ESO207A: Data Structures and Algorithms
Assignment 1 Solution

Ayush Bansal
Roll No. 160177

January 16, 2018

I Problem 1

Universe of n (constant) digit natural number.
I will design it using arrays and each element of this array corresponds to a digit of the number and in implementation each array element will be **1 byte** integer i.e. its type will be **uint8_t**.

During working on arrays of digits the numbers stored in the array will be in **reverse** order i.e. if we have to store 25381, it will be stored as 18352.
The number will be defined as a structure which has 2 arguments which will be the **array** and **length of array**.

```
typedef struct large_number{
  int len;
  uint8_t* arr;
} numb;
```

Addition of 2 numbers, input will be the 2 arrays and their lengths respectively, their sum will of length $max(n1, n2) + 1$.

```
void add(uint8_t arr1[], int n1, uint8_t arr2[], int n2, uint8_t ans[])
{
  uint8_t sum, carry = 0;
  int i = 0;
  while(i < max(n1,n2))
  {
    if(i < n1 && i < n2)
      sum = arr1[i] + arr2[i] + carry;
    else if(i < n1)
      sum = arr1[i] + carry;
    else if(i < n2)
      sum = arr2[i] + carry;
    ans[i] = sum % 10;
    carry = sum / 10;
    i++;
  }
  ans[i] = carry;
}
```

Successor: For this function we can just use **add** function for adding 1 to the answer.

Subtraction: We will compare the 2 inputs and if first number is smaller than second then we raise an **exception**. The function will take 2 numbers and return the ans in the array of $len1$ length.

```
void sub(uint8_t arr1[], int n1, uint8_t arr2[], int n2, uint8_t ans[])
{
  uint8_t arr1_val,borrow = 0;
  int i = 0;
  while(i < n2)
  {
    arr1_val = arr1[i];
    if(borrow && arr1[i] == 0)
    {
      ans[i] = 9 - arr2[i];
      i++;
      continue;
    }
    if(borrow)
    {
      arr1_val = arr1[i] - 1;
      borrow = 0;
    }
    if(arr1_val < arr2[i])
    {
      ans[i] = 10 + arr1_val - arr2[i];
      borrow = 1;
    }
    else
      ans[i] = arr1_val - arr2[i];
    i++;
  }
  while(i < n1)
  {
    if(borrow && arr1[i] == 0)
    {
      ans[i] = 9;
      i++;
      continue;
    }
    if(borrow)
    {
      ans[i] = arr1[i] - 1;
      borrow = 0;
    }
    else
      ans[i] = arr1[i];
    i++;
  }
}
```

Multiplication: In this I will take digits of the second number one at a time and multiply first number with it and add to get the final result in an array of length $len1 + len2 + 1$.

```
void unitmult(uint8_t arr[], int n, uint8_t unit, uint8_t ans[], int offset)
{
  uint8_t mul, carry = 0;
  int i = 0;
  while(i < n)
  {
    mul = arr[i]*unit + carry;
    ans[i+offset] = mul % 10;
    carry = mul / 10;
    i++;
  }
  ans[i+offset] = carry;
}

void mult(uint8_t arr1[], int n1, uint8_t arr2[], int n2, uint8_t ans[])
{
  uint8_t unit;
  int tmp2 = n2;
  while(tmp2 > 0)
  {
    unit = arr2[n2 - tmp2];
    uint8_t temp[n1+(n2-tmp2)+1];
    for(int j = n2-tmp2; j > 0; j--)
      temp[j-1] = 0;
    unitmult(arr1, n1, unit, temp, n2-tmp2);
    add(temp,n1+(n2-tmp2)+1,ans,n1+(n2-tmp2)+1,ans);
    tmp2--;
  }
}
```

Division: I will first compare the second number to 0, if it 0 then raise an **exception**, if $len1 < len2$ then we directly know the answer - 0. The final result will be stored in an array of length $len1 - (len2 - 1)$.

```
void div(uint8_t arr1[], int n1, uint8_t arr2[], int n2, uint8_t ans[])
{
  uint8_t dvd[n1];
  uint8_t dvr[n1];
  for(int i=0;i<n1;i++)
    dvd[i]=arr1[i];
  for(int i=0;i<n1-n2;i++)
    dvr[i]=0;
  for(int i=0;i<n2;i++)
    dvr[i+(n1-n2)]=arr2[i];
  for(int i=n1-n2;i>=0;i--)
  {
    if(compare(dvd,dvr,n1) == -1)
      ans[i] = 0;
    else
    {
      int count = 0;
      while(compare(dvd,dvr,n1) != -1)
      {
        ++count;
        sub(dvd,n1,dvr,n1,dvd);
      }
      ans[i]=count;
    }
    if(i != 0)
    {
      int tmp = dvr[0];
      for(int j=0;j<n1-1;j++)
      {
        dvr[j] = dvr[(j+1)%n1];
      }
      dvr[n1-1] = tmp;
    }
  }
}
```

Constant: It takes the number and returns the string of the number itself

```
void constant(uint8_t arr[], int n, char str[])
{
  for(int i=0;i<n;i++)
    str[n-(i+1)] = arr[i] + '0';
  str[n] = '\0';
}
```

Miscellaneous Functions

These functions will not be available for use outside of **ADT** i.e. these are used for implementing the above functions only.

String to Int: To read integer and store in the array in reverse order.

```
void stringtoint(char str[], uint8_t arr[], int n)
{
  int i = 0;
  while(str[i] != '\0')
  {
    arr[n-(i+1)] = str[i] - '0';
    i++;
  }
}
```

Reverse: To reverse the array.

```
void reverse(uint8_t arr[], int n)
{
  for(int i=0; i<n/2; i++)
  {
    int tmp = arr[i];
    arr[i] = arr[n-i];
    arr[n-i] = tmp;
  }
}
```

Compare: To compare 2 numbers.

```
int compare(uint8_t dvd[], uint8_t dvr[], int n)
{
  int i = n-1;
  while(dvd[i] == dvr[i])
    i--;
  if(i < 0 || dvd[i] >= dvr[i])
    return 1;
  return -1;
}
```

Max: Finding max of 2 integers/

```
int max(int a, int b)
{
  return a>b?a:b;
}
```

Printans: To print the provided number(array).

```
void printans(uint8_t ans[], int n)
{
  int i = n-1;
  while(ans[i]==0 && i >= 0)
    i--;
  if(i<0)
  {
    printf("0\n");
    return;
  }
  while(i >= 0)
  {
    printf("%d", ans[i]);
    i--;
  }
  printf("\n");
}
```

II Problem 2

2.1 Queues

I will design **queues** using **Linked List**.

```
typedef struct Node
{
  int x;
  struct Node *n;
} node;
```

There will be an **enqueue** operation which will take address of pointer to the first element as an argument and add the element to the end of the list.

```
void enqueue(node** head, int elem)
{
  node* new = (node*)malloc(sizeof(node));
  new->x = elem;
  new->n = NULL;
  if(*head == NULL)
  {
    *head = new;
    return;
  }
  node* curr = *head;
  while(curr->n != NULL)
    curr = curr->n;
  curr->n = new;
}
```

There will be a **dequeue** operation which will take the address of pointer to the first element as an argument and remove that element from the list.

```
void dequeue(node** head)
{
  node* curr = *head;
  *head = curr->n;
  free(curr);
}
```

There will be an **isEmpty** operation which will take the address of the pointer to the first element as an argument and report if queue is empty or not.

```
1  void isEmpty(node** head)
2  {
3    if(*head == NULL)
4      return 1;
5    return 0;
6  }
```

The address of the pointer to the first element will be defined as queue and this is what the above functions will be taking as arguments.

2.2 Sequences

I will design **sequences** using **Linked List**.

```
typedef struct Node
{
  int x;
  struct Node *next;
} node;
```

Here I am taking the contents of the sequence as integers but that be changed according to declaration as to whatever the sequence is going to store.
Here, the **address of the pointer to the first element** will be defined as sequence and this is what the below functions will be taking as arguments.

The sequence will have the following major operations.
Length: This will give us take the sequence and give us an integer which will be the length of the sequence.

```
int length(node** head)
{
  node* curr = *head;
  int count = 0;
  while(curr != NULL)
  {
    count++;
    curr = curr->n;
  }
  return count;
}
```

getValue: This will give take the sequence and an integer index and will give us the value corresponding to that index.

```
int getValue(node** head, int index)
{
  int i = index;
  node* curr = *head;
  while(curr != NULL)
  {
    if(i == 0)
      return curr->x;
    i--;
    curr->next;
  }
  printf("Index out of Range!");
  exit(1);
}
```

find: This will take the sequence and an integer element and will give us the index corresponding to that element and outputs -1 in case element is not found.

```
int find(node** head, int elem)
{
  int i = 0;
  node* curr = *head;
  while(curr != NULL)
  {
    if(curr->x == elem)
      return i;
    i++;
    curr = curr->next;
  }
  return -1;
}
```

insertAt: This will add an element to the sequence at the given position, it will take the sequence and an integer position and element to add.

```
void insertAt(node** head, int index, int elem)
{
  int i = index;
  node* new = (node*)malloc(sizeof(node));
  node* curr = *head;
  while(curr != NULL)
  {
    if(i == 0)
    {
      new->x = elem;
      new->next = curr;
      curr = new;
      return;
    }
    i--;
    curr->next;
  }
  if(curr == NULL && i == 0)
  {
    new->x = elem;
    new->next = NULL;
    curr = new;
    return;
  }
  printf("Index out of Range!");
  exit(1);
}
```

delete: This will delete an element from the sequence at a given index, it will take the sequence and an integer position.

```
void delete(node** head, int index)
{
  int i = index;
  node* curr = *head;
  while(curr != NULL)
  {
    if(i == 0)
    {
      node* tmp = curr;
      curr = curr->next;
      free(tmp);
    }
  }
  printf("Can't delete entry: %d", index);
  exit(1);
}
```

III Problem 3

Program for computing n^k using squaring each time.

Algorithm 1 Computing n^k

```
1:  procedure POWER
2:      read n
3:      read k
4:      t ← 1                          ▷ t will act as primary accumulator
5:      while k > 0 do                 ▷ While power is positive
6:          i ← 2
7:          d ← n                      ▷ d will be secondary accumulator
8:          while i ≤ k do             ▷ Taking power of 2 just less than k
9:              d ← d * d              ▷ Squaring
10:             i ← i * 2              ▷ i will be powers of 2
11:         t ← t * d
12:         k ← k − (i/2)
13:     write t
```

In writing **RAM** program, I will store n, k, t, i, d in registers $1, 2, 3, 4, 5$ respectively.

The **RAM Program** for the above algorithm will be as follows.

```
1             READ   1                    // storing n
2             READ   2                    // storing k
3             LOAD   =1
4             STORE  3                    // t = 1
5   while1:   LOAD   2
6             JZERO  endwhile1            // while k!=0 do
7             LOAD   =2
8             STORE  4                    // i = 2
9             LOAD   1
10            STORE  5                    // d = n
11            JUMP   while2
12  while2:   LOAD   4
13            SUB    2                    // i - k
14            JGTZ   endwhile2            // while (i-k)<=0 do
15            LOAD   5
16            MULT   5
17            STORE  5                    // d = d*d
18            LOAD   4
19            MULT   =2
20            STORE  4                    // i = i*2
21            JUMP   while2
22  endwhile2: LOAD  3
23            MULT   5
24            STORE  3                    // t = t*d
25            LOAD   4
26            DIV    =2
27            STORE  4                    // i = i/2
28            LOAD   2
29            SUB    4
30            STORE  2                    // k = k-i
31            JUMP   while1
32  endwhile1: JUMP  output
33  output:   WRITE  3
34            HALT
```

IV Problem 4

TM Program for doubling of an input consisting of k consecutive 1s.
Initially, the **automaton** will be on the rightmost 1 of the input and the state of the automaton will be 0.
So initial Conditions - **Automaton: RightMost** and **State: 0**.

Initial State	Symbol Read	Write Instruction	Move tape	Next State
0	Blank	Blank	Left	2
	0	0	Right	0
	1	0	Left	1
1	Blank	0	Right	0
	0	0	Left	1
2	Blank	Blank	Right	Halt
	0	1	Left	2

If the input is 1111, the output will be 11111111.
The automaton will convert a 1 into a 0 and then add a 0 to the end of the string one at a time, so there will be twice the number of 0s in the string and no 1s, after which it will convert all 0s to 1.

V Problem 5

TM Program for checking if a binay number is divisible by 3 or not.
Initially, the **automaton** will be on the most significant (Leftmost) bit of the input binary number and the state of the automaton will be 0.
So initial Conditions - **Automaton: MSB** and **State: 0**.
If the automaton **halts** on 1 then binary number is divisible by 3 and it has been accepted, if it **halts** on 0 then it is not divisible by 3.
So **Halts on 1: Accepted** and **Halts on 0: Rejected**.

Initial State	Symbol Read	Write Instruction	Move tape	Next State
0	Blank	1	Stay	Halt
	0	0	Left	0
	1	1	Left	1
1	Blank	0	Stay	Halt
	0	0	Left	2
	1	1	Left	0
2	Blank	0	Stay	Halt
	0	0	Left	1
	1	1	Left	2

The **automaton** will move from Most Significant bit to the right which actually changing the number but chaning the states.
Here, each state number represents the remainder we will get when the binary number to the left of automaton is divided by 3.
So, when we get the first blank after the number is fully read, if the state is 0, then the number is **accepted** else it is **rejected**.