CS61B Lecture 5

Friday, January 31, 2020

How to Write a Loop

- Try to give a description of how things look on any arbitrary iteration of the loop.
- This description is known as a loop invariant, because it is always true at the start of each iteration.
- The loop body then must:
    - Start from any situation consistent with the invariant and condition;
    - Make progress in such a way as to make the invariant true again.

    ```
    // Invariant must be true here
    while (condition) { // condition must not have side-effects.
        // (Invariant and condition are not necessarily true here.)
        loop body
        // Invariant must again be true here
    }
    // Invariant true and condition false.
    ```

- So if our loop gets the desired answer whenever invariant is true and condition false, our job is done.
- Another way to see this is to consider an equivalent recursive procedure:

  ```
  /** Assuming Invariant, produce a situation where Inveriant
   *  is true and condition is false. */
   void loop() {    // Invariant assumed true here.
       if (condition) {    // Invariant and condition true here.
           loop body
           // Invariant must be true here.
           loop()
           // Invariant true here and condition false.
       } else { /* condition false here. */ }
  }
  ```

- Here, the invariant is the precondition of the function loop.
- The loop maintains the invariant while making the condition false.
- Idea is to arrange that our actual goal is implied by this post-condition.

Arrays

An array is a structured container whose components are **length** and a sequence of simple containers of the same type, numbered from 0. Length field is usually implicit in diagrams.

They are anonymous, like other structured containers. Thus, they are always referred to with pointers.

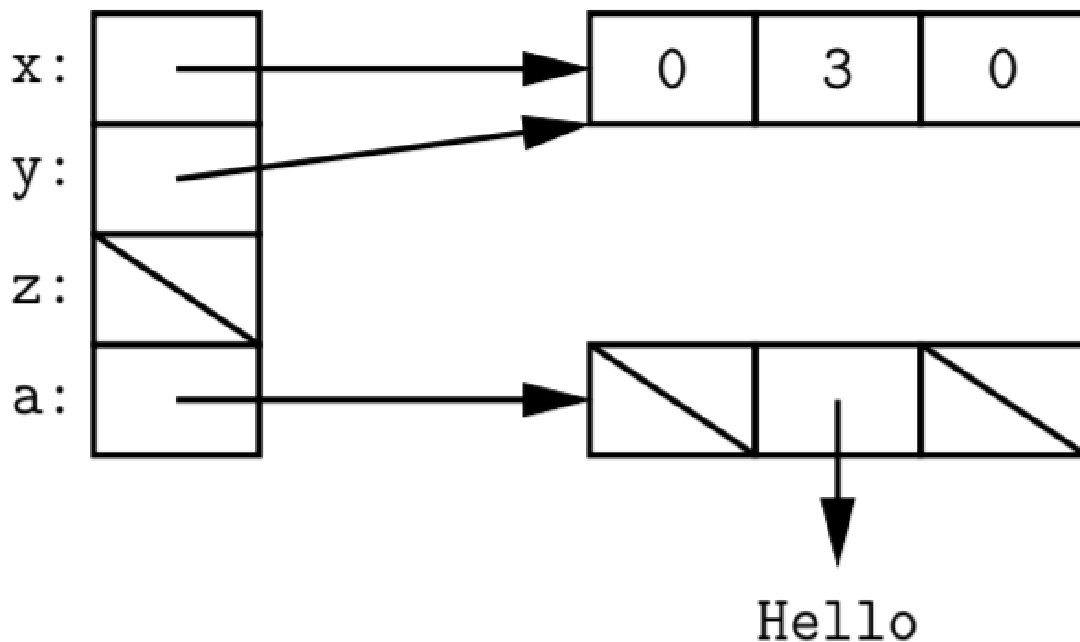
You can get the length of an array `A` with the code `A.length`, or refer to specific elements with `A[i]`, where `i` is an integer expression. Arrays are zero-indexed.

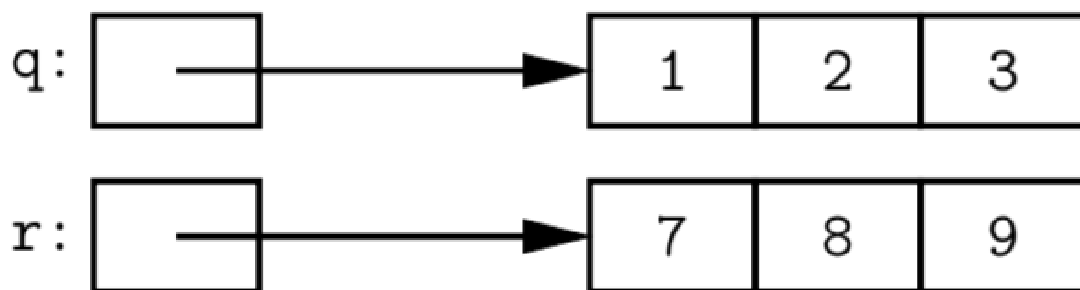
Example

Charting It Out

The following is a very basic example of instantiating arrays and what happens in the environment:

```
int[] x, y, z;
String[] a;
x = new int[3];
y = x;
a = new String[3];
x[1] = 2;
y[1] = 3;
a[1] = "Hello";

int[] q;
q = new int[] { 1, 2, 3 };
// Short form for declarations:
int[] r = { 7, 8, 9 };
```





Accumulate Values

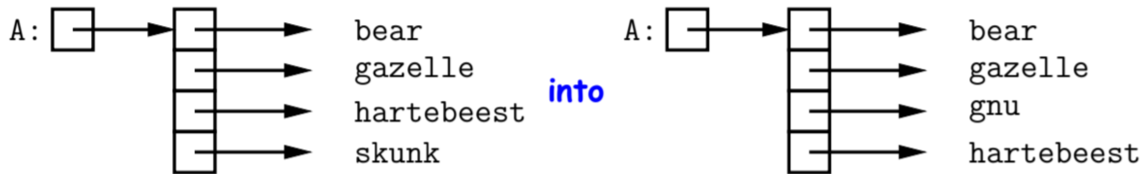
Problem: Sum up the elements of an array.

```
static int sum(int[] A){
    int N;
    N = 0;
    for (int i = 0; i < A.length; i++) {
        N += A[i];
    }
    return N;
}
```

> **Hardcore Code**
> For the hardcore coders, you could also have written:
>
> for (i=0, N=0; i < A.length; N += A[i], i+= 1)

Array Insert

Problem: Want a call `insert(A, 2, "gnu")` to convert (destructively):




```
static void insert (String[] arr, int k, String x) {
    for (int i = arr.length-1; i > k; i -= 1) // Why backwards? {
        arr[i] = arr[i-1];
        /* Alternative to this loop:
           System.arraycopy(arr, k, arr, k+1, arr.length-k-1); */
    arr[k] = x;
}
```

> **Java shortcut**
> Instead of writing `System.arraycopy`, you can avoid this with a shortcut:
>
> ```
> import static java.lang.System.arraycopy;
> ```
>
> This means "define the simple name `arraycopy` to be the equivalent of `java.lang.System.arraycopy` in the source file."
>
> You could also do the same for `out.println`. Or, for example, you are writing a calculator program and need `Math` functions frequently. You can type:
>
> ```
> import static java.lang.Math.*;
> ```

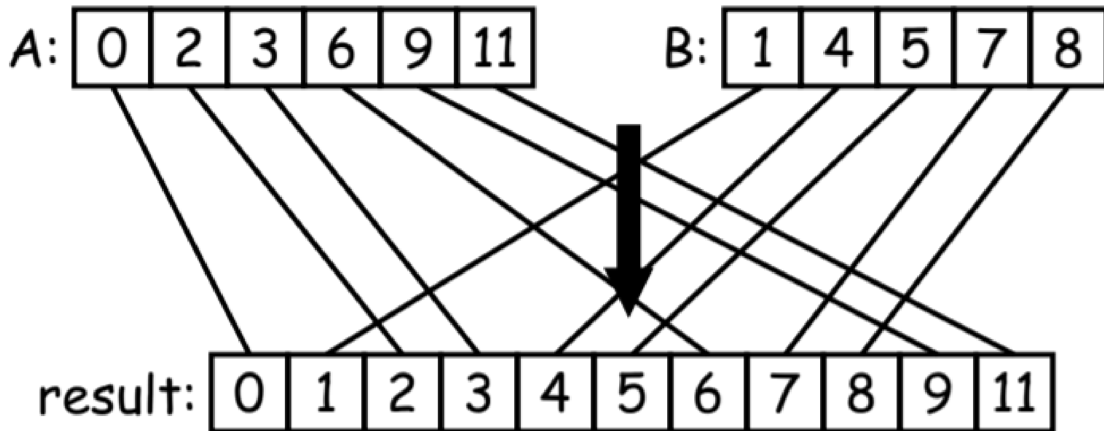
Array Grow

Problem: Suppose that we want to change the description above, so that `A = insert2 (A, 2, "gnu")` does not shove "skunk" off the end, but instead "grows" the array.

```
/** Return array, r, where r.length = ARR.length+1; r[0..K-1]
 *  the same as ARR[0..K-1], r[k] = x, r[K+1..] same as ARR[K..]. */
static String[] insert2(String[] arr, int k, String x) {
    String[] result = new String[arr.length + 1];
    arraycopy(arr, 0, result, 0, k);
    arraycopy(arr, k, result, k+1, arr.length-k);
    result[k] = x;
    return result;
}
```

Array Merge

Given two sorted arrays of ints, A and B, produce their merge: a sorted array containing all items from A and B.



As a strategy, it is useful to solve this recursively by generalizing the original function to allow merging **portions** of the arrays.

```
/** Assuming A and B are sorted, returns their merge. */
public static int[] merge(int[] A, int[] B) {
    return mergeTo(A, 0, B, 0);
}
```

Now that we've written the `merge` functionality, we need to write the general function:

```
/** The merge of A[L0..] and B[L1..] assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1) {
    int N = A.length - L0 + B.length - L1; int[] C = new int[N];
    if (L0 >= A.length) {
        arraycopy(B, L1, C, 0, N);
    } else if {
        (L1 >= B.length) arraycopy(A, L0, C, 0, N);
    } else if (A[L0] <= B[L1]) {
        C[0] = A[L0]; arraycopy(mergeTo(A, L0+1, B, L1), 0, C, 1, N-1);
    } else {
        C[0] = B[L1]; arraycopy(mergeTo(A, L0, B, L1+1), 0, C, 1, N-1);
    }
    return C;
}
```

> **What's wrong with this code?**
> It's rather slow! It's both slow and inefficient (it takes
>
> $$N^2$$
>
> time). So we need to fix something...

Tail-Recursive Merge

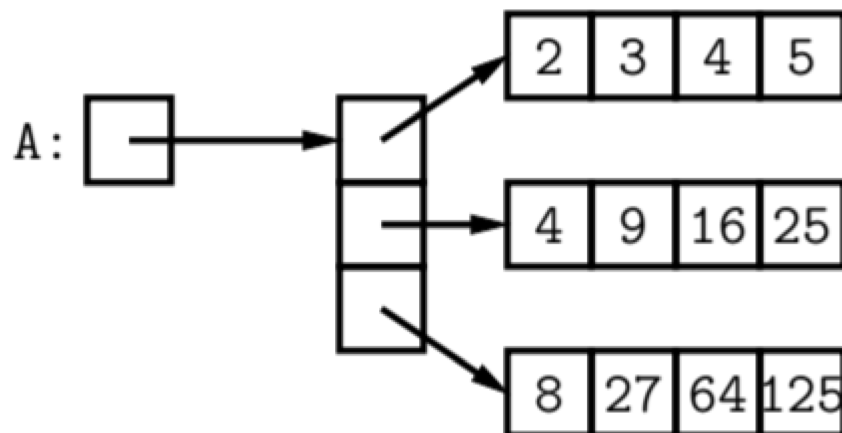
```
public static int[] merge(int[] A, int[] B) {
    return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int
k){
```

```
    if (L0 >= A.length && L1 >= B.length) {
        return C;
    } else if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
        C[k] = A[L0];
        return mergeTo(A, L0 + 1, B, L1, C, k + 1);
    } else {
        C[k] = B[L1];
        return mergeTo(A, L0, B, L1 + 1, C, k + 1);
    }
}
```

Multidimensional Arrays

If we wanted to represent a two or three-dimensional array in Java? The easiest way is to build an array of arrays.

```
int[][] A = new int[3][];
A[0] = new int[] {2, 3, 4, 5};
A[1] = new int[] {4, 9, 16, 25};
A[2] = new int[] {8, 27, 64, 125};
```



Because every element of an array is independent, there is no single "width" property. You could make every array a different length.