Programming Examples Using Arrays

Pei-yih Ting

More examples:

Top 50 Array Coding Problems for Interviews

20+ Array Coding Problems and Questions from Programming Interviews

Search and Sort an Array

- Two common problems in processing arrays
  - **Searching** an array to determine the location of a particular value.
  - **Sorting** an array to rearrange the array elements in numerical order.
- Examples
  - Search an array of student exam scores to determine which student, if any, got a particular score.
  - Rearrange the array elements in increasing (decreasing) order by score.
- Algorithm for searching over a sorted array is much more efficient than over an unsorted array.

Algorithm of Linear Search

(Sequential Search)

1. Assume the **target** has not been found.
2. Start with the initial array element.
3. Repeat while the target is not found and there are more array elements
   - 3.1 if the current element matches the target
     - 3.1.1 Set a flag to indicate that the target has been found
     - else
     - 3.1.2 Advance to the next array element
4. if the target was found
   - 4.1 Return the target index as the search result
   - else
   - 4.2 Return -1 as the search result

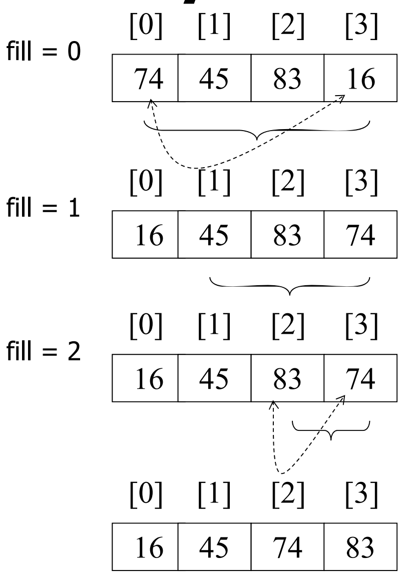
Search an Array

```
int search(const int array[],  /* input - array to search */
                int      target,      /* input - value searched for */
                int      n) {           /* input - number of elements to search */
   int i,
       found = 0,     /*  whether or not target has been found */
       where;          /*  index where target found or NOT_FOUND */

   /*  Compares each element to target */
   i = 0;
   while (!found && i < n) {
      if (array[i] == target)
         found = 1;
      else
         ++i;
   }

   /* Returns index of element matching target or NOT_FOUND */
   if (found)
      where = i;
   else
      where = NOT_FOUND;

   return (where);
}
```

Sorting an Array

Selection sort is an intuitive sorting algorithm.

- Find the index of the smallest element in the array.
- Swap the smallest element with the first element.
- Repeat the above steps for the 2^{nd}, 3^{rd}, …, smallest elements.



Function select_sort

```
int get_min_range(int list[], int first, int last);
void select_sort(int list[],  /* input/output - array being sorted     */
                 int n)       /* input - number of elements to sort    */
{
      int fill,            /* first element in unsorted subarray */
          temp,            /* temporary storage                  */
          index_of_min;    /* subscript of next smallest element */

      for (fill = 0; fill < n-1; ++fill) {
        /* Find position of smallest element in the unsorted subarray */
         index_of_min = get_min_range(list, fill, n-1);

         /* Exchange elements at fill and index_of_min */
         if (fill != index_of_min) {
               temp = list[index_of_min];
               list[index_of_min] = list[fill];
               list[fill] = temp;
         }
      }
}
```


Computing Statistics

- Most common use of arrays is for storage of a collection of related data values.
- Once the values are stored, we can perform some simple statistical computations.

sum = x[0] + x[1] + … + x[MAX_ITEM-1]
mean = sum / MAX_ITEM
sum_square = $x[0]^2 + x[1]^2 + \ldots + x[\text{MAX\_ITEM}-1]^2$
variance = (sum_square – MAX_ITEM * mean^2) / (MAX_ITEM – 1)
standard deviation = sqrt(variance)
histogram?
mode?
median?

Computing Statistics (cont'd)

Figure 8.3

```
#include <stdio.h>
#include <math.h>
#define MAX_ITEM  8  /* maximum number of items in list of data     */
int
main(void)
{
   double x[MAX_ITEM],   /* data list                            */
          mean,          /* mean (average) of the data           */
          st_dev,        /* standard deviation of the data       */
          sum,           /* sum of the data                      */
          sum_sqr;       /* sum of the squares of the data       */
   int    i;

   /* Gets the data   */
   printf("Enter %d numbers separated by blanks or <return>s\n> ",
          MAX_ITEM);
   for  (i = 0;  i < MAX_ITEM;  ++i)
     scanf("%lf", &x[i]);
```

```
/* Computes the sum and the sum of the squares of all data  */
sum = 0;
sum_sqr = 0;
for (i = 0; i < MAX_ITEM; ++i) {
    sum += x[i];
    sum_sqr += x[i] * x[i];
}

/* Computes and prints the mean and standard deviation      */
mean = sum / MAX_ITEM;
st_dev = sqrt((sum_sqr - MAX_ITEM * mean * mean)
                                  / (MAX_ITEM-1));
printf("The mean is %.2f.\n", mean);
printf("The standard deviation is %.2f.\n", st_dev);

/* Displays the difference between each item and the mean   */
printf("\nTable of differences between data values and mean\n");
printf("Index      Item      Difference\n");
for (i = 0; i < MAX_ITEM; ++i)
    printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - mean);

return (0);
}
```


Computing Statistics (cont'd)

```
Enter 8 numbers separated by blanks or <return>s
> 16  12  6  8  2.5  12  14  -54.5

The mean is 2.00.
The standard deviation is 21.75.

Table of differences between data values and mean
Index    Item      Difference
  0      16.00       14.00
  1      12.00       10.00
  2       6.00        4.00
  3       8.00        6.00
  4       2.50        0.50
  5      12.00       10.00
  6      14.00       12.00
  7     -54.50      -56.50
```


Matrix Operations

- Addition
  - Ex. A and B are both 3-by-5, C = A + B

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 3 \\ 4 & 5 & 6 & 5 & 6 \\ 1 & 2 & 5 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 7 & 2 & 3 & 2 & 6 \\ 4 & 1 & 0 & 3 & 2 \\ 1 & 4 & 4 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 4 & 6 & 4 & 9 \\ 8 & 6 & 6 & 8 & 8 \\ 2 & 6 & 9 & 6 & 7 \end{pmatrix}$$

  - $C_{ij} = A_{ij} + B_{ij}$

```
int i, j, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
double c_mat[3][5];
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        c_mat[i][j] = a_mat[i][j] + b_mat[i][j];
```


Matrix Operations (cont'd)

- Multiplication
  - Ex. A and B are both 3-by-5, C = A B^T

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 3 \\ 4 & 5 & 6 & 5 & 6 \\ 1 & 2 & 5 & 4 & 5 \end{pmatrix} \begin{pmatrix} 7 & 4 & 1 \\ 2 & 1 & 4 \\ 3 & 0 & 4 \\ 2 & 3 & 2 \\ 6 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 42 & 18 & 31 \\ 102 & 48 & 70 \\ 64 & 28 & 47 \end{pmatrix}$$

  - $C_{ij} = \sum_{k=1}^{5} A_{ik} B^T_{kj}$

```
int i, j, k, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
double c_mat[3][3];
for (i=0; i<m; i++)
    for (j=0; j<m; j++)
        for (k=0, c_mat[i][j]=0; k<n; k++)
            c_mat[i][j] += a_mat[i][k] * b_mat[j][k];
```

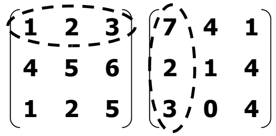
Matrix Operations (cont'd)

- In-place Computation??

  A, B$\in$R^{mxn}, A + B → A; A, B$\in$R^{nxn}, A B^T → A

```
int i, j, k, n=3;
double a_mat[3][3];
double b_mat[3][3], sum;
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
   {
      for (k=0, sum=0; k<n; k++)
         sum += a_mat[i][k] * b_mat[j][k];
      a_mat[i][j] = sum;
   }
```

```
int i, j, m=3, n=5;
double a_mat[3][5], b_mat[3][5];
for (i=0; i<m; i++)
   for (j=0; j<n; j++)
      a_mat[i][j] += b_mat[i][j];
```



Gaussian Elimination

- Solve linear equations using Gaussian Elimination and find out the rank. e.g.

  L1: 2 x + y – z = 8
  L2: -3 x – y + 2 z = -11
  L3: -2 x + y + 2 z = -3

- In matrix notation, we have the following

$$\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right) \Rightarrow \left(\begin{array}{ccc|c} 1 & 0.5 & -0.5 & 4 \\ 0 & 0.5 & 0.5 & 1 \\ 0 & 2 & 1 & 5 \end{array}\right)$$

Normalize 1st row
Multiply 3 and add to 2nd row
Multiply 2 and add to 3rd row

$$\Rightarrow \left(\begin{array}{ccc|c} 1 & 0 & -1 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & -1 & 1 \end{array}\right)$$

Normalize 2nd row
Multiply -0.5 and add to 1st row
Multiply -2 and add to 3rd row

$$\Rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right)$$

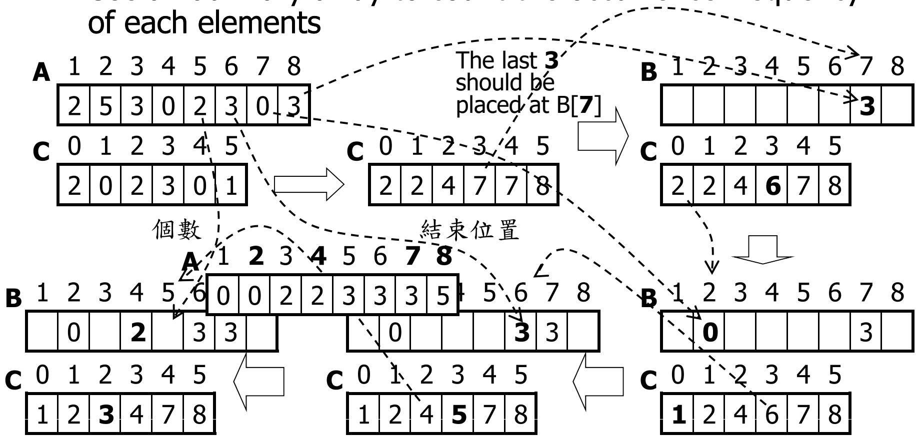
Normalize 3rd row
Multiply 1 and add to 1st row
Multiply -1 and add to 2nd row

x=2, y=3, z = -1 are the result.
Since the left 3 by 3 submatrix is an identity matrix, the number of independent equations is 3. (If one row is all zero, then the number of independent equations is 2 and if two rows are all zero, the number of independent equations is 1, etc.)

Counting Sort

LC1122 Relative Sort Array
ZJ e809 1. Letters

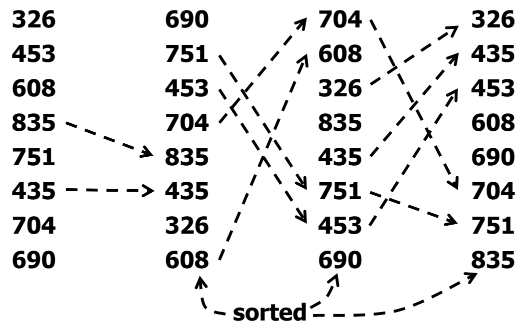
- Elements to be sorted are in a set {0,1,…,k}
- Use an auxiliary array to count the occurrence frequency of each elements



- Non-comparison sort, stable sort, O(n)

Radix Sort

- Stably sort each digits, least significant digits first



Bucket (Bin) Sort
distribute MSB first

- Radix-sort(Array, n)
  1. for i=0 to n-1
  2.   use a stable sort algorithm to sort Array on digit i

Radix-8 Sort (cont'd)

- A radix-8 sort
  - 1-dim array of positive integers to be sorted:
    - e.g. 10**0**, 00**3**, 66**7**, 02**7**, 12**0**, 01**3**, 32**5** **in octal**
  - 2-dim array of integers is used as the working space
    - rows (the buckets) indexed from 0 to 7 and
    - columns indexed from 0 to n-1

	0	1	2		...	n-1
0	100	120				
1						
2						
3	003	013			...	
4						
5	325					
6						
7	667	027				

100	120	003	013	325	667	027

	0	1	2		...	n-1
0	100	003				
1	013					
2	120	325	027			
3					...	
4						
5						
6	667					
7						

100	003	013	120	325	027	667

	0	1	2		...	n-1
0	003	013	027			
1	100	120				
2	325					
3					...	
4						
5						
6	667					
7						

003	013	027	100	120	325	667

Radix-8 Sort (cont'd)

- The radix-8 sorting is done as follows:
  - **Distribute**: Place each value of the one-dimensional vector into a bucket, based on the value's rightmost octal digit. For example, 67 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a distribution pass.
  - **Gather**: Loop through the bucket vector row by row, and copy the values back to the original vector. This procedure is called a gathering pass. The new order of the preceding values in the one-dimensional vector is 100, 3 and 67.
  - **Repeat** this process for each subsequent digit position (2nd rightmost, 3rd rightmost, etc.). e.g. On the second pass, 100 is placed in row 0, 3 is placed in row 0 (3 can be seen as 003) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional vector is 100, 3 and 97. On the third (3rd rightmost) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original vector is in sorted order.

Radix Sort Implementation

```
void radix8Sort(int ndata, int data[]) {
    int buckets[8][MAX], int nBucket[8];
    int i, j, k, index, mult, iBucket;
    int len = maxNumDigits(ndata, data); /* max number of octal digits */
    mult = 1;
    for (i=0; i<len; i++) {
        for (j=0; j<8; j++) nBucket[j] = 0;
        for (j=0; j<ndata; j++) {                    redistribute
            iBucket = data[j] / mult % 8;
            buckets[iBucket][nBucket[iBucket]++] = data[j];
        }
        for (j=0, index=0; j<8; j++)
            for (k=0; k<nBucket[j]; k++)
                data[index++] =
                            buckets[j][k];
        mult *= 8;
    }                                    gather
}
```

```
int maxNumDigits(int ndata,
                         int data[]) {
    int i, max = -1;
    for (i=0; i<ndata; i++)
        if (data[i] > max)
            max = data[i];
    return (log10(max)/log10(8))+1;
}
```


Parallel Arrays

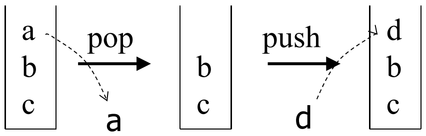
- Two or more arrays with the same number of elements used for **storing related information** about a collection of data objects
- A very common method to organize data with arrays

id[0]	5503	gpa[0]	2.71
id[1]	4556	gpa[1]	3.09
id[2]	5691	gpa[2]	2.98
	...		...
id[49]	9146	gpa[49]	1.92

- id[**i**] and gpa[**i**] refer to the information related to the **i**-th student

Stacks

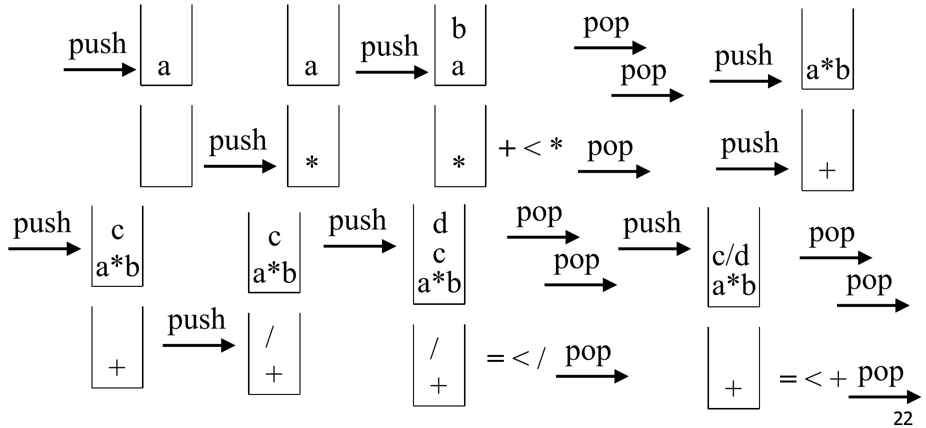
- A **stack** is a data structure in which only the top element can be accessed.
- For example, the plates stored in the spring-loaded device in a buffet line perform like a stack. A customer always takes the top plate; when a plate is removed, the plate beneath it moves to the top.
- Popping the stack: remove a value from a stack.
- Pushing it onto the stack: store an item in a stack.



- Array is one of the approaches to implement a stack.

Algorithm Utilizing Stacks

- Expression evaluation
  a * b + c / d =
  - Two stacks: operand stack, operator stack



Push: Insert a New Element to the Top of Stack

```
#define STACK_SIZE 100
char stack[STACK_SIZE];
int top = -1; /* the position of current stack top */
```

```
push(stack, 'a', &top, STACK_SIZE);
```

```
void
push(char stack[],     /* input/output - the stack */
     char item,        /* input - data being pushed onto the stack */
     int  *top,        /* input/output - pointer to top of stack */
     int  max_size)    /* input - maximum size of stack */
{
     if (*top < max_size-1) {
          ++(*top);
          stack[*top] = item;
     }
}
```


Pop: Remove from Top of Stack an Element

```
char content;
...
content = pop(stack, &top);
```

```
char
pop(char stack[],    /* input/output - the stack */
    int  *top)       /* input/output - pointer to top of stack */
{
    char item;    /* value popped off the stack */

    if (*top >= 0) {
        item = stack[*top];
        --(*top);
    } else {
        item = STACK_EMPTY;
    }
    return item;
}
```

Leader/Dominator

Def: Let **A** be an array storing a sequence of n integers. The leader of the sequence or the dominator of the array is the element whose value occurs more than n/2 times.
For example,

a_0	a_1	a_2	a_3	a_4	a_5	a_6
6	8	4	6	8	6	6

Properties: 1. At most one leader can exist for any sequence.
2. If the dominator exists for a sorted A, $a_{n/2}$ is the dominator.

Problem: Given an array **A** consisting of n integers, find the index of the dominator of **A** if it exists, -1 otherwise.

O(n²) method: for each $\mathbf{a_i}$, loop through the array **A** to determine if it is the leader

O(n log n) method: first sort the array, then verify if $a_{n/2}$ is the dominator of the array.

O(n) method:

Another property of the leader: the leader remains the same for the shorten sequence if two distinct members of the sequence were removed.

let the leader occurs k times, k>n/2

case ❶ if two distinct non-leader members are removed
$k > n/2 > (n-2)/2$

case ❷ if one non-leader and one leader are removed
$k-1 > n/2-1 = (n-2)/2$

Use a **stack** to process the sequence one-by-one, check the top of the stack, ❶ if not equal then pop and remove both
❷ if empty or equal then push

Note: The elements on the stack must always be the same.
Use just size and value to replace a full functional stack.
Check if the value is the leader for the original sequence.

Note: The elements left on the stack must all be the same.
Just check that it is the leader for the original sequence.

```
#include <stdio.h>
int main() {
  int n, a[100000], i, len, first, value;
  while (1==scanf("%d",&n)) {
    for (i=0; i<n; i++)
      scanf("%d", &a[i]);
    for (len=i=0; i<n; i++)
      if (len==0)
        value = a[i], len = 1;
      else
        len += a[i]==value ? 1 : -1;
    if (len>0)
      for (len=0,i=n-1; i>=0; i--)
        if (a[i]==value)
          len++, first=i;
    printf("%d\n",
           len>n/2 ? first : -1);
  }
  return 0;
}
```


- Find the number of EquiLeaders

An **equileader** is an index S such that $0 \le S < n-1$ and two subsequences A[0], A[1], ..., A[S] and A[S+1], A[S+2], ..., A[n−1] have leaders of the same value.

Note:

1. The leader for both subsequences must be the leader for the whole sequence A[0], A[1], …, A[n-1].
2. For each S such that $0 \le S < n-1$, check if both subsequences have the same leaders.

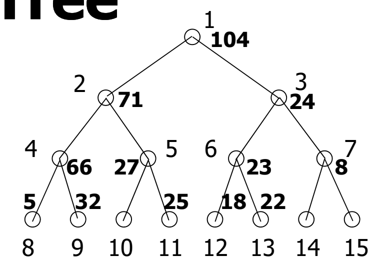
Caterpillar Method

- SpecifiedSubsequenceSum

- CountTriangles

Binary Tree

- A binary tree is a tree data structure in which each node has at most two children.



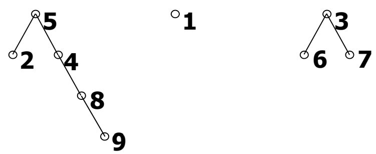
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
104	**71**	**24**	**66**	**27**	**23**	**8**	**5**	**32**	**NIL**	**25**	**18**	**22**	**NIL**	**NIL**

- To represent a binary tree, the value in node label i can be stored in cell i of an array
- The parent of node label i is node label $\lfloor i/2 \rfloor$
  The left child of node label i is node label 2*i
  The right child of node label i is node label 2*i+1

recursion\binaryTreeTraversal.cpp

Disjoint Set

- Pairwise disjoint sets X and Y satisfies $X \cap Y = \phi$
  e.g. {2, 4, 5, 8, 9}, {1}, {3, 6, 7}
- Each set can be represented as an arbitrary structured tree and the root is marked as the representative of that set



- This disjoint sets can be represented as an array – parent, parent[i] is the parent of i. A root's parent is itself.

parent

1	2	3	4	5	6	7	8	9
1	**5**	**3**	**5**	**5**	**3**	**3**	**4**	**8**