CSCI 135 Software Design and Analysis, C++ Homework 4 Solution

Saad Mneimneh
Hunter College of CUNY

Problem 1: The next number
Consider an array a of size n that contains integers in $[0, 9]$. Such an array can be interpreted as an n-digit number, by simply listing the elements in order. For instance, here's a function that takes such a array and its length as parameters, and outputs the numeric representation.

```
void print(int * a, int n) {
  for (int i=0; i<n; i=i+1)
    cout<<a[i];
  cout<<'\n';
}
```

The goal of this problem is to rearrange the array to find the next number that can be represented by the same set of digits. For example, if the array is:

1 4 6 2 9 5 8 7 3

then the next number is given by:

1 4 6 2 9 7 3 5 8

To do this, we have the following algorithm (perform the steps on the above example and make sure you understand them):

1. find the largest i such $a[i] < a[i + 1]$
2. if no such i is found, skip to 6
3. find the largest j such that $a[j] > a[i]$
4. swap $a[i]$ and $a[j]$
5. reverse $a[i + 1 \dots n - 1]$
6. stop

(a) Write a function called swap that takes an array a and two integers i and j and swaps $a[i]$ and $a[j]$.

Solution: This is exactly what we have seen in class.

```
void swap(int * a, int i, int j) {
  int temp=a[i];
  a[i]=a[j];
  a[j]=temp;
}
```

(b) Write a function called reverse, that takes an array a and two integers i and j and reverses the order of elements $a[i], \ldots, a[j]$ in the array.

Solution: We can use swap. When we swap $a[i]$ and $a[j]$, we increase i by 1 and decrease j by 1, and repeatedly swap the extremities until $i \geq j$.

```
void reverse(int * a, int i, int j) {
  while (i<j) {
    swap(a,i,j);
    i=i+1;
    j=j-1;
  }
}
```

(c) Write a function called next that takes an array and its size and rearranges it according to the above algorithm to obtain the next number.

Solution:

```
void next(int * a, int n) {
  int i=n-2;             //start with i=n-2 and decrease i until
  while(a[i]>=a[i+1]) { //we find the largest i s.t. a[i]<a[i+1]
    i=i-1;
  if (i<0)
    return; //if no such i exist, we simply return
  }
  int j=n-1;
  while(a[j]<=a[i]) //now we find the largest j s.t. a[j]>a[i]
    j=j-1;
  swap(a,i,j);          //swap
  reverse(a,i+1,n-1);   //and reverse
}
```

(d) Have fun in main().

Problem 2: The jumping game
Consider an array of size n that contains integers in the range $[0, n-1]$. Such an array will represent a jumping game as follows. We first read $a[0]$. If $a[0] = i$, we read $a[i]$. If $a[i] = j$, we read $a[j]$, and so on. We repeatedly do this until we read a 0. This is when the jumping game terminates. However, there is a possibility that the game will never terminate.

Example 1: if the array is [1,3,4,2,0] then:
a[0]=1
a[1]=3
a[3]=2
a[2]=4
a[4]=0 terminates

Example 2: if the array is [1,3,3,2,0] then:
a[0]=1
a[1]=3
a[3]=2
a[2]=3
a[3]=2
. . . it will never terminate

(a) Write a function called terminate that takes an array and its length as parameters and returns true if the jumping game on this array will terminate, and false otherwise. Your function is not allowed to change the array or use an auxiliary array.

Solution: Since the length of the array is known, if we make $n-1$ jumps before finding 0, we know that we are going to cycle because we have already visited n entries, including $a[0]$.

```
bool terminate(int * a, int n) {
  int i=a[0];
  for (int step=1; step<n && i>0; step=step+1) //this will stop after n-1
    i=a[i];  //jump                            //steps or when 0 is found
  return (i==0);
}
```

(b) Write a function called terminate that takes as parameter an array the size of which is unknown (the length is not passed as a parameter), and returns true if the jumping game on this array will terminate, and false otherwise. This time, your function is allowed to change the array.

Solution: Without knowing the size of the array, we can keep track of the entries we visit. Each time we visit $a[i]$ for some i, we change $a[i]$ to a special value, say -1. If we ever reach -1, we know we have cycled.

```
bool terminate(int * a) {
  int i=a[0];
  while (i>0) { //this will stop if we either find 0 or -1
    int j=i;
    i=a[i];  //jump
    a[j]=-1; //leave a trace
  }
  return (i==0);
}
```

In both cases, if the jumping game does not terminate, your function is not supposed to run indefinitely.

Problem 3: Fishing for primes
We have seen in class a program that generates all primes between 1 and 100. For illustration, I will convert this to a function that takes an integer n as parameter and counts the number of primes less than n.

```
int countPrimes(int n) {
  int count=0;
  for (int i=2; i<n; i++) {
    bool prime=true;
    for (int d=2; d*d<=i; d++)
      if (i%d==0) {
        prime=false;
        break;
      }
    if (prime)
      count=count+1;
  }
  return count;
}
```

To count the number of primes less than 10 millions, we can do the following:

```
int main() {
  cout<<countPrimes(10000000)<<'\n';
}
```

(a) Run the above program and see how long it takes.

(b) If we can use more memory, a faster way to count or generate primes is due to Eratosthenes (250 BC). The idea is to maintain an array of length n, where $a[i]$ is true if and only if i is prime. To begin with, all entries in the array are initialized to true. Then starting from $i = 2$ (the first prime) until $i = n - 1$:

- if $a[i]$ is true, we found a prime i and, as a result, all multiples of i are not prime, i.e. $a[2i], a[3i], a[4i], \dots$ will all be changed to false.
- if $a[i]$ is false, i is not prime.

Write a function called eratothenes analogous to the function countPrimes that implements this algorithm. The function will

- first use dynamic memory allocation to create an array of bool of size n.
- then initialize all the entries in the array to true.
- then perform the algorithm described above to count the number of primes less than n.
- finally delete the array (free memory) and return the count.

Solution:

```
int eratosthenes(int n) {
  bool * a=new bool[n];              //allocate enough memory
  for (int i=2; i<n; i++)
    a[i]=true;                       //initialize a[i] to true for all i
  int count=0;
  for (int i=2; i<n; i++)
    if (a[i]) {                      //true means i is prime
      count++;
      for (int j=i+i; j<n; j=j+i)  //set all multiples of i to false
        a[j]=false;
    }
  delete[] a;                        //free memory
  return count;
}
```

(c) Compare the running time of the two functions when n is 10 millions. What about 100 millions?