Algorithms and Data Structures (ADS2)

Laboratory Sheet 2

This lab sheet contains material based on Lectures 3-5. This exercise is not for submission but should be completed to gain sufficient experience in the implementation of elementary sorting algorithms, and the testing thereof.

Setup

Datasets for tests can be downloaded from Moodle under Labs/Files.

Exercise

You are to implement some simple recursive algorithms and the MERGE-SORT algorithm via generic Java methods. You should have already implemented programs to check and test performance of your implementations in Lab1.

Part 1

a) Implement in Java the pseudocode for FACT introduced in Lecture 4 (slide 3).

```
public static int fact(int n){
  assert (n > 0);
  if (n == 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

b) Implement in Java `fact-tail`, a tail recursive version of FACT. What is the complexity of `fact-tail`?

```
public static int fact-tail(int n, int acc){
  assert (n > 0);
  assert (acc > 0);
  if (n == 1)
    return acc;
  else
    return fact-tail(n – 1, n * acc);
}
```

O(n)

c) Implement in Java `fact-iter`, an iterative version of FACT. What is the complexity of `fact-iter`?

```
public static int fact-iter(int n){
  assert (n > 0);
  int res = 1;
  for (int j=1; j<=n; j++)
    res *= j;
  return res;
}
```

O(n)

Part 2

a) Implement in Java the pseudocode for FIB introduced in Lecture 4 (slide 25).

```
public static int fib(int n){
  assert (n >= 0);
  if n <= 1
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

b) Write a tail recursive version of the algorithm using linear recursion. This algorithm should have O(n) complexity.

```
public static int fib-lin(int n, int curr, int prev){
  assert(n >= 0);
  assert(curr >= 1);
  assert(prev >= 0);
  if(n == 0)
    return prev;
  return fib-lin(n - 1, curr+prev, curr);
}
```

c) Implement in Java the pseudocode for SHELLSORT.

```
public class ShellSort{

  public static void sort(int a[]){
    int n = a.length;
    int h = 1;

    while (h < n/3) h = 3*h + 1;

    while (h >= 1) {
      for (int i = h; i < n; i++){
        for (int j = i; j >= h && a[j] < a[j-h]); j -= h){
          swap(a, j, j-h);
        }
      }
      h /= 3;
    }
  }
```

Part 3

a) Implement in Java the pseudocode for MERGE-SORT introduced in Lecture 5 (slides 4 and 15).

```
public class MergeSort{

  static void merge(int a[], int p, int q, int r){
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] L = new int[n1 + 1];
    int[] R = new int[n2 + 1];

    for (int i=0; i<n1; i++)
      L[i] = a[p + i];
    for (int j=0; j<n2; j++)
      R[j] = a[q + 1+ j];
    L[n1] = Integer.MAX_VALUE;
    R[n2] = Integer.MAX_VALUE;

    int i = 0;
```

```
    int j = 0;
    for (k=p; k<= r; k++){
      if(L[i] <= R[j]){
        a[k] = L[i];
        i++;
      }
      else{
        a[k] = R[j];
        j++;
      }
    }
  }

  public static void sort(int a[], int p, int r){
    if (p < r){
      int q = (p+r)/2;
      sort(a, p, q);
      sort(a , q+1, r);
      merge(a, p, q, r);
    }
  }
}
```

b) Implement a variant of MERGE-SORT which uses INSERTION-SORT to sort small instances (Lecture 5, slide 36). You may have to reimplement `InsertionSort` to rearrange in-place subarray A[p..r].

```
public static void sortCutOff(int a[], int p, int r, int n){

  if (r <= p + n - 1){
    InsertionSort.sort(a, p, r);
    return;
  }
  int q = p + (r - p) / 2;
  sort (a, p, q);
  sort (a, q+1, r);
  merge(a, p, q, r);
}
```

Reimplementation of INSERTION-SORT

```
public static void sort(int a[], int p, int r){
  for (int i = p + 1; i < r; i++){
    for (int j = i; j > p && a[j] < a[j-1]; j--){
      swap(a, j, j-1);
    }
  }
}
```

c) Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

```
static void merge(int a[], int p, int q, int r){
  int i, j, k;
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[] = new int[n1];
  int R[] = new int[n2];

  for (i = 0; i < n1; i++)
    L[i] = a[p + i];
  for (j = 0; j < n2; j++)
    R[j] = a[q + 1+ j];
```

```
    i = 0;
    j = 0;
    k = 1;
    while (i < n1 && j < n2){
      if (L[i] <= R[j]){
        a[k] = L[i];
        i++;
      }
      else{
        a[k] = R[j];
        j++;
      }
      k++;
    }

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

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

d) Implement bottom up MERGE-SORT (Lecture 5, slide 36). What is the complexity of this version of the algorithm?

```
public static void sort(int a[], int p, int r){
  int n = r - p;
  for (int sz = 1; sz < n; sz = sz+sz)
    for (int p = 0; p < n-sz; p += sz+sz)
      merge(a, p, p+sz-1, Math.min(p+sz+sz-1, n-1));
}
```

O(n log n) as the outer for loop does not increment the counter linearly.

e) Compare all the different versions of the algorithm (try experimenting with different cutoff values in 3b) using `TimeSortingAlgorithms` you implemented in Lab1. Use datasets provided in Lab/Files. Also compare with the running times of `SelectionSort` and `InsertionSort`. Explain your findings.

You should observe the asymptotical behavior of the various algorithms, i.e. quadratic for `SelectionSort` and `InsertionSort`, and n log n for `MergeSort`. Different versions of MERGE-SORT should only differ by constant factors when the inputs are sufficiently large. A plot of the running times against the input size could help visualise this.

Hints

(1) Use an accumulator as the second argument of `fact-tail`.

(2) Note: in practice, the Gamma function is used to compute factorials.

(3) Note: in practice, matrix exponentiation is used to compute Fibonacci sequences in O(log n).

(4) You can use `Integer.MAX_VALUE` for sentinels.

(5) Recall that bottom up MERGE-SORT is an iterative algorithm. This is how the algorithm operates on a sample input array [1,3,0,2,8,5,2,1]:
   - Merge subarrays of length 1 = [1,3,0,2,5,8,1,2]

- Merge subarrays of length 2 = [0,1,2,3,1,2,5,8]
- Merge subarrays of length 4 = [0,1,1,2,2,3,5,8]

(6) Also use the new file in the dataset (`intBig.txt`).