ALP III — 1. Aufgabenblatt

Tobias Lohse, Marvin Kleinert, Anton Drewing – Gruppe 1.8 – 22. Oktober 2014

Aufgabe 1

Lösung mit der Rechnungen mit Mathematica:

```
Alg1Time[n_] := (n*Log[n, 2])*3
Alg2Time[n_] := ( 1/2*n^2 - 1/2*n)*1
Alg3Time[n_] := (1/2*n!)*3/2
```

a)

```
TableForm[
 Table[N[Apply[f, {i}], 1] ns,
   {i, {10, 100, 1000, 10000}}, {f, {Alg1Time, Alg2Time, Alg3Time}}],
 TableHeadings -> {{"10", "100", "1000", "10000"},
   {“3n*log_2(n)”, “(n^2-n)/2", “3/2*n!”}}]
```

	$3n\cdot\log_2(n)$	$(n^2-n)/2$	$3/2\cdot n!$
10	9. ns	$5.\times 10^{1}$ ns	$3.\times 10^{6}$ ns
100	$5.\times 10^{1}$ ns	$5.\times 10^{3}$ ns	$7.\times 10^{157}$ ns
1000	$3.\times 10^{2}$ ns	$5.\times 10^{5}$ ns	$3.\times 10^{2567}$ ns
10000	$2.\times 10^{3}$ ns	$5.\times 10^{7}$ ns	$2.\times 10^{35659}$ ns

b)

```
s := 10^9; m := 60 s; h := 60*m; d := 24 h; y := (356 + 1/4) d

TableForm[
 Table[Abs@N[x /. FindInstance[i == f[x], x], 2],
   {i, {s, m, h, d, y}}, {f, {Alg1Time, Alg2Time, Alg3Time}}],
 TableHeadings -> {{"second", "minute", "hour", "day", "year"},
   {“3n*log_2(n)”, “(n^2-n)/2", “3/2*n!"}}]
```

	$3n\cdot\log_2(n)$	$(n^2-n)/2$	$3/2\cdot n!$
second	1.1×10^{10}	4.5×10^{4}	16.
minute	7.9×10^{11}	3.5×10^{5}	19.
hour	5.5×10^{13}	2.7×10^{6}	20.
day	1.5×10^{15}	1.3×10^{7}	21.
year	6.1×10^{17}	2.5×10^{8}	22.

Aufgabe 2

```java
import java.util.*;

public class a2 {

  static int[] rndArray(int size){
    int[] arr = new int[size];
    for ( int i=0; i<size; i++ ) {
      arr[i] = (int) (Math.random()*size*2);
    }
    return arr;
  }
```

a)

```java
  static boolean isSorted(int[] arr){
    for ( int i=1; i<arr.length; i++ ) {
      if ( arr[i-1] > arr[i] ) return false;
    }
    return true;
  }

  static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  static void permutationSort(int[] arr) {
    permutationSort(arr, 0);
  }
  static boolean permutationSort(int[] arr, int k) {
    for ( int i=k; i<arr.length; i++ ) {
      swap(arr, i, k);
      if ( permutationSort(arr, k+1) ) {
        return true;
      } else {
        swap(arr, k, i);
      }
    }
    if (k==arr.length-1) {
      if ( isSorted(arr) ) {
        return true;
```

```
      }
    }
    return false;
  }
  // Der Algorithmus wird mit dem Array der Länge n  aufgerufen und erstellt
  // zunächst alle Vertauschungen  der Stelle k mit den Stellen k <= i <= n.
  // Für jede dieser Vertauschungen ruft er sich rekursiv auf, um alle
  // Vertauschungen an der Stelle k+1 zu erstellen. Sobald der Algorithmus
  // einmal an der Stelle k = n angekommen ist, hat er eine Permutation
  // erzeugt. Nun prüft er, ob das Array bereits sortiert ist und gibt einen
  // entsprechenden Wahrheitswert zurück. Nachdem eine Rekursion abgearbeitet
  // ist, bricht der Algorithmus ab, wenn das Array bereits sortiert ist,
  // oder macht ein backtracking, so dass für den nächsten Schleifendurchlauf,
  // nur die Stellen i und k vertauscht sind.
```

b)

```
  static void shuffle(int[] arr){
    for ( int swaps=0; swaps<arr.length ; swaps++ ) {
      int i = (int) (Math.random()*arr.length);
      int j = (int) (Math.random()*arr.length);
      swap(arr, i, j);
    }
  }

  static void bogoSort(int[] arr) {
    int shuffles = 0;
    while ( !isSorted(arr) ) {
      shuffle(arr);
      shuffles++;
    }
  }
  // Der Algorithmus shuffled das Array der Länge n zunächst. Dafür wählt
  // er n mal zwei Zufallszahlen zwischen 1 und n aus und vertauscht diese
  // beiden Stellen im Array. Danach tested er, ob das geshuffelte Array
  // sortiert ist, wenn ja beendet er, wenn nein, wiederholt er die Prozedur.

  static long measureTimeBS(int[] arr) {
    int[] arrC = new int[arr.length];
    System.arraycopy(arr, 0, arrC, 0, arr.length);
    long time = System.nanoTime();
    bogoSort(arrC);
    time = (System.nanoTime()-time)/1000000;
    return time;
  }

  static long measureTimePS(int[] arr) {
    int[] arrC = new int[arr.length];
    System.arraycopy(arr, 0, arrC, 0, arr.length);
    long time = System.nanoTime();
    permutationSort(arrC);
    time = (System.nanoTime()-time)/1000000;
    return time;
  }

  // MAIN \\
  public static void main(String[] args) {
    for ( int i=6; i<=12 ; i+=3 ) {
      int[] array = rndArray(i);
      System.out.println("\nLenth: "+i);
      System.out.println(Arrays.toString(array));
      System.out.println("PermutationSort:\t"+measureTimePS(array)+"ms");
      for ( int j=0; j<10; j++ ) {
        System.out.println("BogoSort:\t\t"+measureTimeBS(array)+"ms");
      }
    }
  }
}
```

Aufgabe 3

Quick Schraube-Mutter Matching Algorithmus:

- Der Algorithmus wird mit allen Muttern M und Schrauben S aufgerufen.
- Wähle zufällig eine Schraube s_p aus S aus.
- Teste alle Muttern in M gegen diese Schraube s_p und teile M in zu kleinen Muttern M_l und zu großen Muttern M_g auf, halte die passende Mutter m_p fest.
- Vergleiche die passende Mutter m_p mit allen Schrauben in S und teile diese in zu kleine Schrauben S_l und zu große Schrauben S_g auf.
- Schraube die passende Mutter m_p auf die Schraube s_p.
- Rufe den Algorithmus rekursiv auf zum ersten mit den zu kleinen Muttern und Schrauben (M_l,S_l) und zum zweiten mit den zu großen Muttern und Schrauben (M_g,S_g).

Laufzeitanalyse des Algorithmus:

Die Anzahl der Vergleiche $C(n,k)$ bei $|M|=|S|=n$ Schrauben und Muttern, wenn zufällig die k-te Schraube als s_p ausgewählt wird, ist offensichtlich:

$$C(n,k) = C(n-k) + C(k-1) + 2*n - 1$$

Da die Schraube zufällig ausgewählt wird, ist die Wahrscheinlichkeit $p(k)$, dass Schraube k ausgewählt wird gegeben durch $1/n$. Damit ergibt sich die Anzahl der Vergleiche im Mittel zu:

$$\langle C(n)\rangle = \Sigma_{k=1}^{n} p(k) * C(n,k) = 1/n * (\Sigma_{k=1}^{n} C(n-k) + C(k-1)) + 2*n - 1$$

Mit der Formel aus der Vorlesung ergibt sich die Anzahl der Vergleiche dann mittels der harmonischen Reihe zu:

$$\langle C(n)\rangle \approx 1.38 * n*\log(n) + O(n) + n = 1.38 * n*\log(n) + O(n)$$