CS 677 Homework Assignment 03

Hai Nguyen

Semtember 18, 2018

1. Consider the following recursive algorithm.

   ALGORITHM *Enigma*(A[0..n-1])
   //Input: An array A[0..n-1] of integer number
   for $i \leftarrow 0$ **to** n - 2 **do**
   for $j \leftarrow i + 1$ **to** n - 1 **do**
   if $A[i] == A[j]$
   return false
   return true

   (a) What does this algorithm do?

   Check for 2 duplicate elements within an array of integer numbers. If there is at least two duplicate elements, the algorithm returns False, otherwise returns True.

   (b) Compute the running time of this algorithm.

   ALGORITHM *Enigma*(A[0..n-1])
   for $i \leftarrow 0$ **to** n - 2 **do** *[Cost c1, n times]*
   for $j \leftarrow i + 1$ **to** n - 1 **do** *[Cost c2, n - i times]*
   if $A[i] == A[j]$ *[Cost c3, n - i - 1 times, worst case]*
   return false
   return true

   Total cost:

$$\begin{aligned} T(n) &\leq c_1 n + \sum_{i=0}^{n-2}(n-i) + \sum_{i=0}^{n-2}(n-i-1) \\ &= c_1 n + \sum_{k=2}^{n} k + \sum_{k=1}^{n-1} k \\ &= O(n^2) \end{aligned}$$

2. (a) Implement in C/C++ a version of bubble sort that alternates left-to-right and right-to-left passes through the data.

   Source code:

```
#include <stdio.h>

int numComp = 0;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

/* Function to print an array */
void printArray(int arr[], int size)
{
```

```
    int i;
    for (i=0; i < size; i++)
        printf("%c ", arr[i]);
}

void bubbleSort(int arr[], int n)
{
   int i, j;
   int leftStartIdx = 1;
   int rightStartIdx = n;

   while (leftStartIdx < rightStartIdx)
   {
       // Left2Right pass
         for (j = leftStartIdx; j <= rightStartIdx - 1; j++)
         {
            numComp++;

            if (arr[j-1] > arr[j])
            {
               swap(&arr[j], &arr[j-1]);
            }
         }


        rightStartIdx -= 1;


        printf("Left2Right pass: ");
        printArray(arr, n);
        printf("\n");

        if (leftStartIdx >= rightStartIdx)
            break;

        // Right2Left pass
         for (j = rightStartIdx - 1; j >= leftStartIdx; j--)
         {
            numComp++;

            if (arr[j - 1] > arr[j])
            {
               swap(&arr[j - 1], &arr[j]);

            }
         }

         printf("Right2Left pass: ");
         printArray(arr, n);
         printf("\n");
```

```
            if (leftStartIdx >= rightStartIdx)
                break;

            leftStartIdx += 1;
        }
    }

int main()
{
    int arr[] = {'E', 'A', 'S', 'Y', 'Q', 'U', 'E', 'S', 'T', 'I', 'O', 'N'};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    return 0;
}
```

Outputs:

```
Left2Right pass: A E S Q U E S T I O N Y
Right2Left pass: A E E S Q U I S T N O Y
Left2Right pass: A E E Q S I S T N O U Y
Right2Left pass: A E E I Q S N S T O U Y
Left2Right pass: A E E I Q N S S O T U Y
Right2Left pass: A E E I N Q O S S T U Y
Left2Right pass: A E E I N O Q S S T U Y
Right2Left pass: A E E I N O Q S S T U Y
Left2Right pass: A E E I N O Q S S T U Y
Right2Left pass: A E E I N O Q S S T U Y
Left2Right pass: A E E I N O Q S S T U Y
```

(b) How many comparisons does this modified version of bubble sort make?

- First Left2Right pass: We have n elements, and it takes us n - 1 comparisons
- First Right2Left pass: After the first left2right pass, we only have n-1 elements, and it takes us n - 2 comparisons
- We continue until we only have 2 elements and we only need to make 1 comparions
  Total comparisons needed:

$$1 + 2 + 3 + \ldots + n - 1 = \frac{n(n-1)}{2}$$

3. Non-recurisve merge sort

   Code:

```
#include <stdio.h>

void printArray(int arr[], int size, bool c = false)
{
    int i;
    for (i=0; i < size; i++)
    {
```

```
        if (c)
            printf("%c ", arr[i]);
        else
            printf("%d ,", arr[i]);
    }
    printf("\n");
}

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
```

```
        k++;
    }
}

void nonRecursiveMerge (int arr [], int n)
{
    int m = 1;
    int i = 0;
    int minVal = 0;

    while (m < n)
    {
        i = 0;
        while (i < n - m)
        {
            minVal = (i + 2 * m - 1 < n - 1) ? (i + 2 * m - 1) : (n - 1);
            merge(arr, i, i + m - 1, minVal);
            i += 2*m;
        }

        printArray(arr, n, true);

        m *= 2;
    }
}

int main()
{
    int arr[] = {'A', 'S', 'O', 'R', 'T', 'I', 'N', 'G',
     'E', 'X', 'A', 'M', 'P', 'L', 'E'};
    int n = sizeof(arr)/sizeof(arr[0]);
    nonRecursiveMerge(arr, n);
    return 0;
}
```

Output:

```
A S O R I T G N E X A M L P E
A O R S G I N T A E M X E L P
A G I N O R S T A E E L M P X
A A E E G I L M N O P R S T X
```

4. Use a loop invariant to prove that the following algorithm computes a^n:

Exp(a, n)

{

    i ← 1

    pow ← 1

    while (i ≤ n)

{

pow ← pow*a

i ← i + 1

}

return pow

}

Use the following loop invariant:

$$\text{pow}_i = \text{a}^i$$

Prove the loop invariant:

- Initialization
  i = 0: $\text{pow}_0 = a^0 = 1$
- Maintenance: Assume that at the start of the i-th iteration $\text{pow}_i = \text{a}^i$
  Then, at the start of the (i+1)-th iteration we will have: $\text{pow}_{i+1} = \text{pow}_i \times a = \text{a}^{i+1}$
- Termination: The loop terminate when i = n. Thus after the loop execution we have:

$$\text{pow}_n = \text{a}^n$$

5. Consider another algorithm for solving the same problem as the one in Homework 2 (problem 1), which recursively divides an array into two halves (call Min2 (A[0...n-1])):

   (a) Set up a recurrence relation for the algorithm's basic operation count and solve it
   ALGORITHM *Min2*(A[*left..right*])
   if *left* = *right* **return** A*[left]* *[Cost c1]*
   else *temp1* ← Min2(A[*left*..(*left* + *right*)/2) *[Cost Theta*(*n*/2)*]*
   temp2 ← Min2(A[(*left* + *right*)/2+1..*right*) *[Cost Theta*(*n*/2)*]*
   if *temp1* ≤ *temp2* **return** *temp1* *[Cost c2]*
   else return *temp2* *[Cost c3]*
   Recurrence relationship:

$$T(n) = \begin{cases} c, & \text{if } n = 1. \\ 2T(n/2) + c, & \text{if n} > 1. \end{cases}$$

   Solve it:

$$\begin{aligned} T(n) &= 2T(n/2) + c \\ &= 2^{lgn}T(1) + c \sum_{i=0}^{i=lgn-1} 2^i \\ &= nT(1) + c(n-1) \end{aligned}$$

   (b) Which of the algorithms Min1 (from Homework 2) or Min2 is faster?
   Min1 algorithm:
   ALGORITHM *Min1*(A[0..n-1])
   if n = 1 return A[0] [Constant time c1]

else *temp* $\leftarrow$ *Min1*(A[0..n-2]) [c2 + T(n-1)]
if *temp* $\leq$ A[n-1] **return** *temp* [Constant time c3]
else return A[n-1] [Constant time c4]

Recurrence relationship:

$$T(1) = c1$$
$$T(n) = c + T(n-1)$$

Solve it:

$$\begin{aligned} T(n) &= c + T(n-1) \\ &= T(1) + (n-1)c \\ &= \Theta(n) \end{aligned}$$

Both algorithms are $\Theta(n)$ so they are equal, however Min1 uses less assignments.