CS1010E: Programming Methodology

Tutorial 07: Arrays

20 Mar 2017 - 24 Mar 2017

1. Discussion Questions

(a) [1D Traversal] What is/are the output of code fragments below?

i.
```
int foo(int arr[], int n) {
  int i, s=0;
  for(i=0; i<n; i++)
    s+=arr[i];
  return s;
}
int main() {
  int arr[] = {1,2,3,4,5,6,7,8,9,10};
  printf("%d", foo(arr, 5));
}
```

i. ________________________________

ii.
```
double foo(int arr[], int n) {
  int i, s=0;
  for(i=0; i<n; i++)
    s+=arr[i];
  return s/n;
}
int main() {
  int arr[] = {1,2,3,4,5,6,7,8,9,10};
  printf("%d", foo(arr, 6));
}
```

ii. ________________________________

iii.
```
double foo(int arr[], int n) {
  int i, s=0;
  for(i=0; i<n; i++)
    if(arr[i]%2) s++;
  return s*1.0/n;
}
int main() {
  int arr[] = {1,2,3,4,5,6,7,8,9,10};
  printf("%d", foo(arr, 7));
}
```

iii. ________________________________

(b) [Array Modification] What is the content of the array `arr` at the end of the code fragment below?

i.
```c
void foo(int arr[], int n) {
  int i;
  for(i=0; i<n; i++)
    if(arr[i] < 0) arr[i] *= -1;
}
int main() {
  int arr[] = {0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5};
  foo(arr, 10);
}
```

i. ______________________

ii.
```c
void foo(int arr[], int arr2[], int n) {
  int i;
  for(i=0; i<n; i++)
    arr2[i] = arr[i];
}
int main() {
  int arr[]  = {1, 2,  3, 4,  5, 6,  7, 8,  9,   10};
  int arr2[] = {0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5};
  foo(arr2, arr, 8);
}
```

ii. ______________________

2. Program Analysis

(a) [1D Reasoning] What is/are the output of code fragments below?

i.
```c
int foo(int arr[], int n) {
  return n==0 ? arr[0] : foo(arr, n-1) < arr[n] ? foo(arr, n-1) : arr[n];
}
int main() {
  int arr[] = {3, 2, 5, 9, 6, 7, 8, 1, 4, 0};
  printf("%d %d", foo(arr, 3), foo(arr, 9));
}
```

i. `2 0 (finding minimum)`

ii.
```c
int bar(int arr[], int n) {
  return n==0 ? 0 : arr[bar(arr, n-1)] > arr[n] ? bar(arr, n-1) : n;
}
void foo(int arr[], int n) {
  if(n == 0) return;
  int i  = bar(arr, n), t = arr[n];
  arr[n] = arr[i]; arr[i] = t;
  foo(arr, n-1);
}
int main() {
  int arr[] = {2, 3, 5, 0, 6, 7, 8, 1, 4, 9}, i;
  foo(arr, 9);
  for(i=0; i<10; i++)
    printf("%d ", arr[i]);
}
```

ii. `0 1 2 3 4 5 6 7 8 9  (sorting)`

iii.
```
int main() {
  int arr[10] = {0}, i, j, k;
  for(i=0; i<10; i++) {
    k = rand()%10;
    for(j=0; j<i; j++) if(arr[j] == arr[i]) {
      i--; break;
    }
    if(j == i) arr[i] = k;
  }
  for(i=0, j=0; i<10; i++)
    j += arr[i];
  printf("%d", j);
}
```

iii. 45

(b) [Reference] What is/are the output of code fragments below?

i.
```
double P(int d, double p[], int n) {
  if(d/1000 >= 0 && d/1000 < n) return p[d/1000]*d/1000 + 0.5;
  else                          return p[n-1]   *d/1000 + 0.5 + 1.5;
}
void I(double p[], int n, double r) {
  while(n-->0) p[n] *= (1.0 + r);
}
void E(double p1[], double p2[], int n) {
  int i=n;
  while(n-->0) p2[n] = p1[n];
  I(p2, i, 0.1);
}
int main() {
  double CC[] = {0.5, 1.5, 2.5}, YT[3] = {0};
  int d1 = 1600, d2 = 2600; printf("%.2lf ", P(d1, CC, 3));
  E(CC, YT, 3);             printf("%.2lf ", P(d2, YT, 3));
  I(CC, 3, 0.6);            printf("%.2lf ", P(d1, CC, 3));
                            printf("%.2lf ", P(d2, YT, 3));
  return 0;
}
```

i. 2.90 7.65 4.34 7.65 (note how 2^{nd} == 4^{th})

3. Designing a Solution

(a) [Sorting; Simulation] Grading for a course in NUS is based on the performance of the entire class. The better the class is, the harder it is to get a good grade. Conversely, the worse the class is, the easier it is to get a good grade. However, if the entire class is doing so well (respectively, so poorly), it is possible for the entire class to get good grades (respectively, everyone can fail).

Assuming a class of 100 students, the following is the rule for determining the grades of each students:

A: Given to the best 10 students. However, any students with `marks` $\geq$ `90` is also granted grade A.
B: Given to the next 30 students after the last student with grade A.
C: Given to the next 20 students after the last student with grade B.
D: Given to the next 20 students after the last student with grade C.
E: Given to the next 10 students after the last student with grade D.
F: Given to the last 10 students. However, any students with `marks` $<$ `30` is considered to have failed the course.

Given a list of student marks, print the lowest mark for each of the grades (*we call this lowest mark the* `"cutoff"` *mark, although it is not quite a best fit for the term*). If there are no students with the given grade, print `"-1"` (*without quote*). For instance, when *condensed* into only 10 students (*for illustration only*), the following marks: `[50, 65, 66, 67, 70, 99, 98, 96, 12, 11]` will yield the following output:

```
96    | 3 students with Grade A: lowest = 96
66    | 3 students with Grade B: lowest = 66
50    | 2 students with Grade C: lowest = 50
-1    | No one with Grade D
-1    | No one with Grade E
11    | 2 students with Grade D: lowest = 11
```

Write your program below:

```
// Additional Helper Functions:
// - Swap adjacent values in an array
void swap(int arr[], int idx) {
  int temp = arr[idx];
  arr[idx] = arr[idx+1];
  arr[idx+1] = temp;
}
// - Sort the array marks in descending order
void sort(int marks[]) {
  int i, j; bool changed = true;
  do {
    changed = false;
    for(i=0; i<99; i++) {
      if(marks[i] < marks[i+1]) { // if not sorted in descending order
        swap(marks, i);           // - swap adjacent values
        changed = true;           // - mark changes: may lead to other changes
      }
    }
  } while(changed);
}

void grading(int marks[]) {
  /* marks: the marks of each students */
  int i, num, idx;                    // indexing variable
  int cutoff[] = {-1,-1,-1,-1,-1,-1}; // cutoff point recorded
  int counter[] = {30, 20, 20, 10};   // limit for grade B-E
  sort(marks); // Sorting in descending order

  // Grade A
  num = 0; idx = 0;
  while(idx < 100 && (num < 10 || marks[idx] >= 90)) {
    num++; idx++; // get the next person
  }
  cutoff[0] = marks[idx - 1];

  // Grade B-E: i=0 --> B, i=1 --> C, i=2 --> D, i=3 --> E
  for(i=0; i<4; i++) {
    if(idx < 100 && marks[idx] >= 30) { // check for end or failure
      num = 0;
      while(idx < 100 && num < counter[i] && marks[idx] >= 30) {
        num++; idx++;
      }
      cutoff[i + 1] = marks[idx - 1];
    }
  }
  if(idx < 100) cutoff[5] = idx[99]; // Grade F: the remainder
  for(i=0; i<6; i++) { // Printing
    printf("%d\n", cutoff[i]);
  }
}
```

(b) [Array Modification] Array rotation is an operation on array that treats the entire array as circle. In a sense, the end point of the array is connected to the starting point. The actual operation is to shift the elements in the array in the given direction. Due to the circular property, no elements will be out-of-bounds. A simple example is shown below:

```
[1 2 3 4 5 6 7 8 9] // original array
[7 8 9 1 2 3 4 5 6] // after shift right by 3
```

i. Write a function to *rotate* the array to the *right* by 1. Use the following template:

```
void rotate(int arr[], int n) {
  /* arr: the array to be rotated, n: the size of the array */
  int i, temp = arr[n-1];
  for(i=n-1; i>0; i--) {
    arr[i] = arr[i-1];
  }
  arr[0] = temp;
}
```

ii. Write a function to *rotate* the array to the *right* by `K`. Use the following template:

```
void rotateByK(int arr[], int n, int K) {
  /* arr: the array to be rotated, n: the size of the array, K: #rotation */
  int i;
  for(i=0; i<K; i++) {
    rotate(arr, n);
  }
}
```

(c) [Array Modification] Array reversal is an operation on array such that the final array is a mirror of the initial array when reflected right in the middle. A simple example is shown below:

```
[1 2 3 4 5 6 7 8 9] // original array
[9 8 7 6 5 4 3 2 1] // reversed array
```

i. Write a function to *reverse* an array. Note that you do not know what is the maximum size of the array, and hence, you cannot create a temporary array. Use the following template:

```
void reverse(int arr[], int n) {
  /* arr: the array to be rotated, n: the size of the array */
  int s_idx = 0, e_idx = n-1, temp; // start & end index and temp value
  while(s_idx < e_idx) {            // swap until midway (why just midway?)
    temp       = arr[s_idx];
    arr[s_idx] = arr[e_idx];
    arr[e_idx] = temp;
  }
}
```

4. Challenge

(a) [Efficiency; In-Place] Look back at the array rotation function above. Note how inefficient the function is. In fact, it can be made much more efficient if given an index i, we can know where element at this location should land at. Assuming that index is j, we can then *swap* the two elements. This should be repeated until all the elements have been swapped into its correct location. Write a function to *rotate* the array to the *right* by K. Note that you **cannot** create a temporary array to store the intermediate result. Use the following template:

```
void efficientRotateByK(int arr[], int n, int k) {
  /* arr: the array to be rotated, n: the size of the array, K: #rotation */
  int outer_loop = gcd(size, k), inner_loop = size/outer_loop;
  int i, j, go_to, idx, temp, val; // iterator & temp variables
  for(i=0; i<outer_loop; i++) {
    idx = i; val = arr[i];
    for(j=0; j<inner_loop; j++) { // swap to the correct location
      go_to = (idx + k) % n;      // final index (idx goto (idx+k)%n)

      /* SWAP */
      temp       = val;
      val        = arr[go_to];
      arr[go_to] = temp;

      idx = go_to;                // update current index
    }
  }
}
```

Hint: *Consider a simple case when* K *is* ***relatively prime*** *with* n. *In this case, there is only a single rotation needed. The element at index i is only placed at index $(i + K)$* mod *n (**why?**). For instance, the sequence can be shown below for $n = 8$ and $K = 7$:*

```
[1, 2, 3, 4, 5, 6, 7, 8]
[6, 2, 3, 4, 5, 1, 7, 8]  | i = 0, j = i+K%n = (0+5)%8 = 5
[3, 2, 6, 4, 5, 1, 7, 8]  | i = 5, j = i+K%n = (5+5)%8 = 2
[8, 2, 6, 4, 5, 1, 7, 3]  | i = 2, j = i+K%n = (2+5)%8 = 7
[5, 2, 6, 4, 8, 1, 7, 3]  | i = 7, j = i+K%n = (7+5)%8 = 4
[2, 5, 6, 4, 8, 1, 7, 3]  | i = 4, j = i+K%n = (4+5)%8 = 1
[7, 5, 6, 4, 8, 1, 2, 3]  | i = 1, j = i+K%n = (1+5)%8 = 6
[4, 5, 6, 7, 8, 1, 2, 3]  | i = 6, j = i+K%n = (6+5)%8 = 3
[4, 5, 6, 7, 8, 1, 2, 3]  | i = 3, j = i+K%n = (3+5)%8 = 0
```

On cases where the two values are not co-prime, you have to find a special property of numbers to continue with this line of reasoning. Additionally, the solution will not have this special function implemented, look back at earlier Lecture Notes to find an efficient implementation of this function.

(b) [Optimization] Knapsack problem is a problem in combinatorial optimization where you are given a set of Item. Each Item is associated with a Weight and a Value. Assume that there is only one of each Item (*this can be generalized by having duplicate entry in the set of Item*).

When selecting an Item, you only have a *limited* space in the knapsack. This space is abstracted as a maximum Weight it can carry. Given the set of Item and the maximum Weight that can be carried in the knapsack, find the maximum value that can be obtained from the set of Item. Write your program below:

```
int knapsack(int W[], int V[], int n, int maxW) {
  /* W: weights of items, V: values of items, n: #items, maxW: max weight */
  return knapsackREC(W, V, n, maxW, 0);
  // basically just call knapsackREC with initialized value
  // - knapsackREC is called "helper" function
}
int knapsackREC(int W[], int V[], int n, int maxW, int val) {
  /* ..., val: current max value */
  int taken = -1, not_taken = -1; // value when taken and not taken

  // end of item OR max weight has been reached
  if(n == 0 || maxW == 0) {
    return val;
  }

  if(W[n-1] <= maxW) { // check if item can be taken
    taken = knapsackREC(W, V, n-1, maxW-W[n-1], val+V[n-1]); // item taken
  }
  not_taken = knapsackREC(W, V, n-1, maxW, val);             // item NOT taken

  return (taken > not_taken) ? taken : not_taken; // return maximum
}
```

Hint: *Consider the following two cases when you are given the item at index i*

- Take the item at index *i* (*if possible*): *What is the resulting maximum value obtained from this action? Is there a function to compute that value?*
- Do not take the item at index *i* (*always possible*): *What is the resulting maximum value obtained from this action? Is there a function to compute that value?*

Once you have considered all the possible items at all indexes, your job is done.