Recreation

What is the sum of the coefficients of

\[(1 - 3x + 3x^2)^{743}(1 + 3x - 3x^2)^{744}\]

after expanding and collecting terms?
CS61B Lecture #6: Arrays

- An array is a structured container whose components are
  - **length**, a fixed integer.
  - a sequence of **length** simple containers of the same type, numbered from 0.
  - (.length field usually implicit in diagrams.)
- Arrays are anonymous, like other structured containers.
- Always referred to with pointers.
- For array pointed to by \( A \),
  - Length is \( A\.length \)
  - Numbered component \( i \) is \( A[i] \) (\( i \) is the **index**)
  - Important feature: index can be **any integer expression**.
A Few Samples

Java

```java
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 };
```

Results

```
x: 0 3 0
y:
z:
a: Hello
q: 1 2 3
r: 7 8 9
```

Last modified: Wed Sep 12 13:40:26 2018
Example: Accumulate Values

Problem: Sum up the elements of array A.

```java
static int sum(int[] A) {
    int N;
    N = 0;
    for (int i = 0; i < A.length; i += 1)
        N += A[i];
    return N;
}

// For the hard-core: could have written
int N, i;
for (i=0, N=0; i<A.length; N += A[i], i += 1)
    { } // or just ;

// But please don’t: it’s obscure.
```
Example: Insert into an Array

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

```
A: [bear, gazelle, hartebeest, skunk] into
A: [bear, gazelle, gnu, hartebeest]
```

```java
/** Insert X at location K in ARR, moving items K, K+1, ... to locations
 * K+1, K+2, .... The last item in ARR is lost. */
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;
}
```
(Aside) Java Shortcut

**Useful tip:** Can write just `arraycopy` by including at the top of the source file:

```java
import static java.lang.System.arraycopy;
```

This means “define the simple name `arraycopy` to be the equivalent of `java.lang.System.arraycopy` in the current source file.”

**Can do the same for `out` so that you can write**

```java
out.println(...);
```

in place of

```java
System.out.println(...);
```

**Finally, a declaration like**

```java
import static java.lang.Math.*;
```

means “take all the (public) static definitions in `java.lang.Math` and make them available in this source file by their simple names (the name after the last dot).”

**Useful for functions like `sin`, `sqrt`, etc.**
Growing an Array

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

\[
\begin{aligned}
A: & \quad \text{bear} \\
    & \quad \text{gazelle} \\
    & \quad \text{hartebeest} \\
    & \quad \text{skunk}
\end{aligned}
\]

Why do we need a different return type from \text{insert2}??

Last modified: Wed Sep 12 13:40:26 2018
Example: Merging

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

A: \[0, 2, 3, 6, 9, 11\]  \quad B: \[1, 4, 5, 7, 8\]

result: \[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11\]
Example: Merging Program

Problem: Given two sorted arrays of ints, A and B, produce their merge: a sorted array containing all from A and B.
Remark: In order to solve this recursively, it is useful to generalize the original function to allow merging portions of the arrays.

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

/** 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 is wrong with this implementation?
A Tail-Recursive Strategy

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) {
    ...
}

This last method merges part of A with part of B into part of C. For example, consider a possible call mergeTo(A, 3, B, 1, C, 2)
A Tail-Recursive Solution

```java
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 (??) {
        return C;
    } else if (??) {
        C[k] = A[L0];
        return mergeTo(A, ??, B, ??, C, ??)
    } else {
        C[k] = B[L1];
        return mergeTo(A, ??, B, ??, C, ??)
    }
}
```
A Tail-Recursive Solution

```java
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 (??) {
        C[k] = A[L0];
        return mergeTo(A, ??, B, ??, C, ??)
    } else { 
        C[k] = B[L1];
        return mergeTo(A, ??, B, ??, C, ??)
}
}
```
A Tail-Recursive Solution

```java
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, ??, B, ??, C, ??)
    } else {
        C[k] = B[L1];
        return mergeTo(A, ??, B, ??, C, ??)
    }
}
```
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, ??, B, ??, C, ??)
    }
}
A Tail-Recursive Solution

```java
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);
    }
}
```
Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```java
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    // mergeTo(A, 0, B, 0, C, 0)
    int L0, L1, k;
    L0 = L1 = k = 0;

    while (L0 < A.length && L1 < B.length) {
        if (L0 < A.length && A[L0] <= B[L1]) {
            C[k] = A[L0];
            L0++;
        } else {
            C[k] = B[L1];
            L1++;
        }
        k++;
    }
    return C;
}
```
Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```java
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    // mergeTo(A, 0, B, 0, C, 0)
    int L0, L1, k;
    L0 = L1 = k = 0;

    while (L0 < A.length || L1 < B.length) {
        if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
            C[k] = A[L0];
            ??
        } else {
            C[k] = B[L1];
            ??
        }
    }
    return C;
}
```
Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```java
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    // mergeTo(A, 0, B, 0, C, 0)
    int L0, L1, k;
    L0 = L1 = k = 0;

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

The same, with a for loop:

```java
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int L0, L1;
    L0 = L1 = 0;
    for (int k = 0; k < C.length; k += 1) {
        if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
            C[k] = A[L0]; L0 += 1;
        } else {
            C[k] = B[L1]; L1 += 1;
        }
    }
    return C;
}
```
Alternative Solution: Removing k

Claim: An invariant of the loop is that $k = L_0 + L_1$.

```java
public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int L0, L1;
    L0 = L1 = 0;
    while (L0 + L1 < C.length) {
        if (L1 >= B.length || (L0 < A.length && A[L0] < B[L1])) {
            C[L0 + L1] = A[L0]; L0 += 1;
        } else {
            C[L0 + L1] = B[L1]; L1 += 1;
        }
    }
    return C;
}
```
Multidimensional Arrays

What about two- or higher-dimensional layouts, such as

\[
A = \begin{bmatrix}
2 & 3 & 4 & 5 \\
4 & 9 & 16 & 25 \\
8 & 27 & 64 & 125
\end{bmatrix}
\]
Multidimensional Arrays in Java

These are not primitive in Java, but we can build them as arrays of arrays:

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

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

// or
int[][] A = { {2, 3, 4, 5},
              {4, 9, 16, 25},
              {8, 27, 64, 125} };

// or
int[][] A = new A[3][4];
for (int i = 0; i < 3; i += 1)
    for (int j = 0; j < 4; j += 1)
        A[i][j] = (int) Math.pow(j + 2, i + 1);
```
Exotic Multidimensional Arrays

- Since every element of an array is independent, there is no single “width” in general:

```java
int[][] A = new int[5][];
A[0] = new int[] {};
A[1] = new int[] {0, 1};
A[3] = new int[] {6, 7, 8};
```

- What does this print?

```java
int[][] ZERO = new int[3][];
    new int[] {0, 0, 0};
ZERO[0][1] = 1;
System.out.println(ZERO[2][1]);
```
Exotic Multidimensional Arrays

• Since every element of an array is independent, there is no single “width” in general:

```java
int[][] A = new int[5][];
A[0] = new int[] {};
A[1] = new int[] {0, 1};
A[3] = new int[] {6, 7, 8};
```

• What does this print?

```java
int[][] ZERO = new int[3][];
    new int[] {0, 0, 0};
ZERO[0][1] = 1;
System.out.println(ZERO[2][1]);
```