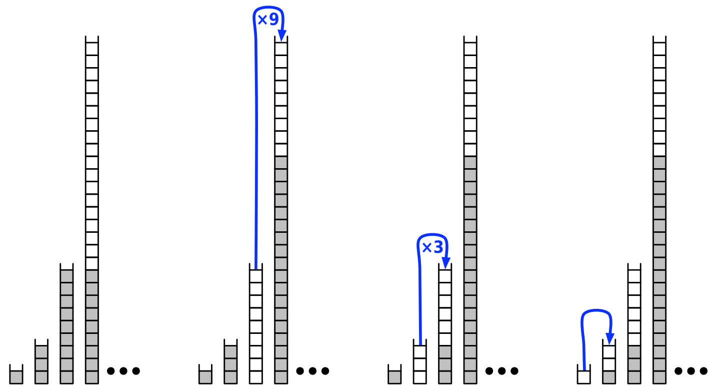
1. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2}.) Moving a single element from one stack to another takes $O(1)$ time.

   Here is pseudocode for the multistack operations MSPush and MSPop. The internal stacks are managed with the subroutines Push and Pop.

```
MPush(x):
  $i \leftarrow 0$
  while $S_i$ is full
      $i \leftarrow i+1$
  while $i > 0$
      $i \leftarrow i-1$
      for $j \leftarrow 1$ to $3^i$
          Push($S_{i+1}$, Pop($S_i$))
  Push($S_0$, $x$)
```

```
MPop(x):
  $i \leftarrow 0$
  while $S_i$ is empty
      $i \leftarrow i+1$
  while $i > 0$
      $i \leftarrow i-1$
      for $j \leftarrow 1$ to $3^i$
          Push($S_i$, Pop($S_{i+1}$))
  return Pop($S_0$)
```



Making room in a multistack, just before pushing on a new element.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.

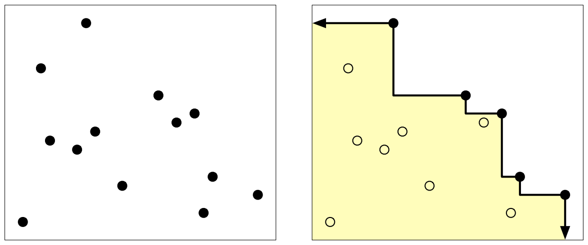
   (c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

   - CREATE( ) creates and returns a new list
   - PUSH(L, x) appends x to the end of L
   - POP(L) deletes the last entry of L and returns it
   - LOOKUP(L, k) returns the kth entry of L

   Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses *anything* fancier, you must give an explicit implementation. Your data structure must support all operations in amortized constant time. In addition, your data structure must support each LOOKUP in *worst-case* $O(1)$ time. At all times, the size of your data structure must be linear in the number of objects it stores.

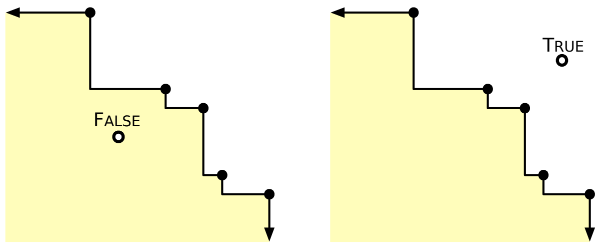
3. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

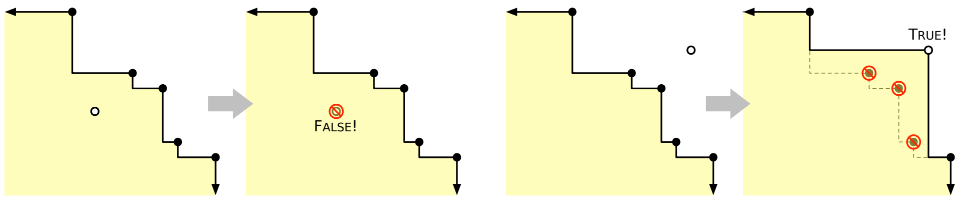
(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.



Two staircase queries.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function INSERT(x, y) that adds the point (x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.



Two staircase insertions.