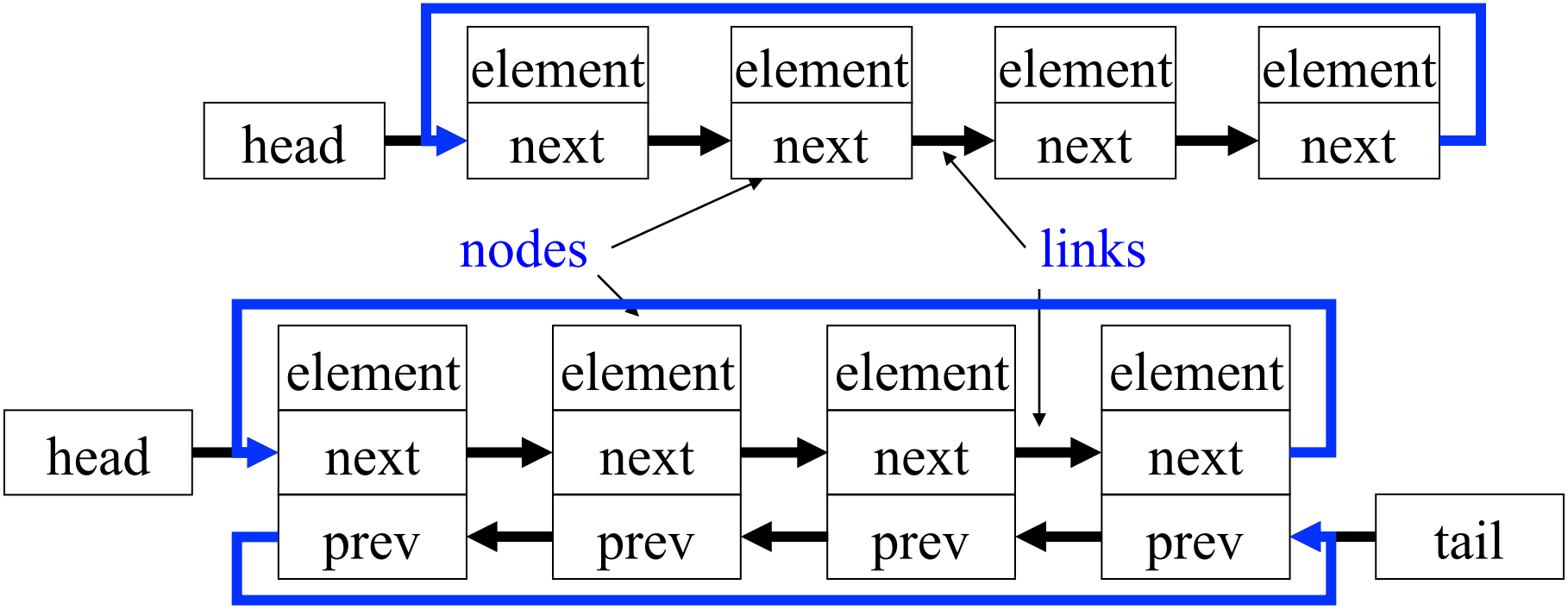
Singly and Doubly Linked Lists

- A *position* of an element is defined *relatively* (i.e., in terms of its neighbors)



Circular Lists

- A *position* of an element is defined *relatively* (i.e., in terms of its neighbors)



Algorithm insertAfter(p, e)

Doubly linked list

```
Create a new node v
v.element ← e
v.prev ← p
v.next ← p.next
(p.next).prev ← v
p.next ← v
return v
```

Algorithm remove(p)

Doubly linked list

```
t ← p.element
(p.prev).next ← p.next
(p.next).prev ← p.prev
p.prev ← null
p.next ← null
return t
```

Running Times

- first(): O(1)
- last(): O(1)
- isFirst(p): O(1)
- isLast(p): O(1)
- before(p): O(1)
- after(p): O(1)
- replaceElement(p, e): O(1)
- swapElements(p, q): O(1)
- insertFirst(e): O(1)
- insertLast(e): O(1)
- insertBefore(p, e): O(1)
- insertAfter(p, e): O(1)
- remove(p): O(1)

Sorting

Reading Assignment: Chapter 2

- **Sorting definition.** The process of ordering a sequence of objects according to some linear order.
- **Total versus partial order.** If any two elements in a set are comparable, then the set can be totally ordered otherwise partially ordered.
- Total order
  - 9 9 14 17 86
  - Coady Müller Stege Storey Thomo
- Partial order (Topological sort)
  - CSC 110<CSC 115<CSC 225 | SENG 321<SENG 371<SENG 426

Sorting

Reading Assignment: Chapter 2

- The basic unit for analyzing the time complexity of sorting algorithms is a comparison
  - $< \leq = \geq >$ int, char, string
  - $\leq = \geq$ should not be used for real or double
- Standard character sets
  - ASCII (256 characters)
  - Unicode (95,221 characters)
    http://www.i18nguy.com/unicode/char-count.html
  - ISO
  - EBCDIC

Sorting Reference Lists

Unsorted

Thagard, P., & Verbeurgt, K. (1998). *Coherence as constraint satisfaction.*

⋮

Garey, M. R. & Johnson, D. S. (1979). *Computers and intractability.*

⋮

Flood, M. M. (1956). The traveling-salesman problem. *Operations Research.*

⋮

Gross, J. & Yellen, J. (1999). *Graph theory and its applications*.

Sorted

Flood, M. M. (1956). The traveling-salesman problem. *Operations Research.*

Garey, M. R. & Johnson, D. S. (1979). *Computers and intractability.*

Gross, J. & Yellen, J. (1999). *Graph theory and its applications*.

⋮

Thagard, P., & Verbeurgt, K. (1998). *Coherence as constraint satisfaction.*

Sorting Address/Phone Books

Unsorted

J. Smith, 1420 Cook street, (250) 234 5627

⋮

D. S. Johnson, 130 Fort street, (250) 380 5627

⋮

M. Brent, 1110 Quadra street, (250) 380 9876

⋮

J. T. Vui, 230 Linden Ave., (250) 240 6517

⋮

R. Tristan, 1200 Dallas Road, (250) 340 3425

Sorted

M. Brent, 1110 Quadra street, (250) 380 9876

⋮

D. S. Johnson, 130 Fort street, (250) 380 5627

⋮

J. Smith, 1420 Cook street, (250) 234 5627

⋮

R. Tristan, 1200 Dallas Road, (250) 340 3425

⋮

J. T. Vui, 230 Linden Ave., (250) 240 6517

Index-Entries are Sorted

(after **Dictionary of Algorithms and Data Structures**
At http://www.nist.gov/dads/)

A

abstract data type

accepting state

Ackermann's function

acyclic graph

algorithm

amortized cost

approximation algorithm

array

B

backtracking

big-O notation

binary tree

M

matching

matrix

merge sort

Z

Zeller's congruence

Zipfian distribution

Zipf's law

Classes of Sorting Algorithms

Algorithm	**Complexity**	**Remarks**
Insertionsort	$O(n^2)$	Simple
Bubblesort	$O(n^2)$	
Selectionsort	$O(n^2)$	
Shellsort	$O(n^2)$	
Mergesort	O(n log n)	External sorting
Quicksort	$O(n^2)$ worst O(n log n) expected	Hoare 1962 Most practical
Heapsort	O(n log n)	
Smoothsort	O(n log n) O(n) best case	Dijkstra 1979
Lexicographic	O(n)	Restricted input
Bin or radixsort	O(n)	Restricted input

Animations of Sorting Algorithms

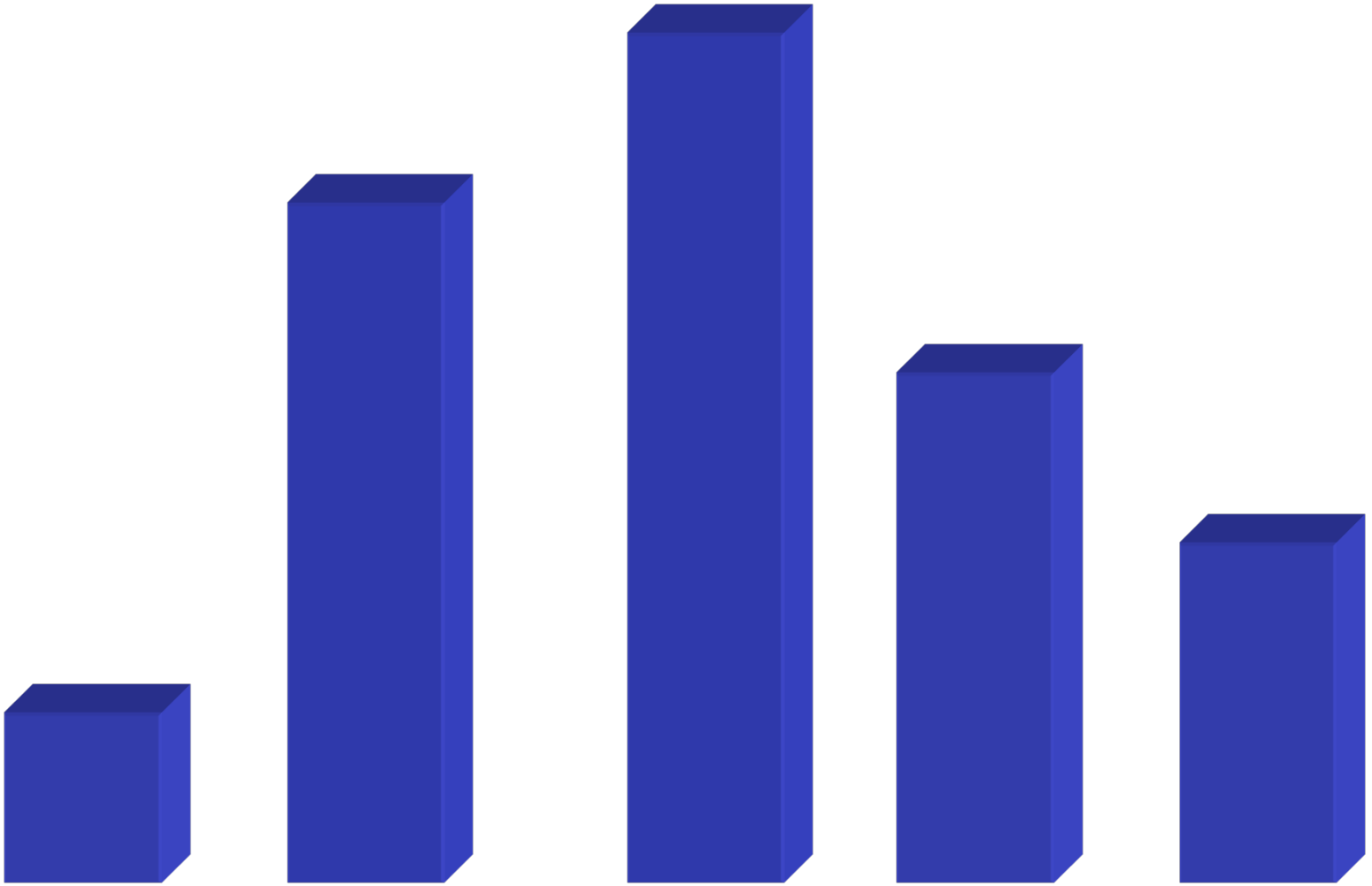
- http://research.compaq.com/SRC/JCAT/jdk10/sorting/
- http://math.baruch.cuny.edu/~bshaw/index-8.html
- http://www.cs.duke.edu/csed/jawaa2/examples/sort.html
- http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/sorting.html
- http://www.inf.ethz.ch/~staerk/algorithms/SortAnimation.html
- http://www.cs.hope.edu/~alganim/ccaa/sorting.html

Algorithm Design Technique
Brute Force

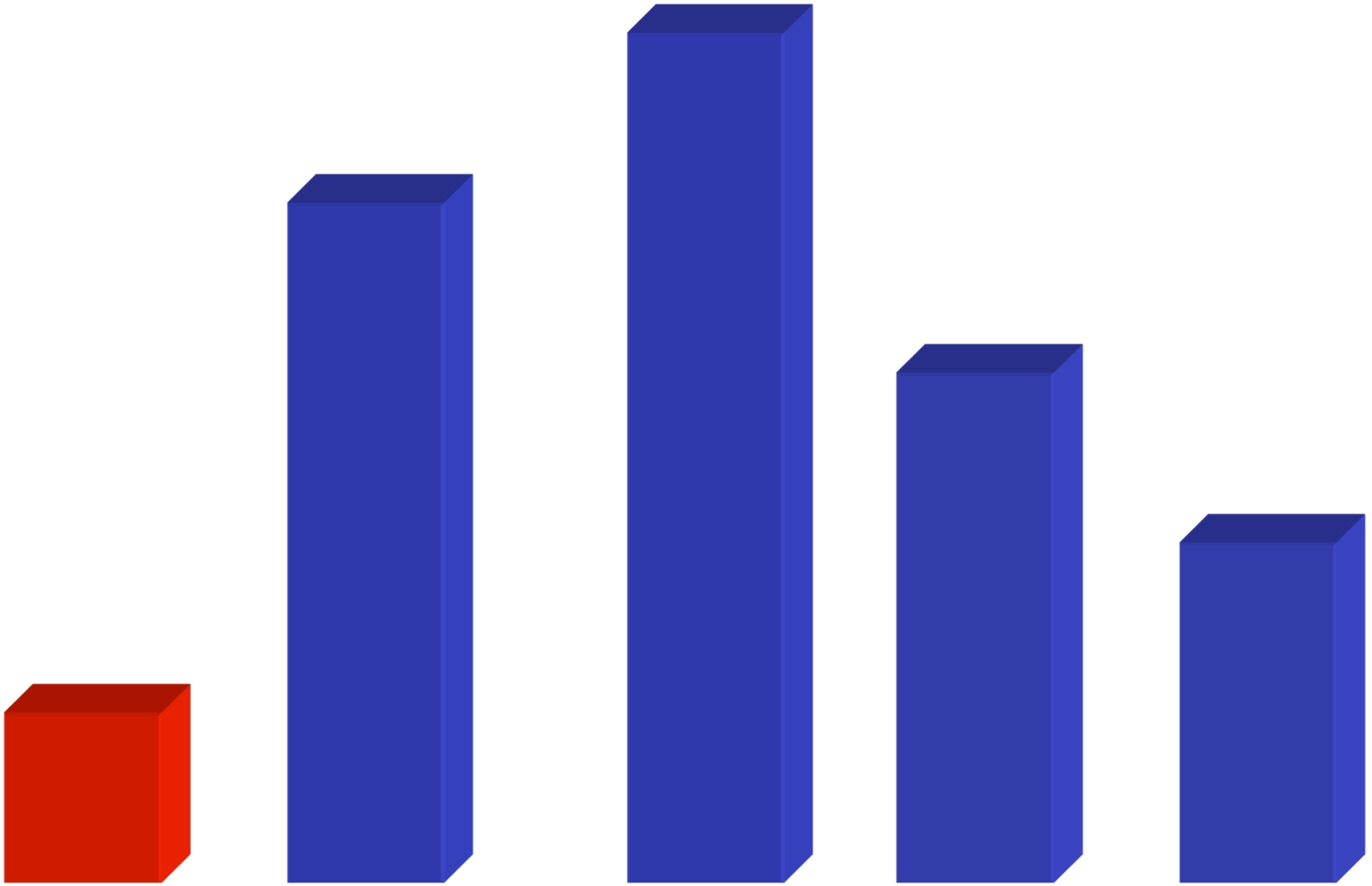
- **Brute force** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

- Simplest algorithm design technique
- Does not usually produce the most elegant or most efficient algorithm
- Examples of the Brute Force technique
  - Selection sort
  - Bubble sort
  - Sequential search
  - Exhaustive search

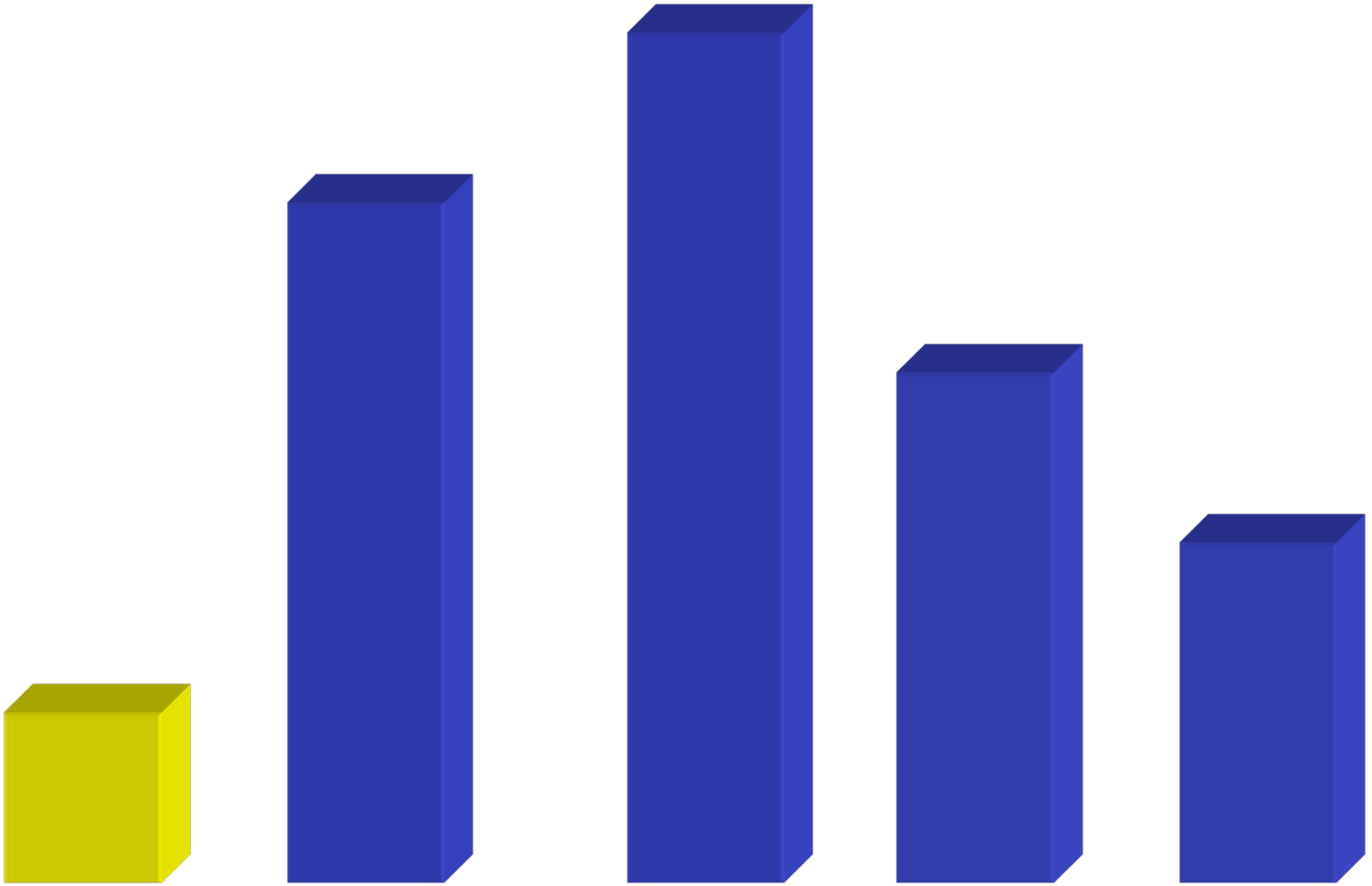
Selection Sort Animation



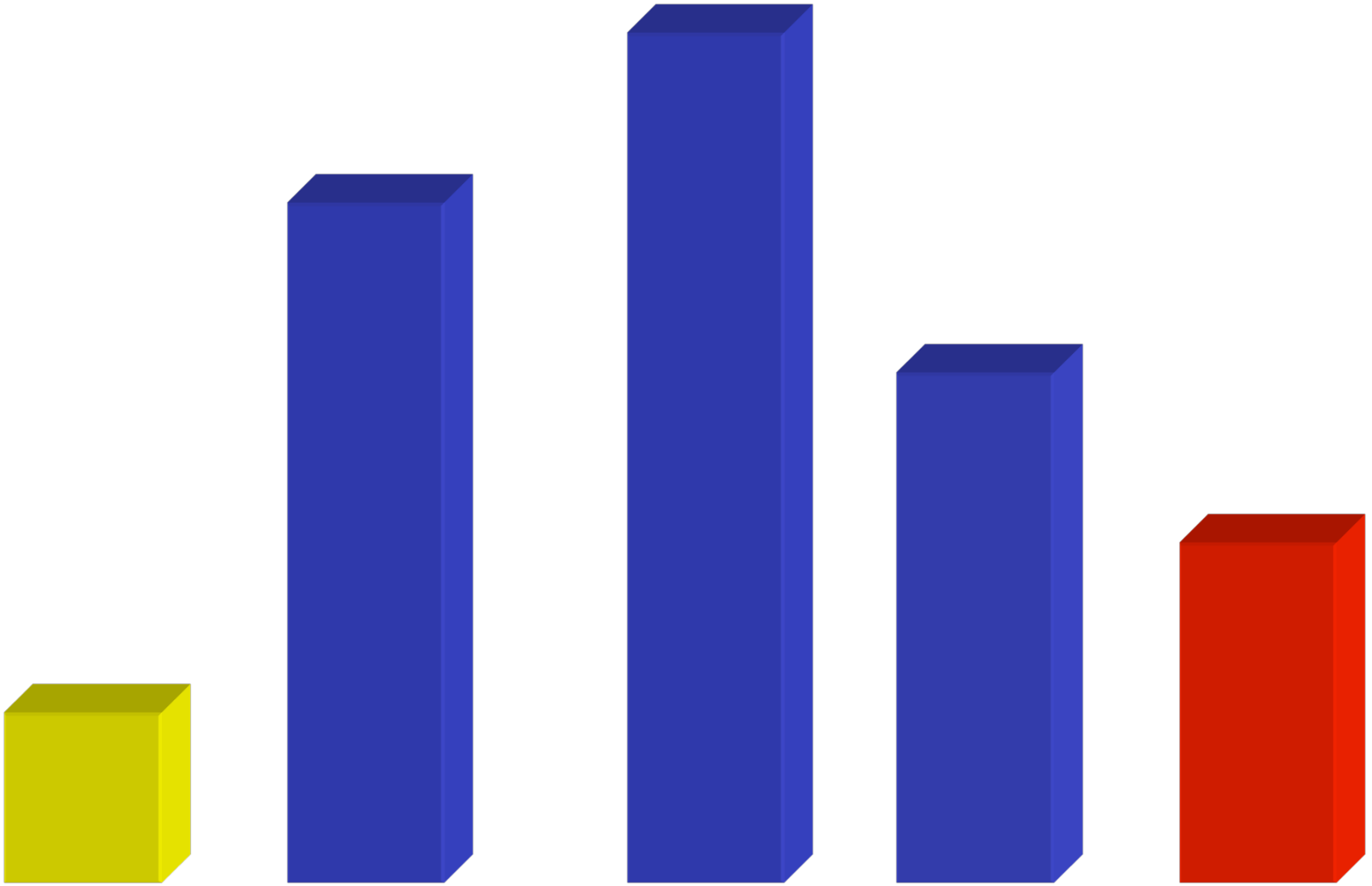
Selection Sort Animation



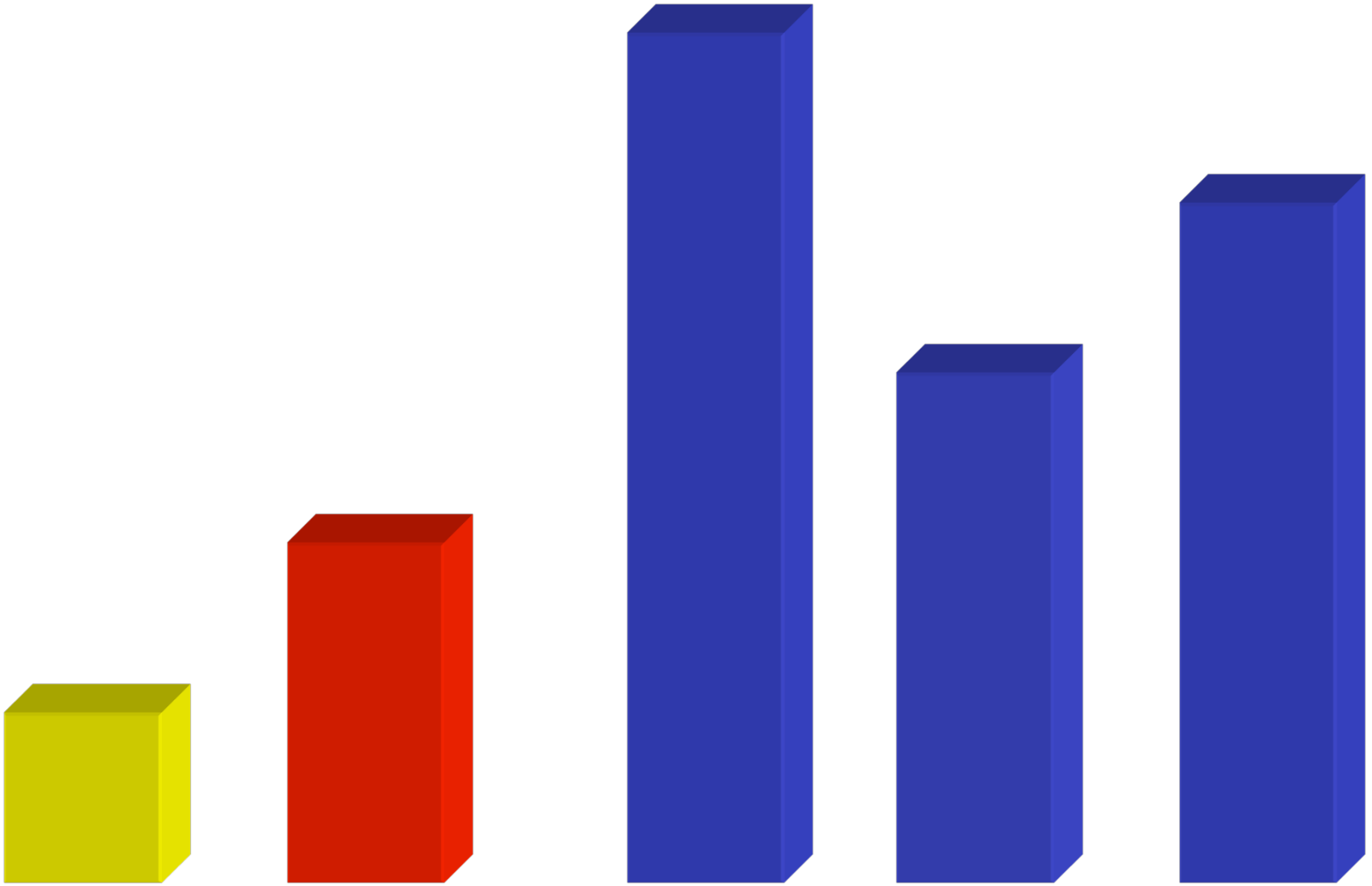
Selection Sort Animation



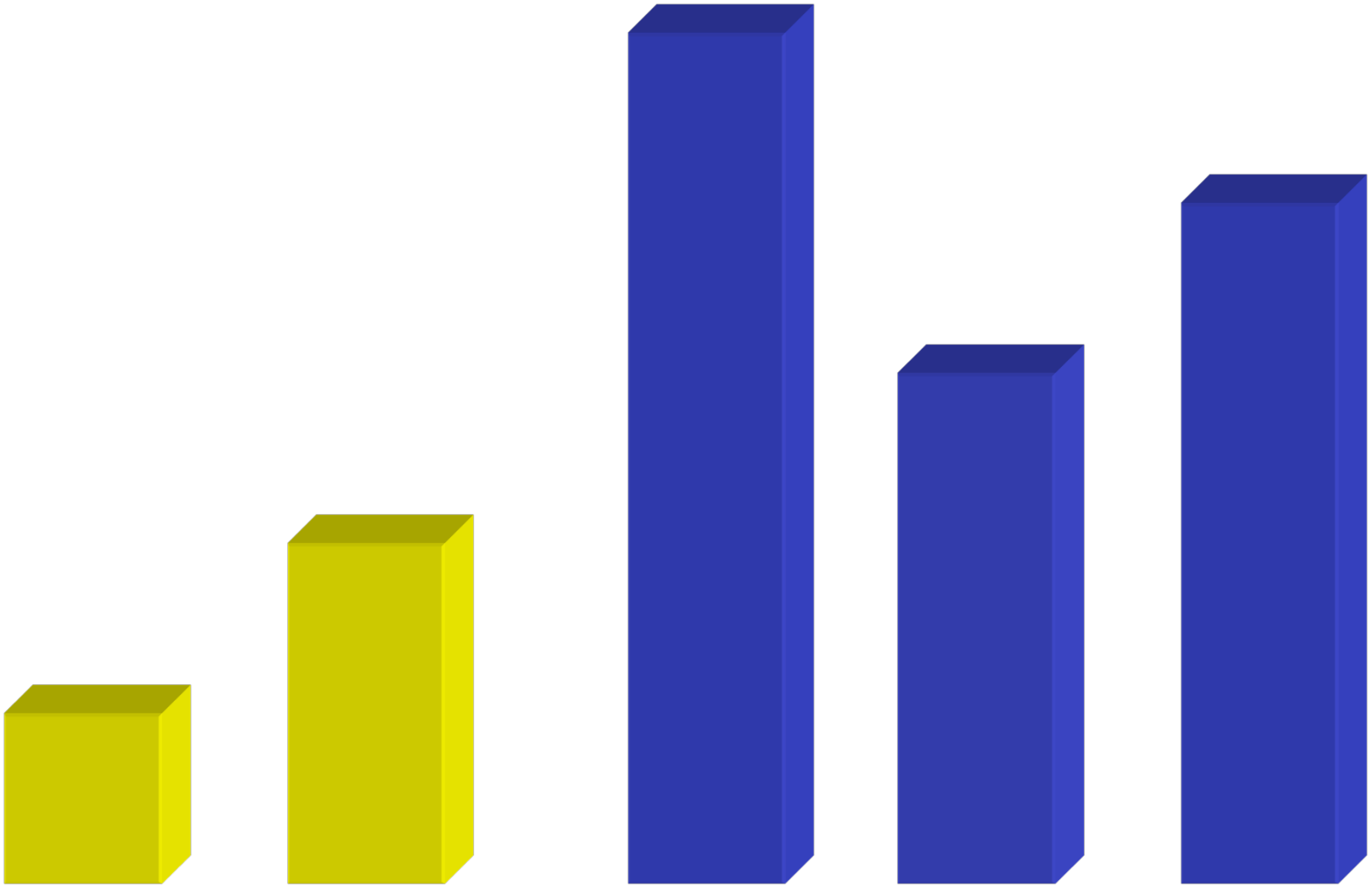
Selection Sort Animation



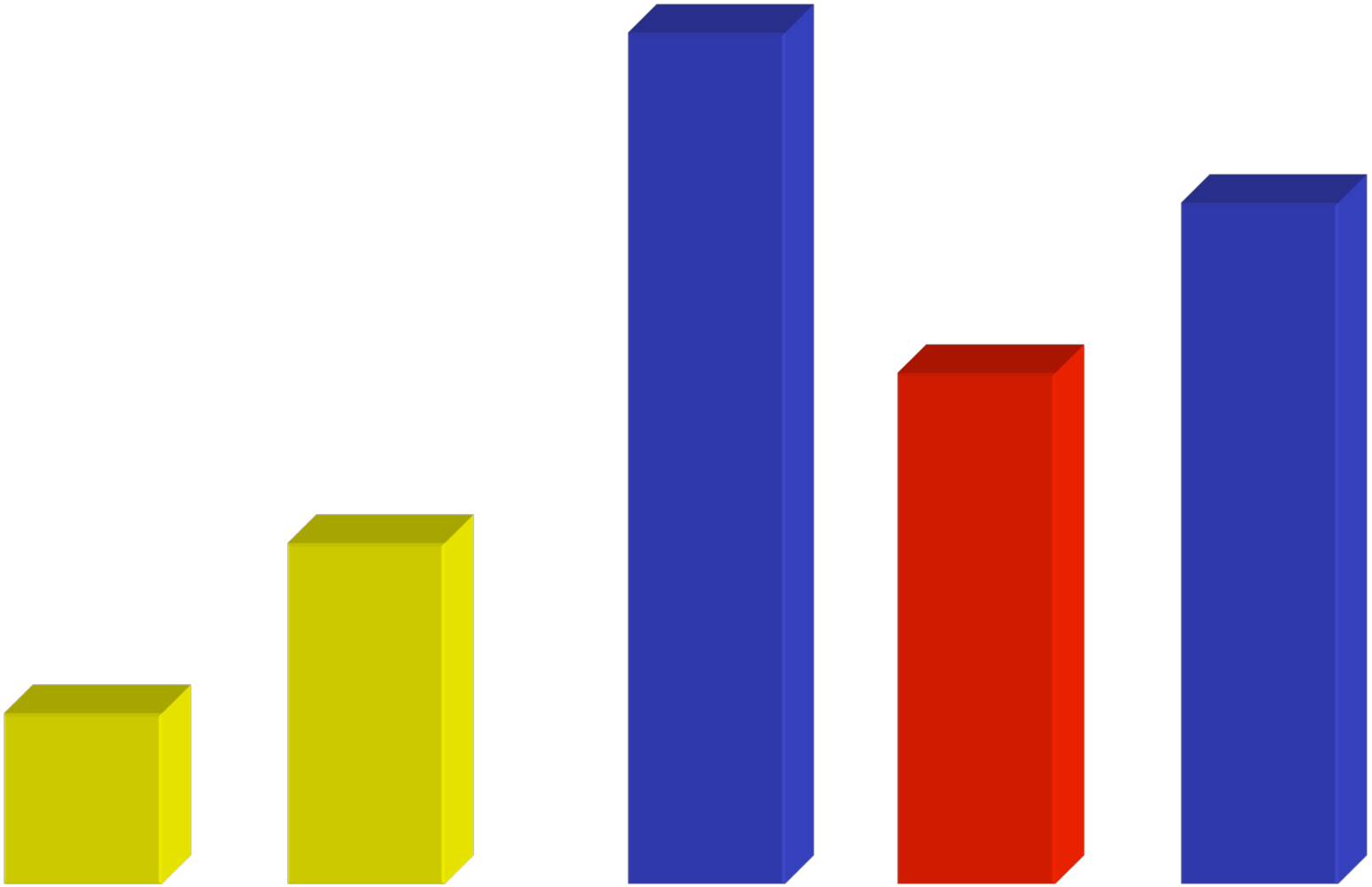
Selection Sort Animation



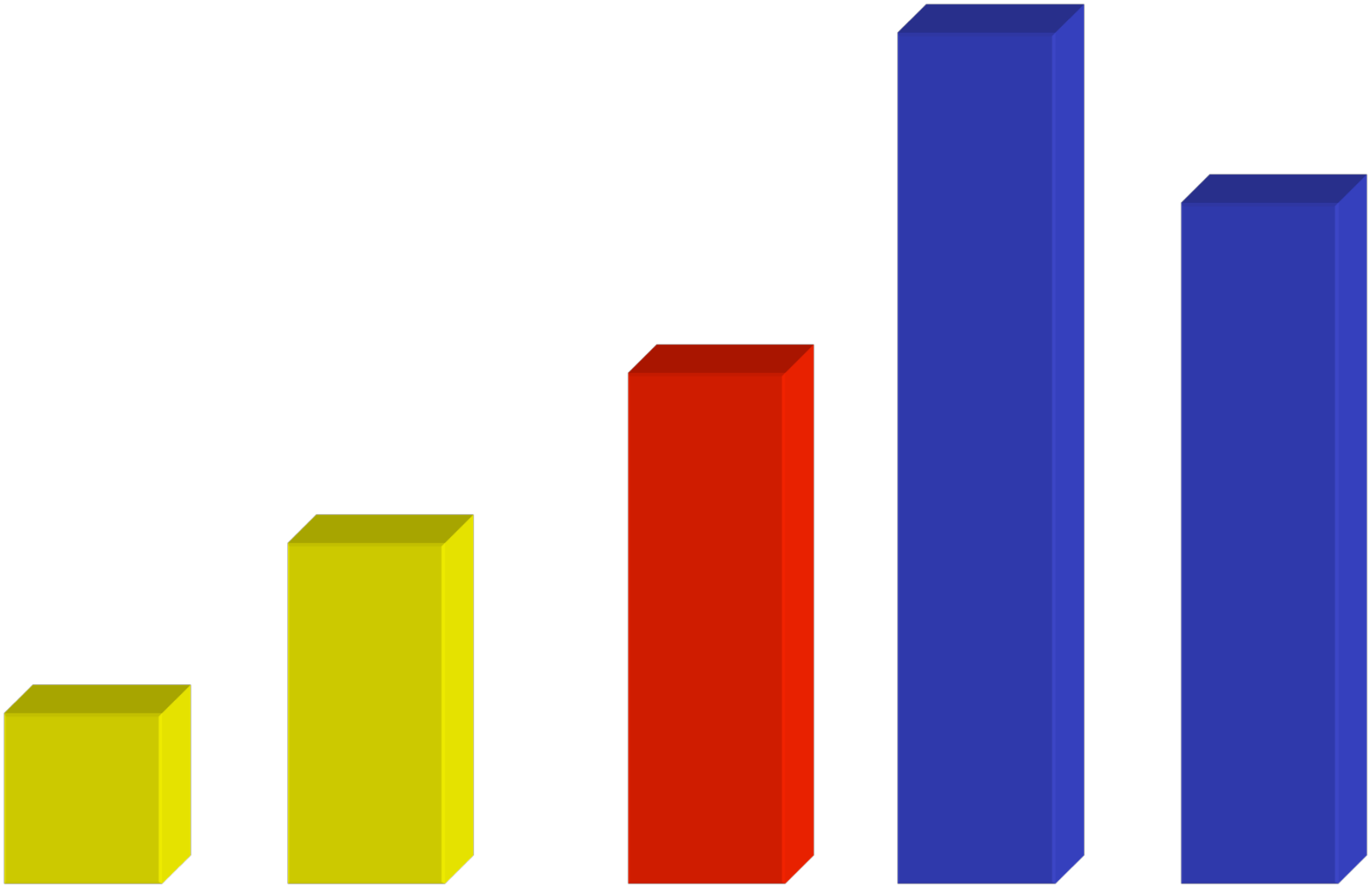
Selection Sort Animation



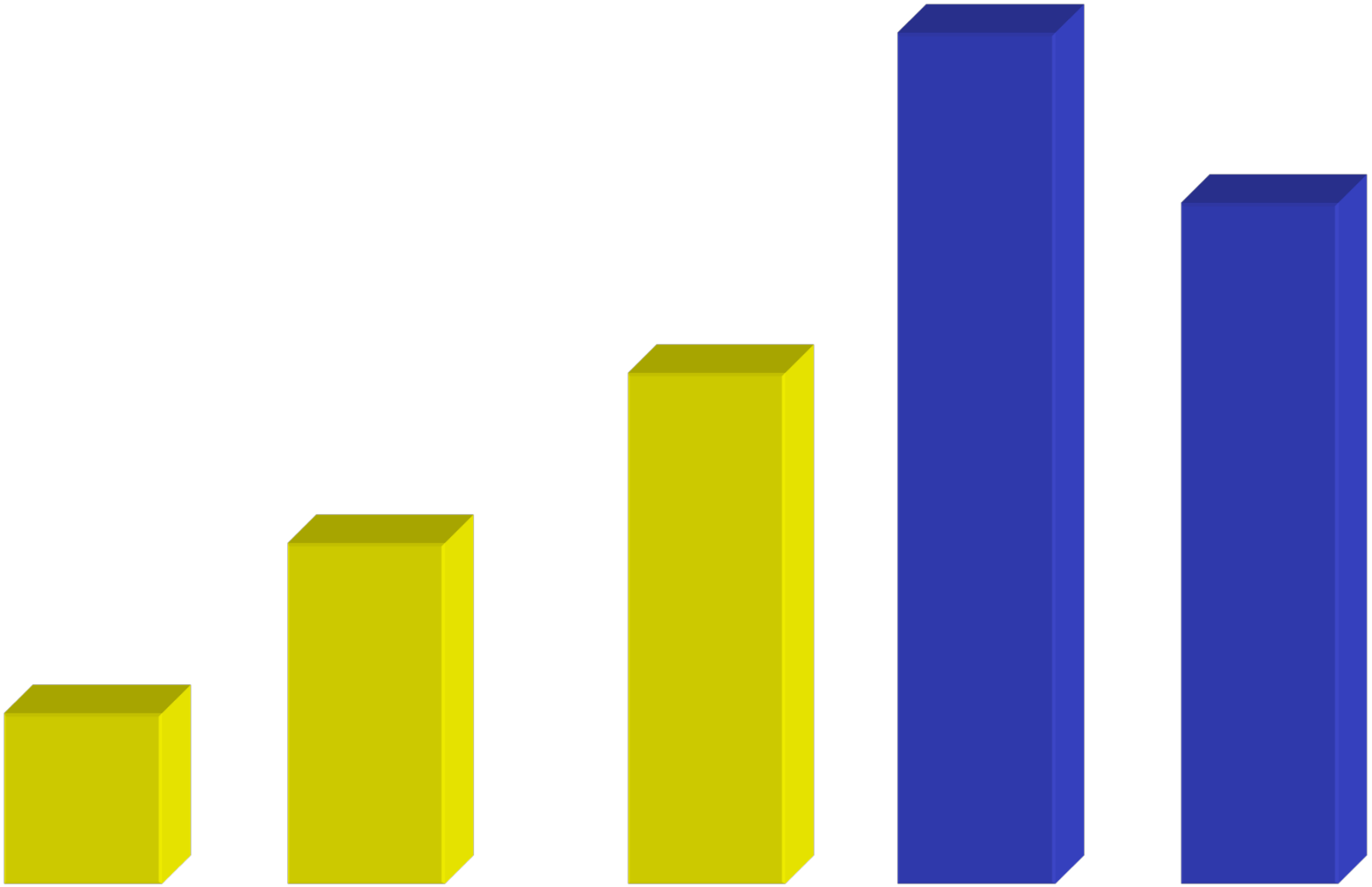
Selection Sort Animation



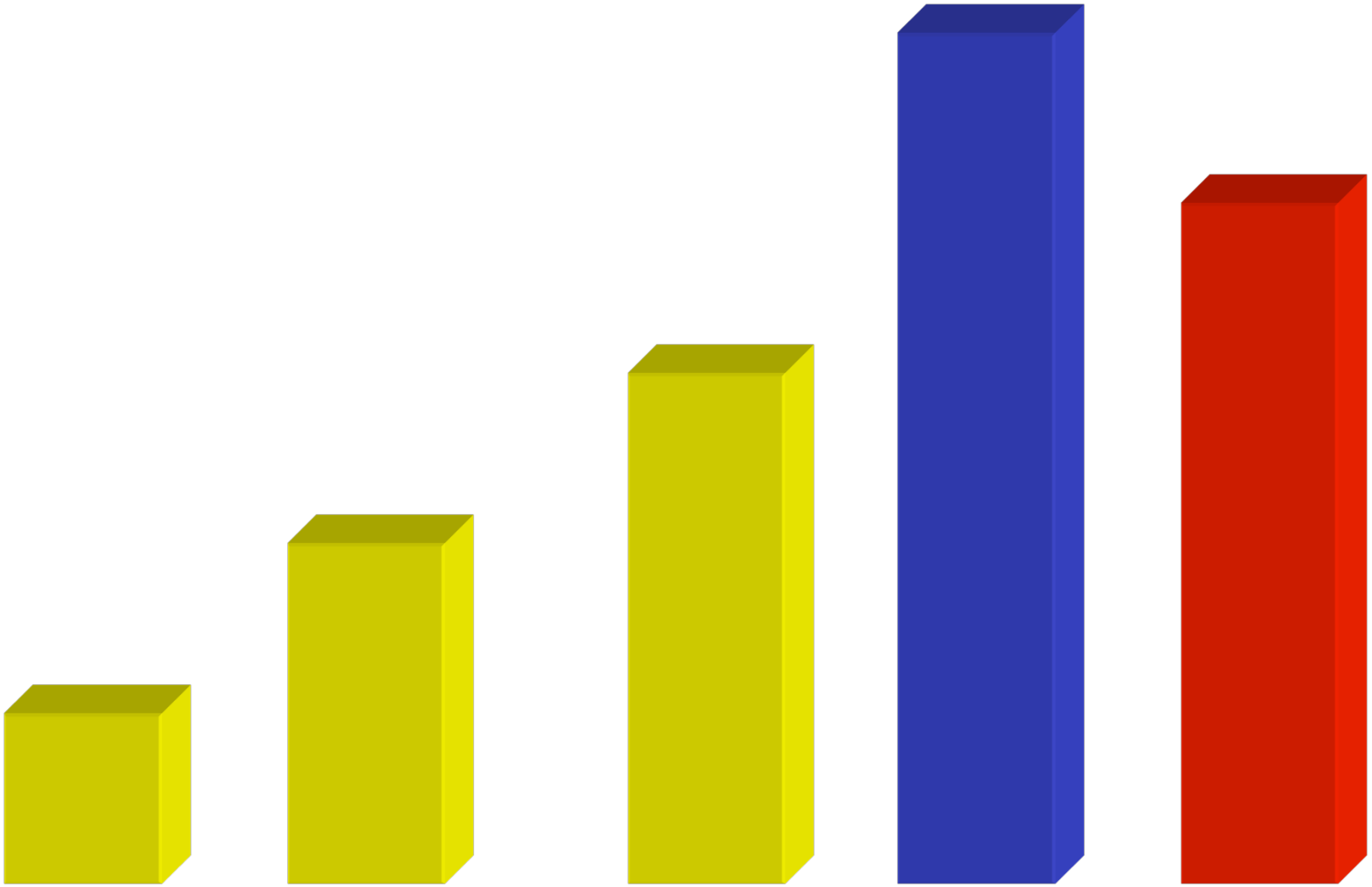
Selection Sort Animation



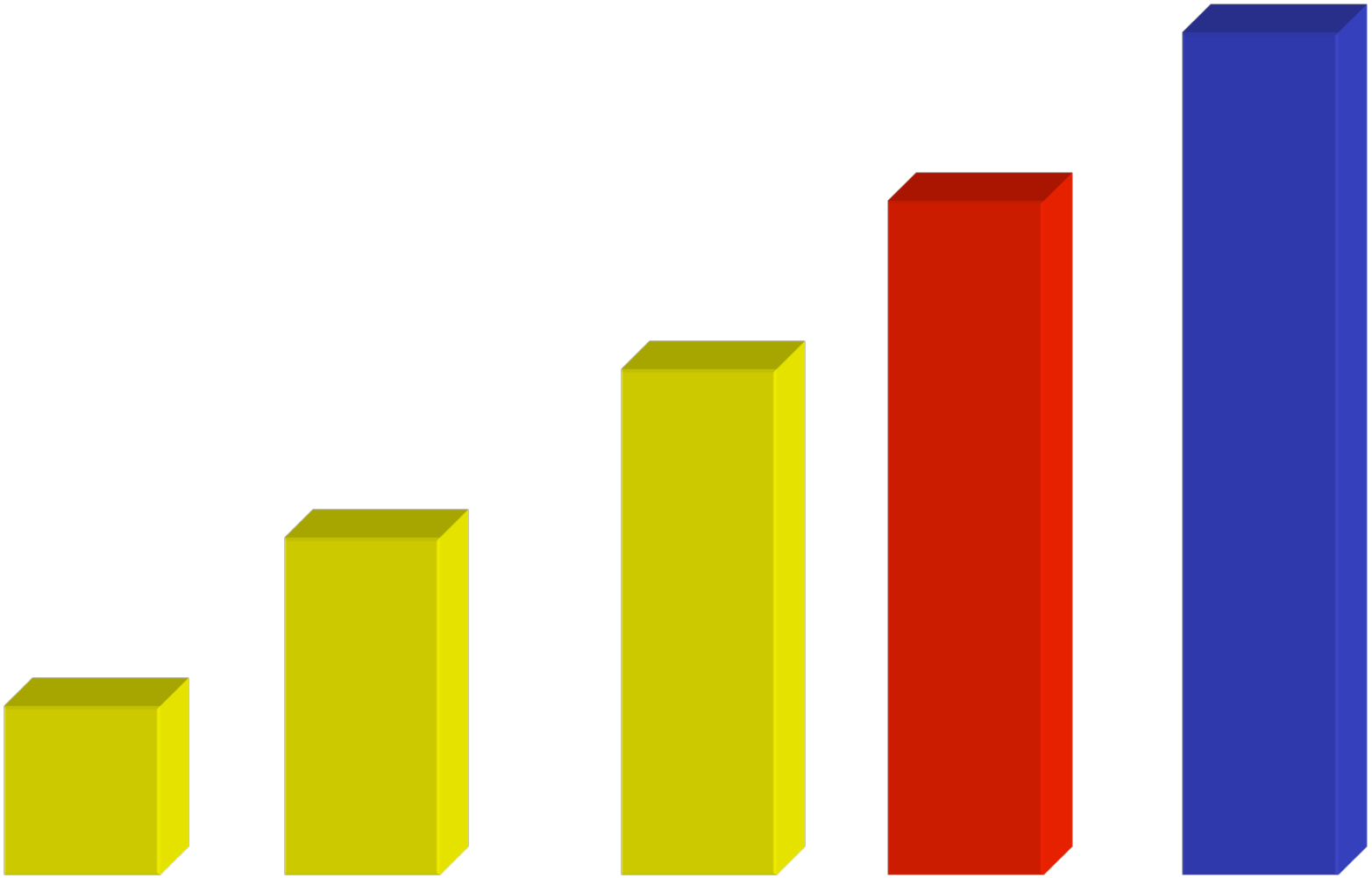
Selection Sort Animation



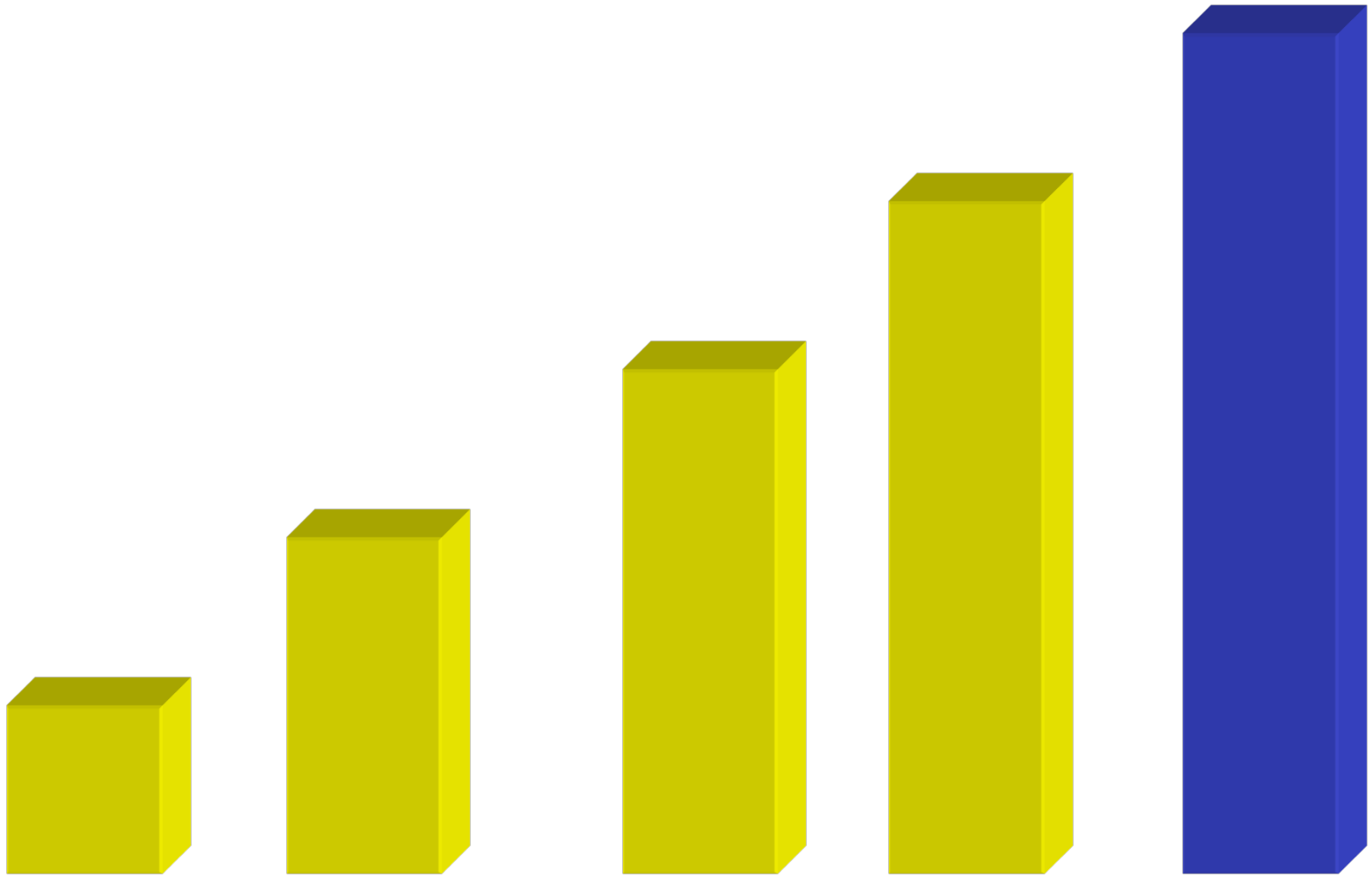
Selection Sort Animation



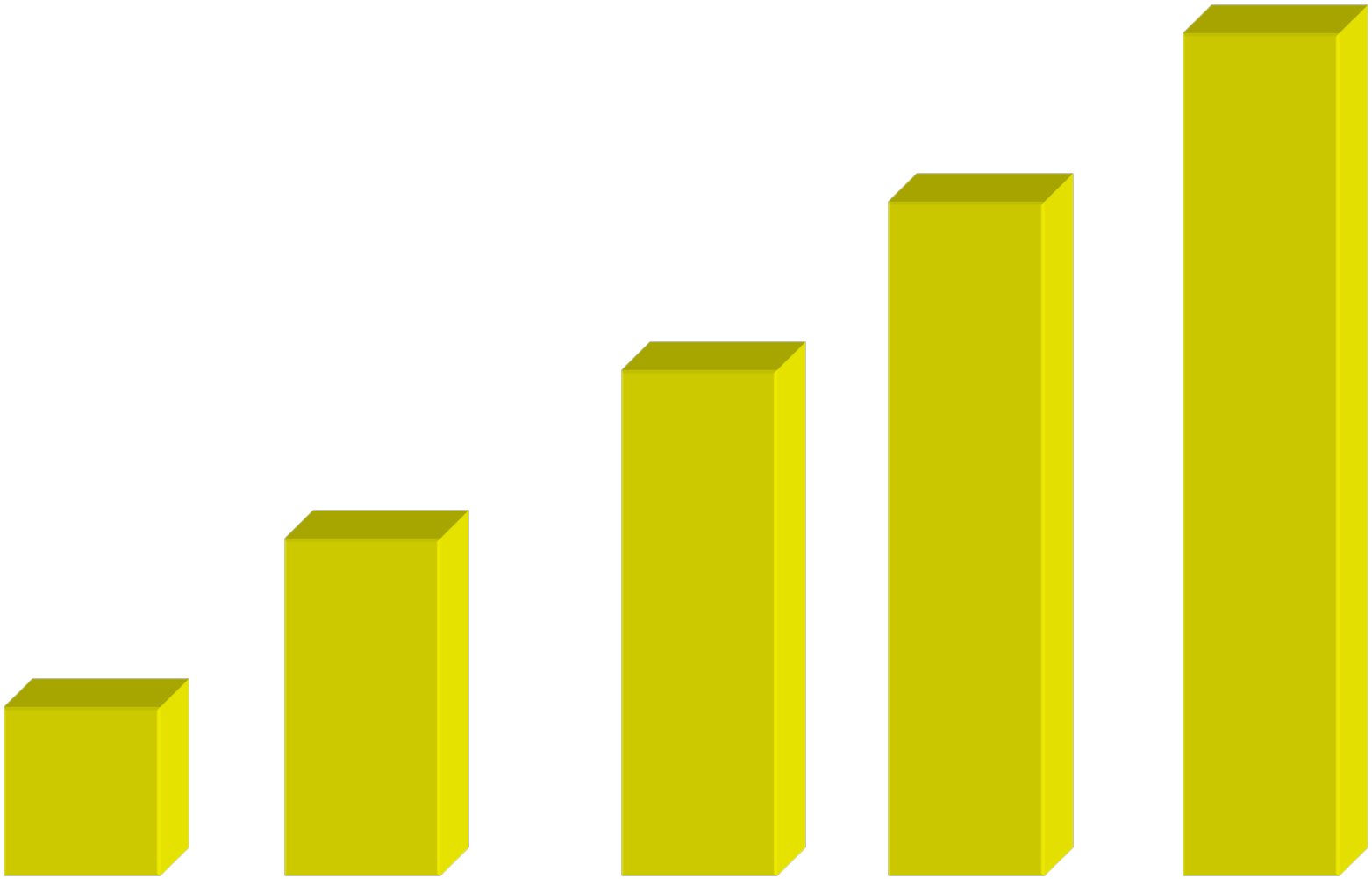
Selection Sort Animation



Selection Sort Animation



Selection Sort Animation



What is the Worst-case Running Time of Selection Sort?

for $k \leftarrow 0$ **to** $n-2$ **do**
    Find the smallest item $A[j]$ between $A[k]$ and $A[n-1]$
    swap $A[k]$ and $A[j]$
end

for $k \leftarrow 0$ **to** $n-2$ **do**
    min $\leftarrow$ A[*k*]
    for $j \leftarrow k+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ **then** $min \leftarrow$ A[*j*]**end**
    end
    swap($A[k]$, $A[min]$)
end

What is the Worst-case Running Time of Selection Sort?

$$T(n) = \sum_{k=0}^{n-2} \sum_{j=k+1}^{n-1} \text{cmps} = \sum_{k=0}^{n-2} \sum_{j=k+1}^{n-1} 1 = \sum_{k=0}^{n-2} ((n-1)-(k+1)+1)$$

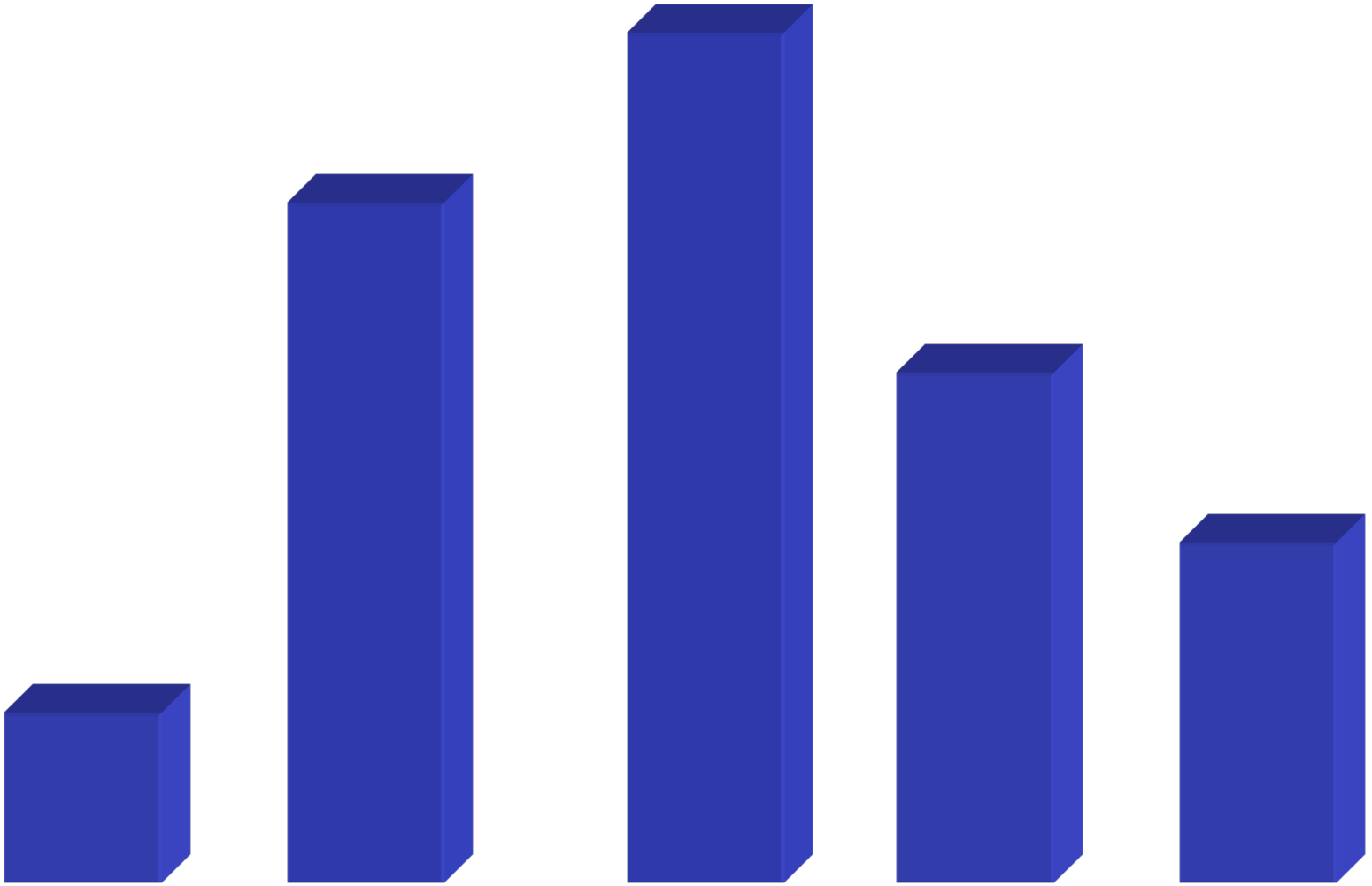
$$T(n) = \sum_{k=0}^{n-2} (n-1-k) = \sum_{k=0}^{n-2} n - \sum_{k=0}^{n-2} 1 - \sum_{k=0}^{n-2} k$$

$$T(n) = (n-1)n - (n-1) - \frac{(n-2)(n-1)}{2}$$

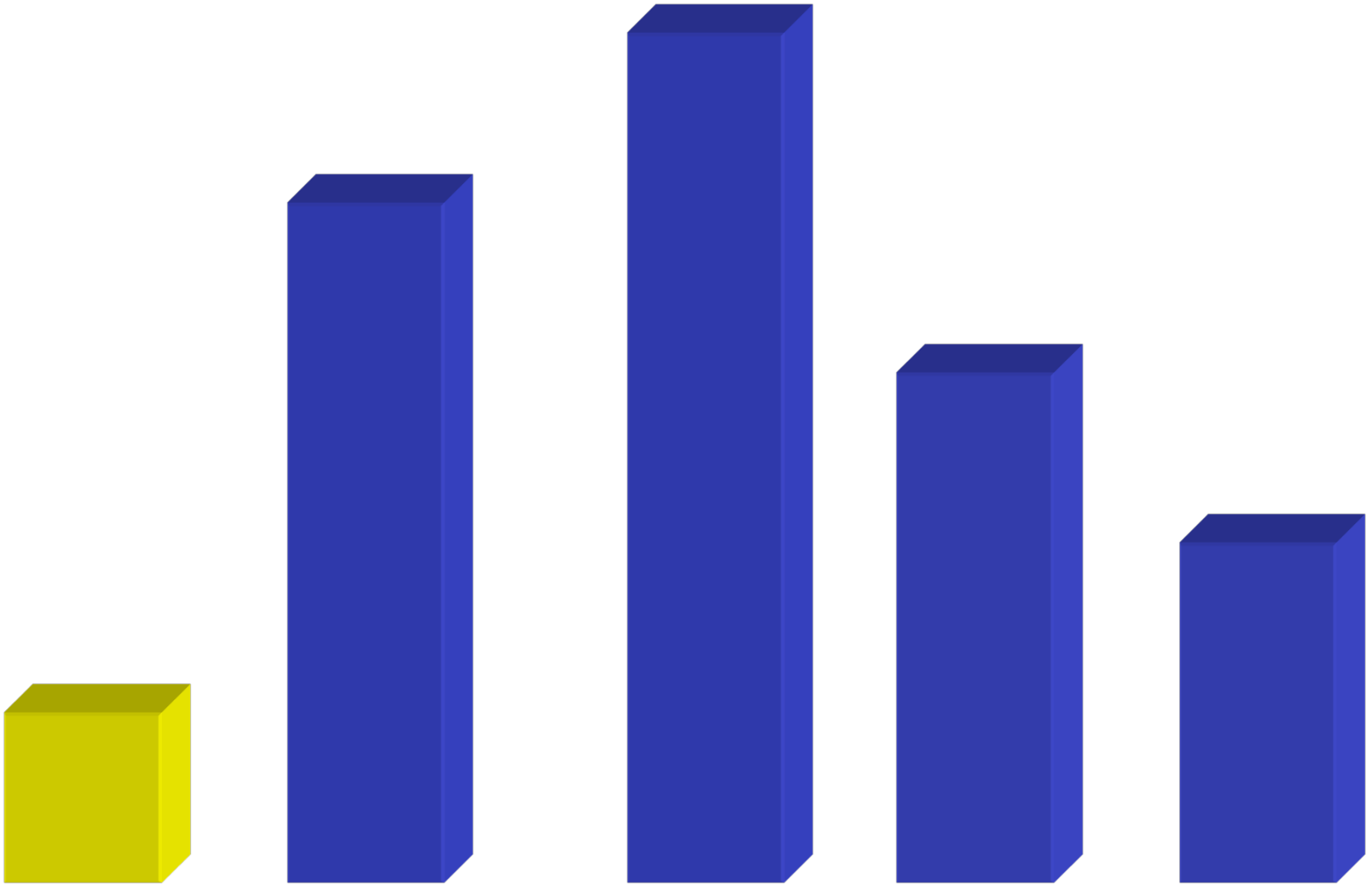
$$T(n) = \frac{2(n-1)n - 2(n-1) - (n-2)(n-1)}{2}$$

$$T(n) = \frac{(n-1)(2n-2-n+2)}{2} = \frac{(n-1)n}{2} \in O(n^2)$$

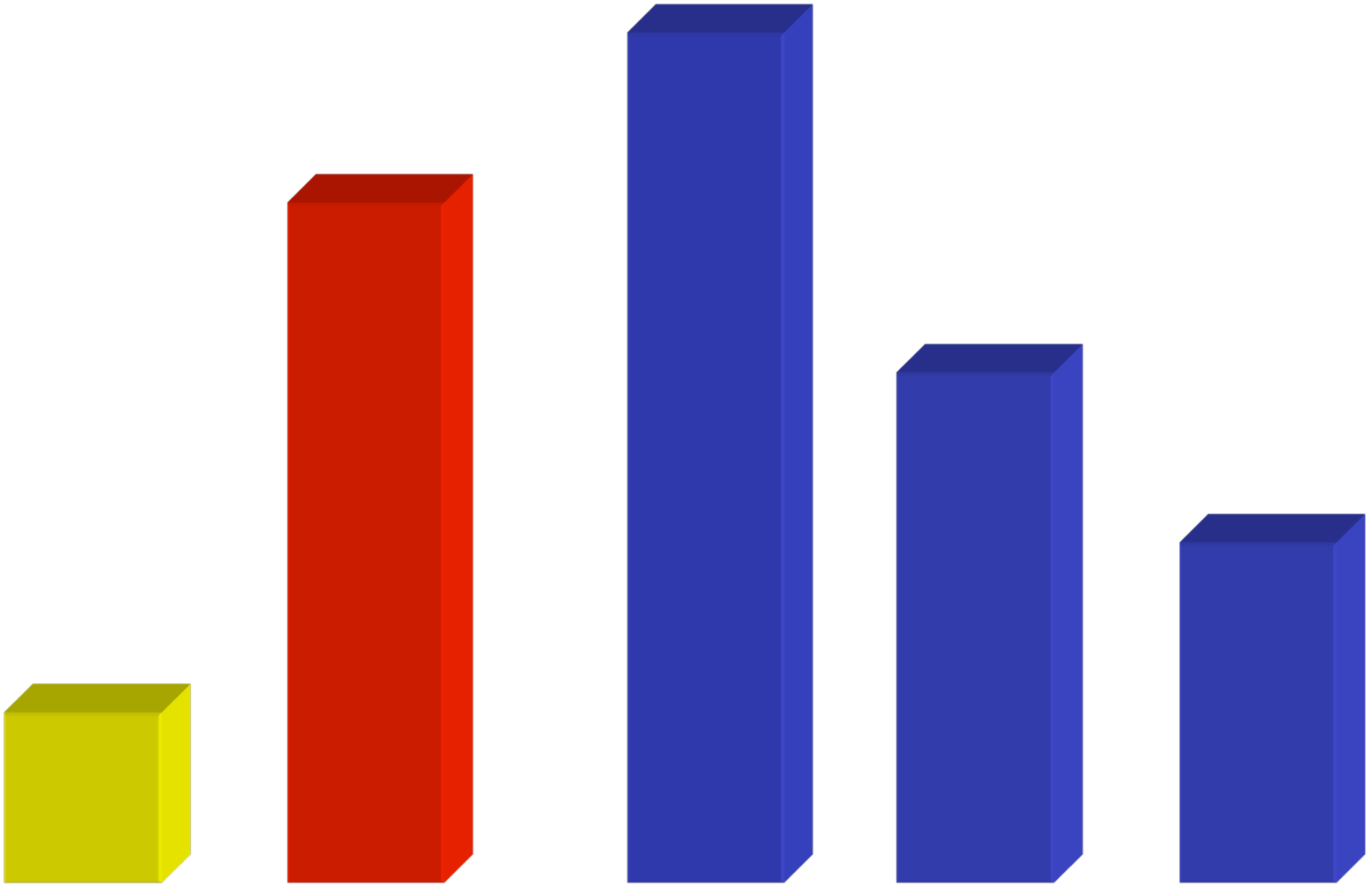
Insertion Sort



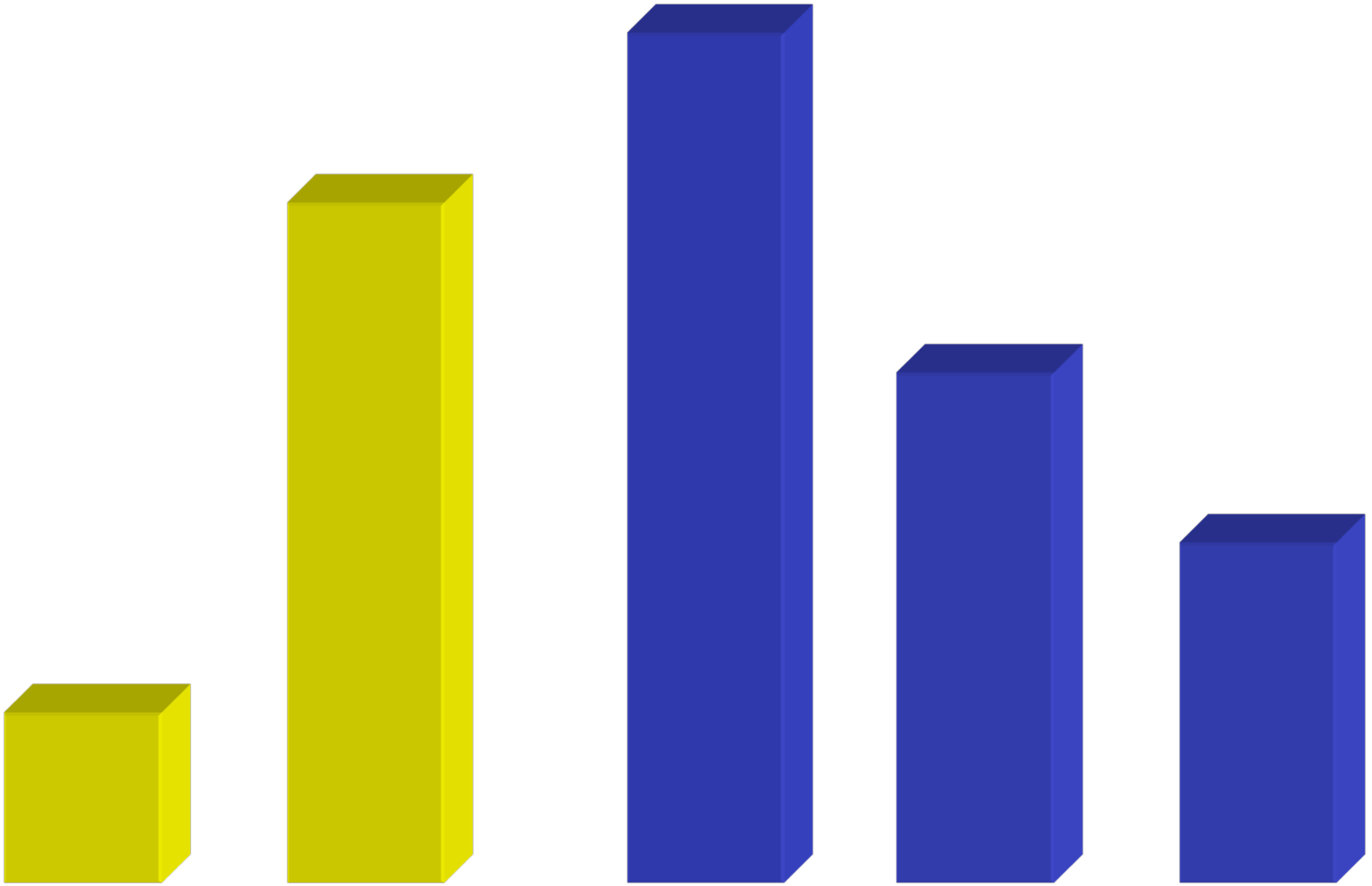
Insertion Sort



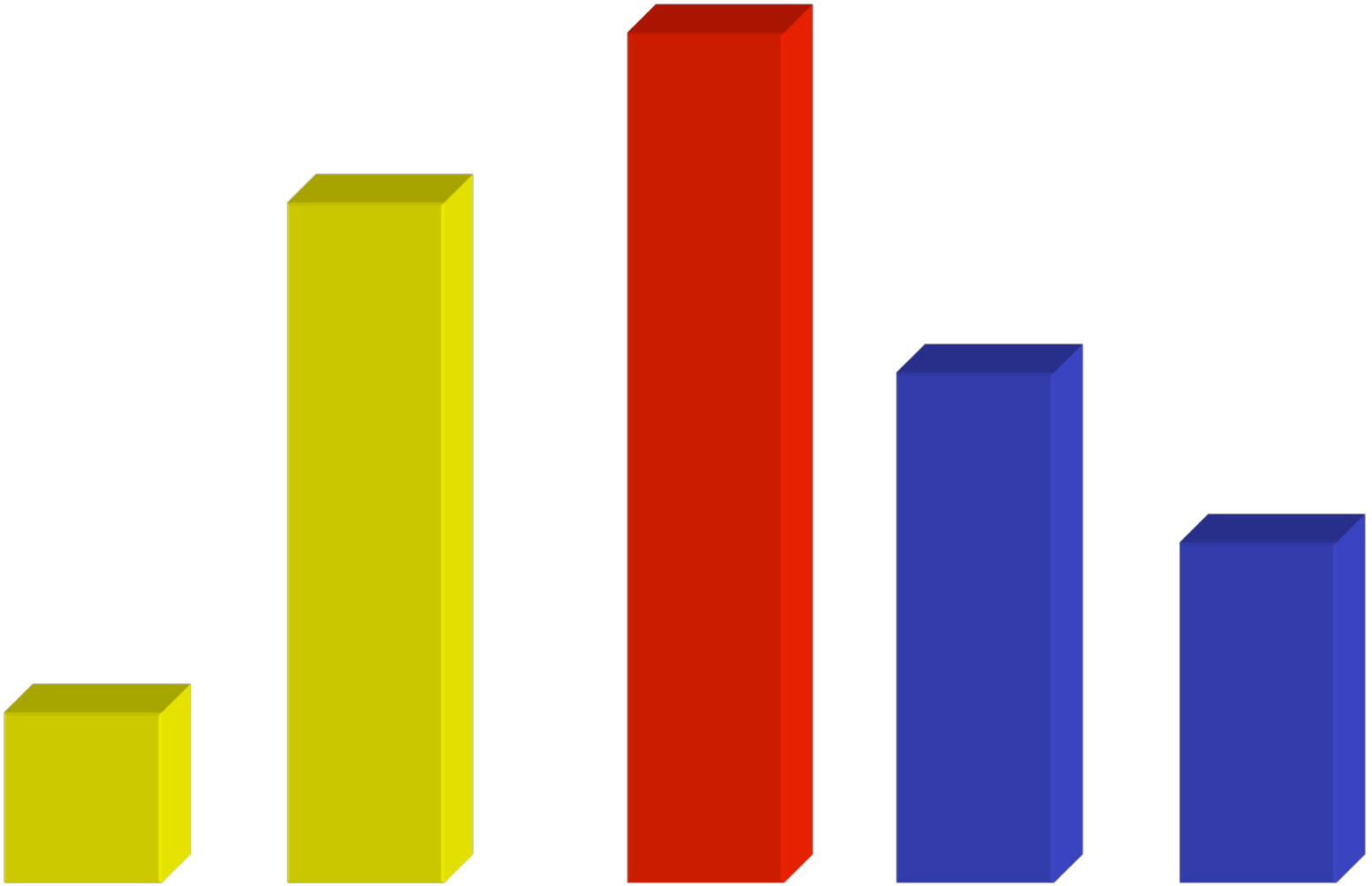
Insertion Sort



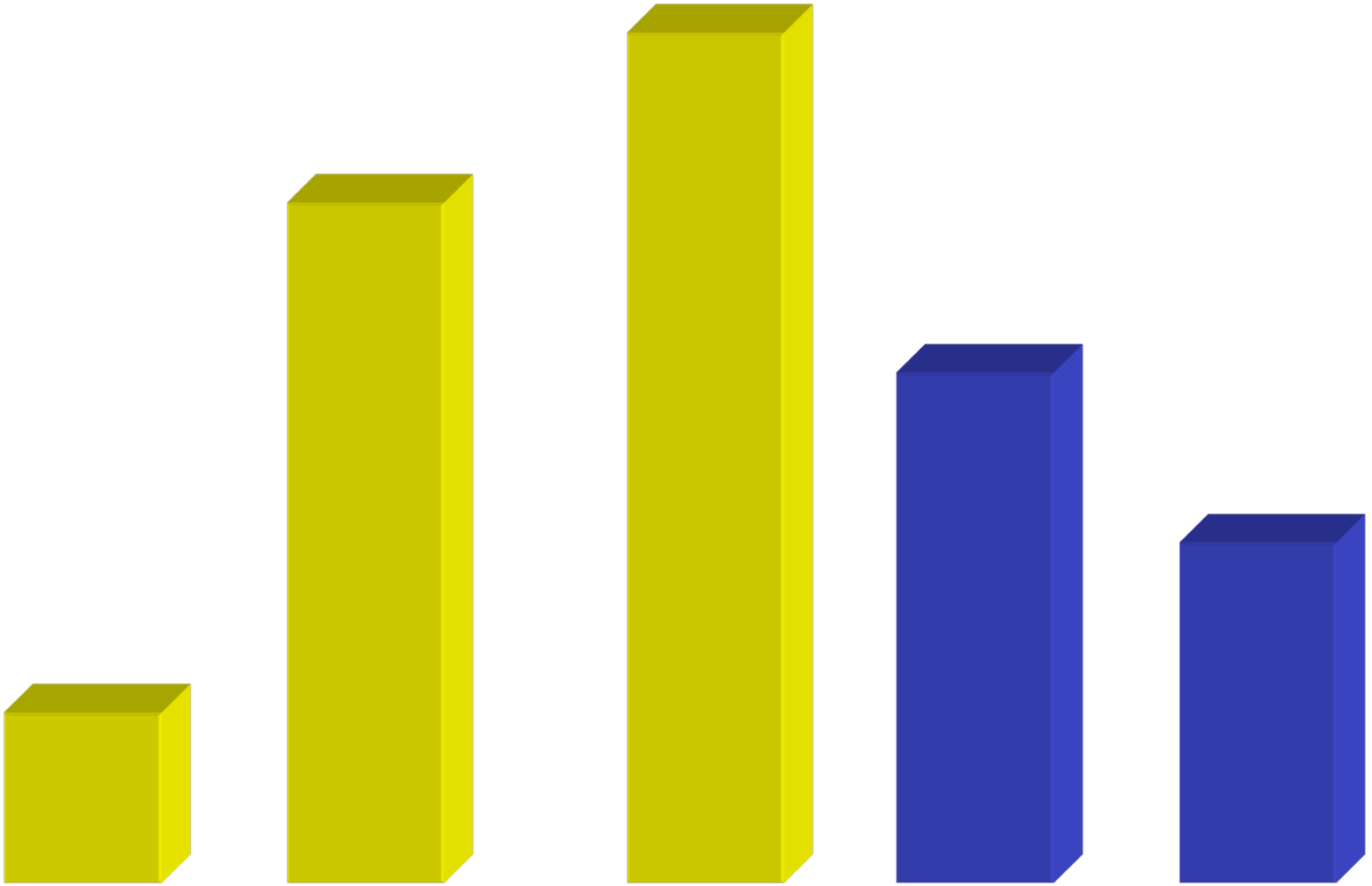
Insertion Sort



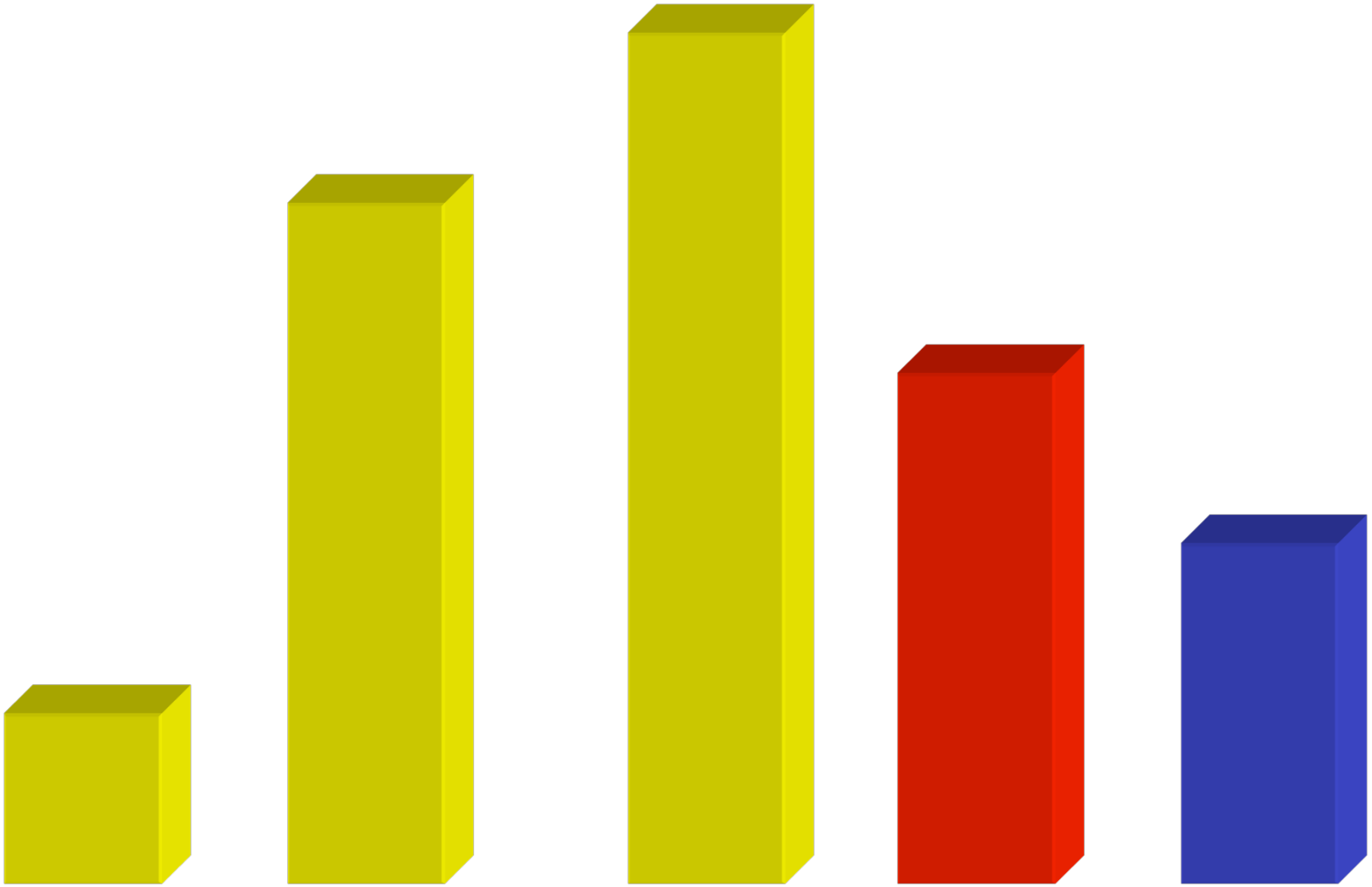
Insertion Sort



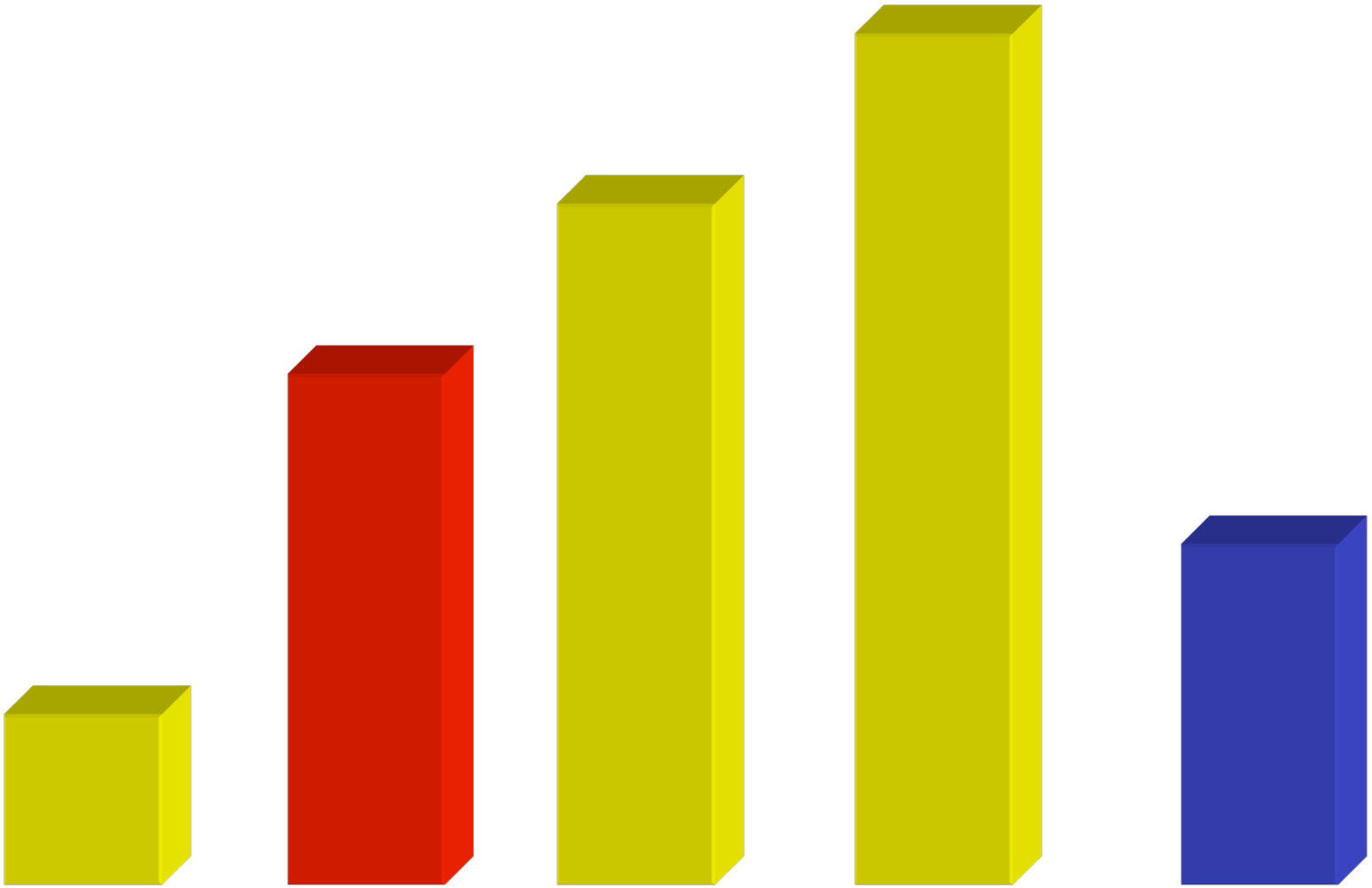
Insertion Sort



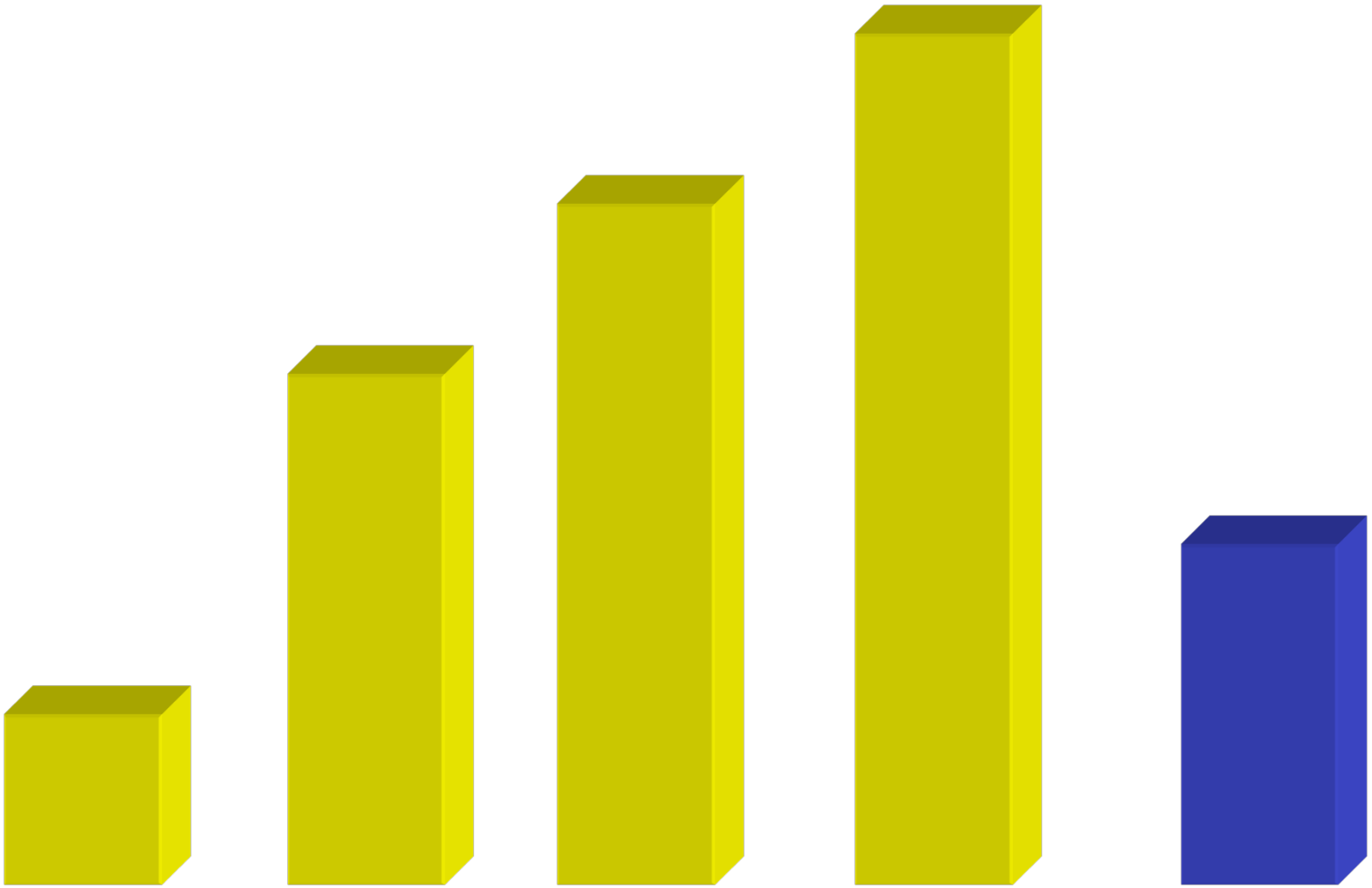
Insertion Sort



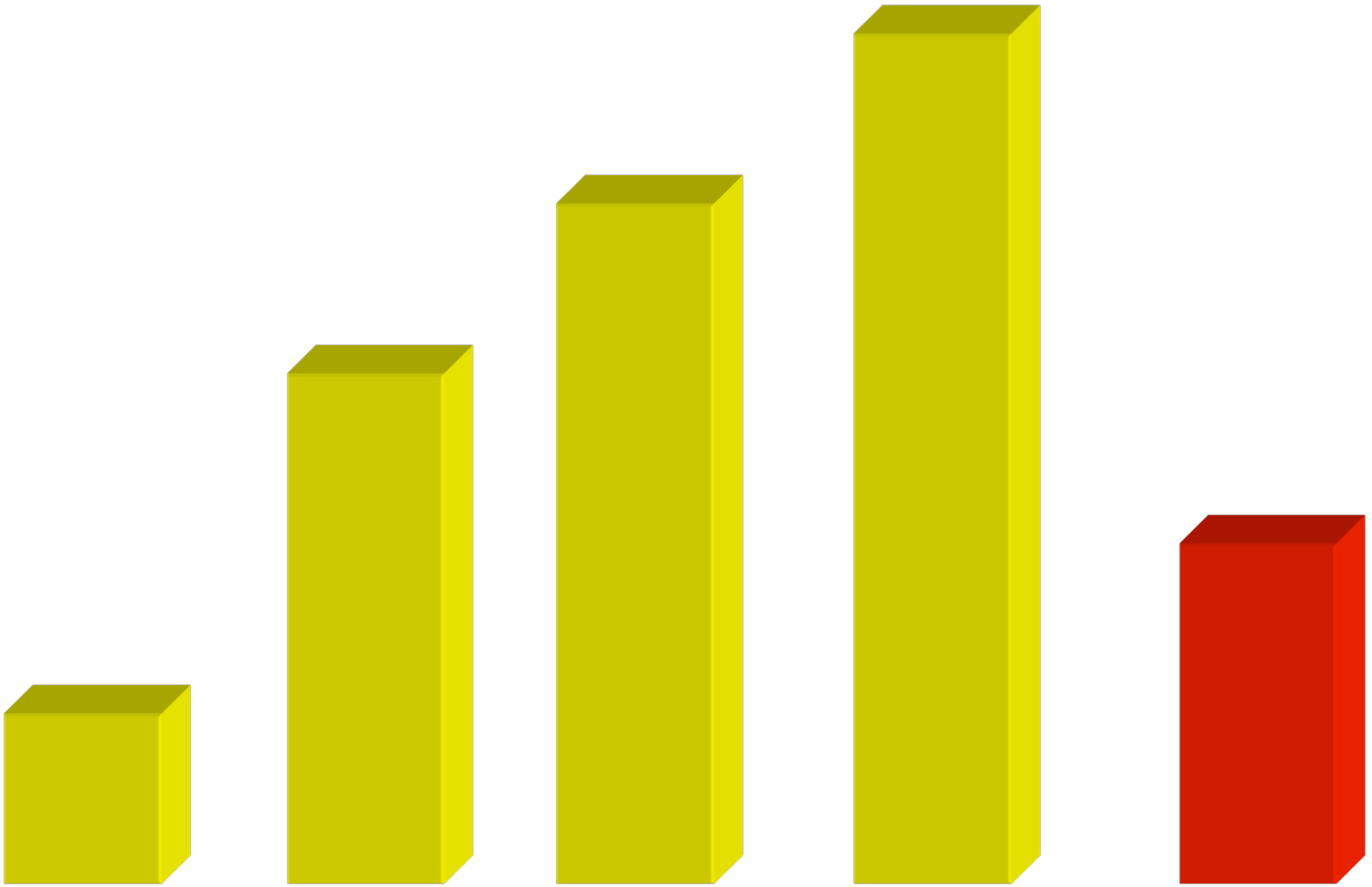
Insertion Sort



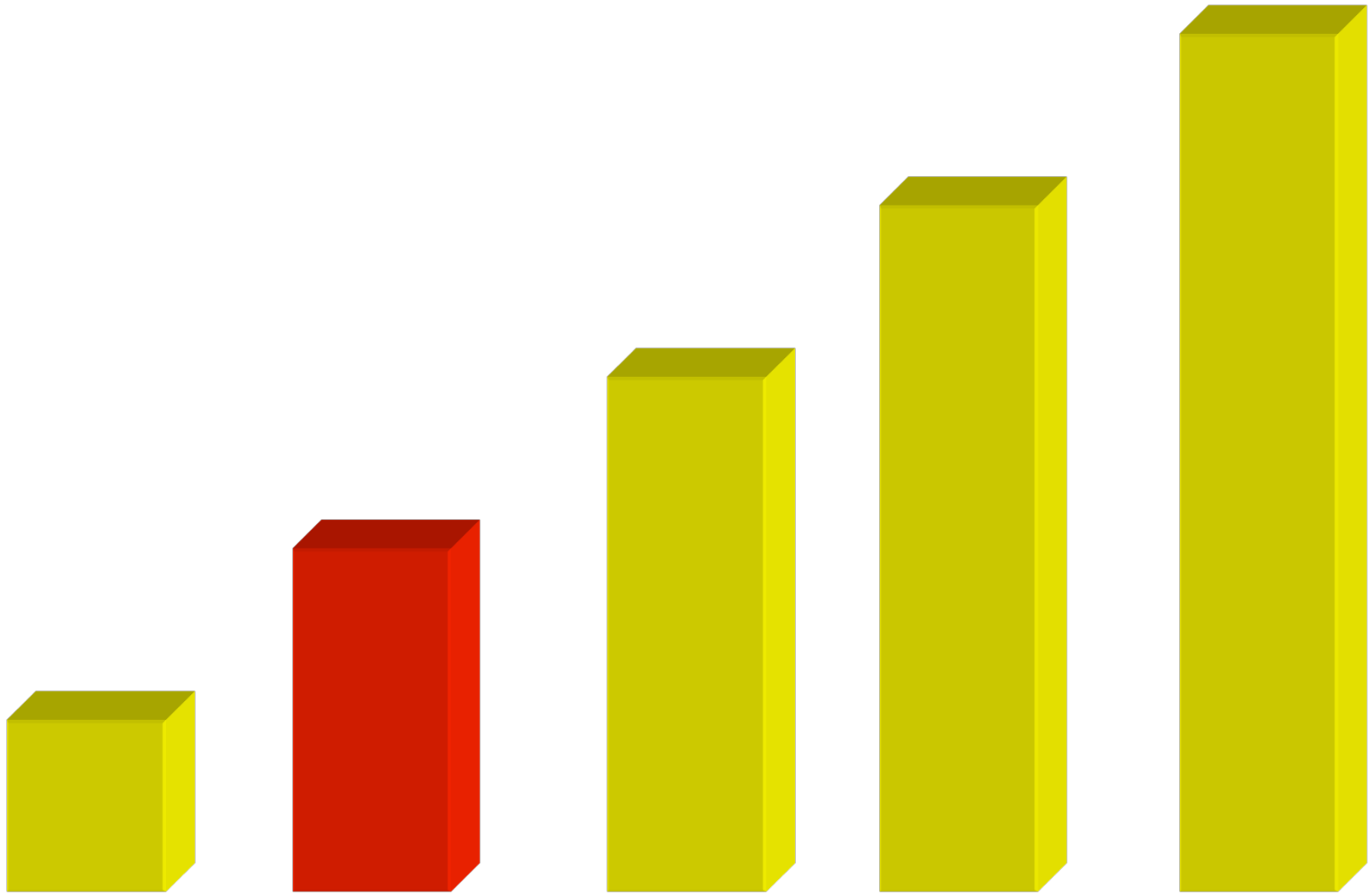
Insertion Sort



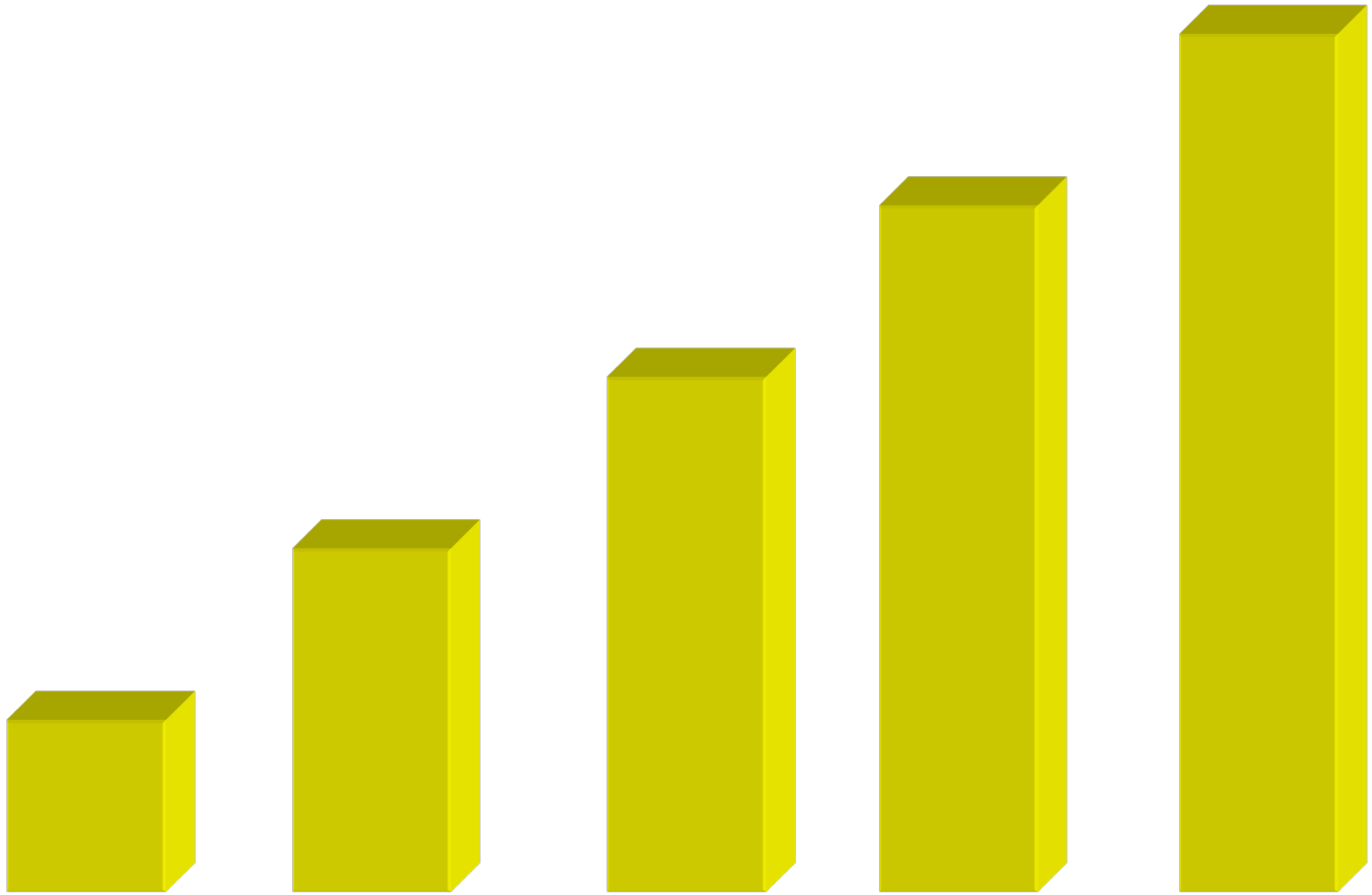
Insertion Sort



Insertion Sort



Insertion Sort



What is the Worst-case Running Time of Insertion Sort?

for $k \leftarrow 2$ **to** n **do**
    Move $A[k]$ forward to position j ≤ k such that
    $A[k] < A[p]$ for $j \leq t < k$ and
    either $A[k] \geq A[j-1]$ or $j = 1$
end

for $k \leftarrow 1$ **to** $n-1$ **do**
    val $\leftarrow A[k]$ $j = k-1$
    while $j \geq 0$ **and** $A[j] >$ val **do**
        $A[j+1] \leftarrow A[j]$ $j = j - 1$
    end
    $A[j+1]$ = val
end

What is the Worst-case Running Time of Insertion Sort?

$$T(n) = \sum_{k=1}^{n-1} \sum_{j=0}^{k-1} cmps = \sum_{k=1}^{n-1} \sum_{j=0}^{k-1} 1 =$$

$$T(n) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} \in O(n^2)$$