Informatik II - Übung 11 (Spoilerfree)

Pascal Schärli

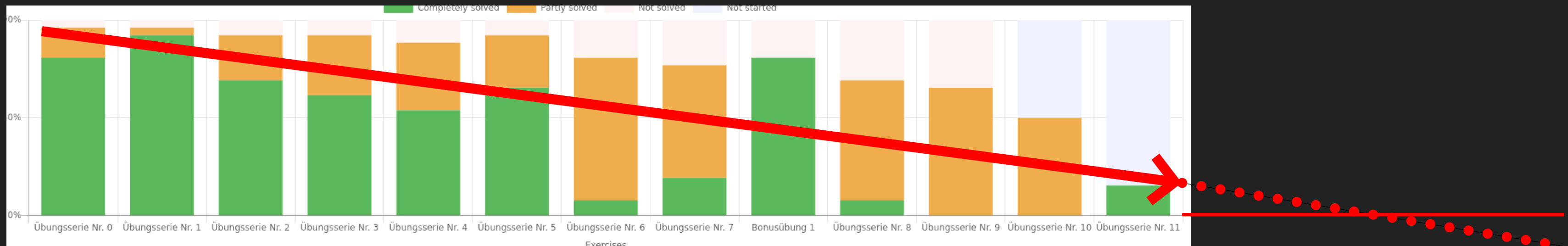
o(yeah)@pascscha.ch

06.12.2019

ETH zürich

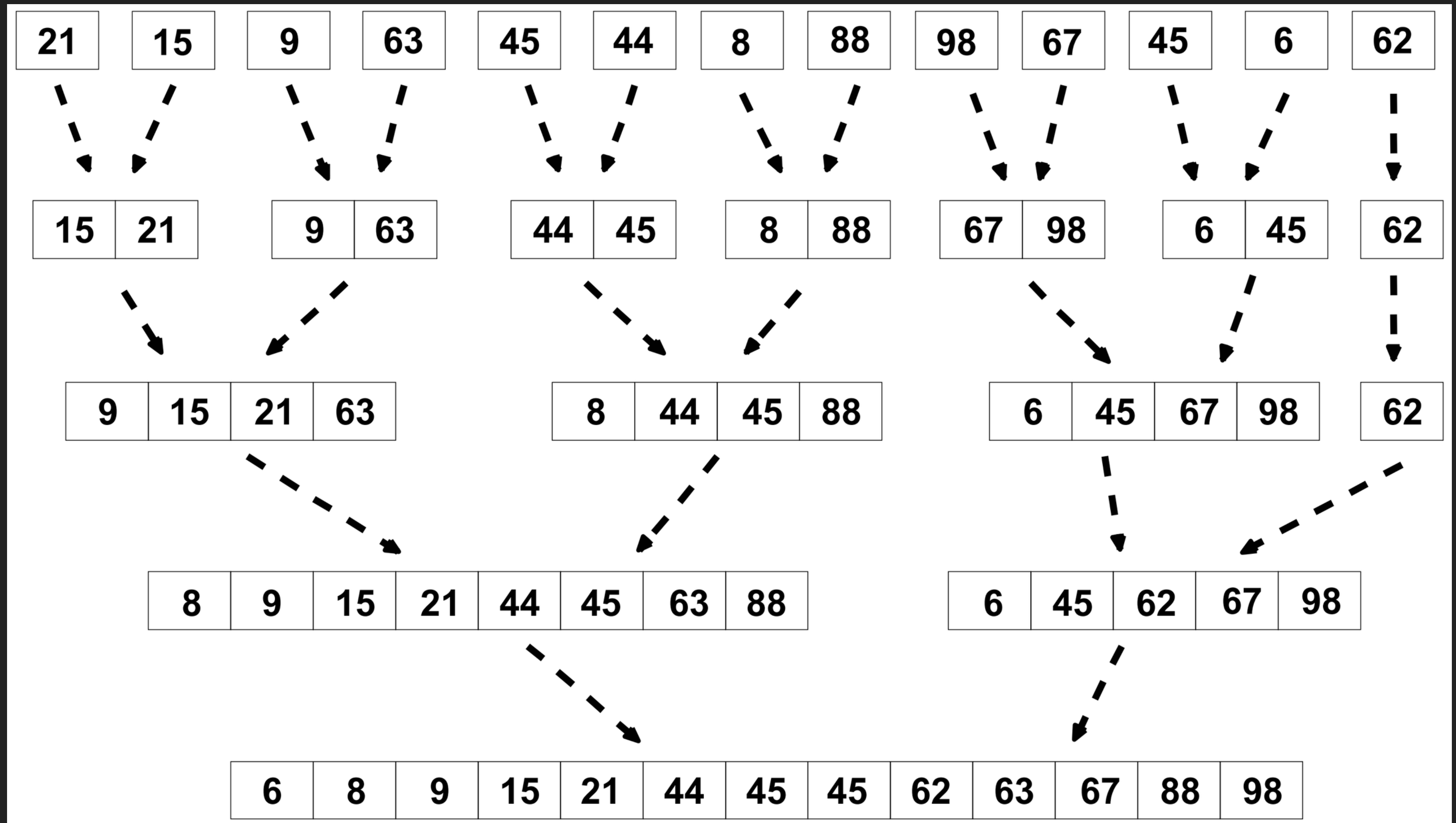
Nachbesprechung

Participation

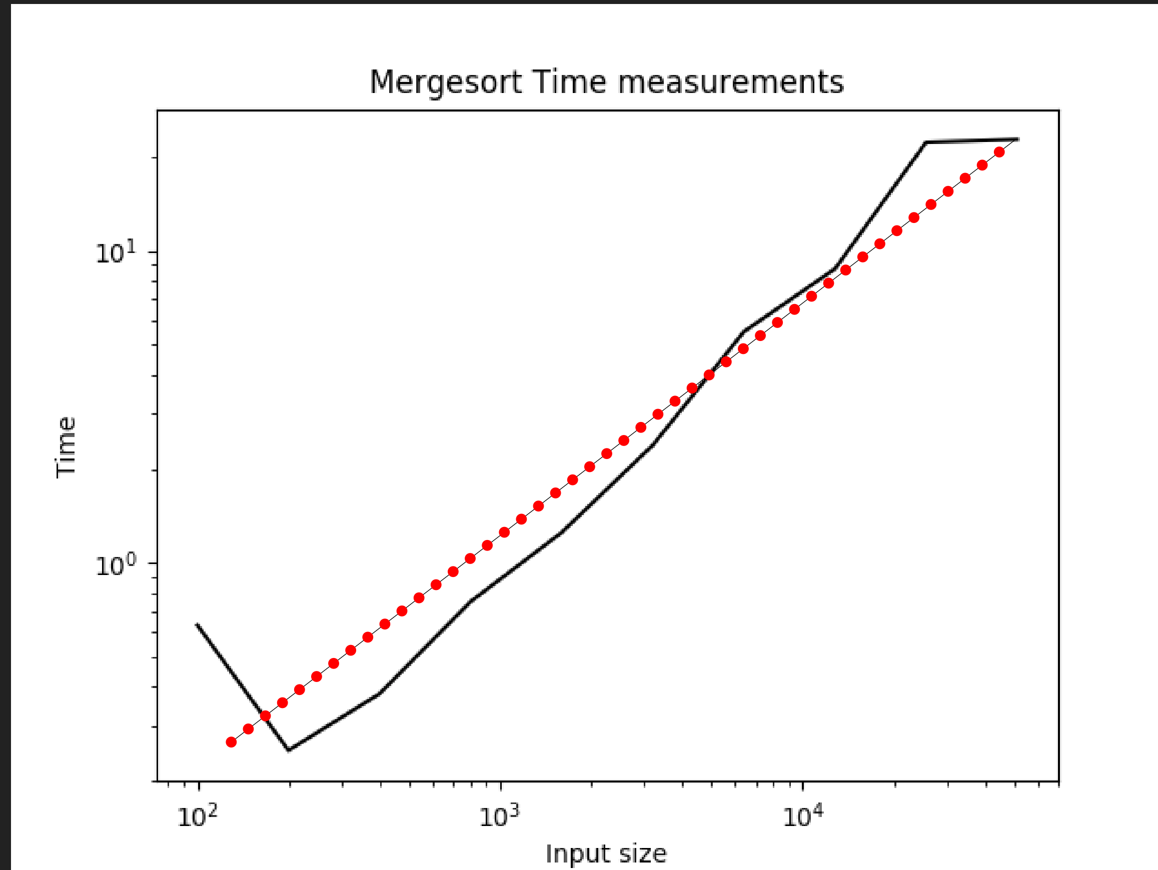


Die Programmieraufgaben lohnen sich wirklich schon unter dem Semester zu lösen, da es sehr schwierig ist, sich die Routine welche man dadurch bekommt in ein paar Wochen Lernphase anzueignen.

Mergesort Textaufgaben



Mergesort Textaufgaben



Mergesort Programmieren

```java
public ArrayList<T> sort(ArrayList<T> items) {
    return sortRec(items, 0, items.size());
}

private ArrayList<T> sortRec(ArrayList<T> items, int begin, int end) {
    if (begin == end) {
        return new ArrayList<T>();
    }
    if (begin + 1 == end) {
        ArrayList<T> result = new ArrayList<T>();
        result.add(items.get(begin));
        return result;
    }

    int middle = begin + (end - begin) / 2;
    ArrayList<T> left = sortRec(items, begin, middle);
    ArrayList<T> right = sortRec(items, middle, end);

    // [...]
```

Mergesort Programmieren

```
1      // [...]
2
3      int leftIdx = 0;
4      int rightIdx = 0;
5      ArrayList<T> sorted = new ArrayList<T>(end - begin);
6      while (true) {
7          if (leftIdx == left.size()) {
8              sorted.addAll(right.subList(rightIdx, right.size()));
9              return sorted;
10         }
11         if (rightIdx == right.size()) {
12             sorted.addAll(left.subList(leftIdx, left.size()));
13             return sorted;
14         }
15         if (left.get(leftIdx).compareTo(right.get(rightIdx)) < 0) {
16             sorted.add(left.get(leftIdx));
17             leftIdx += 1;
18         } else {
19             sorted.add(right.get(rightIdx));
20             rightIdx += 1;
21         }
22     }
23 }
```

Türme von Hanoi

1. Die Türme werden zyklisch in der Reihenfolge 3, 2, 1 nicht benutzt.

2.
```
1 moves = 2^4-1;
2 counter = 0;
3 while (counter < moves)
4   make available move between tower 1 and tower 2
5   make available move between tower 1 and tower 3
6   make available move between tower 2 and tower 3
7   increment counter by 3
```

3. Falls n ungerade ist, ändert sich die Reihenfolge, in der die Türme "nicht benutzt" werden:
   - n gerade: 3 – 2 – 1
   - n ungerade 2 – 3 – 1

Best of Vorlesung

Laufzeitkomplexität

- Beschreibt Anzahl Rechenschritte die ein Algorithmus für das Lösen in Abhängigkeit von der Inputgrösse n braucht
- Wir betrachten die **asymptotische** Laufzeit, also nicht wie gross der Zeitaufwand ist, sondern wie schnell er bei einem grösseren Input wächst
- Dies Abschätzungen machen wir mit der Gross-O-Notation
- Wir betrachten nur den am schnellsten Wachsenden Summanden und ignorieren alle konstanten Faktoren

Laufzeitkomplexität

```java
1 private static int f1(int n) {
2   int out = 0;
3
4   for(int i = 0; i < n; i++) {
5     out++;
6   }
7
8   return out;
9 }
```

Laufzeitkomplexität

```java
private static int f2(int n) {
  int out = 0;

  for(int i = 0; i * 5 < n; i++) {
    out++;
  }

  return out;
}
```

Laufzeitkomplexität

```java
private static int f3(int n) {
  int out = 0;

  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++){
      out++;
    }
  }

  return out;
}
```

Laufzeitkomplexität

```java
private static int f4(int n) {
  int out = 0;

  for(int i = 0; i < n; i++) {
    for(int j = 0; j < 1000; j++){
      out++;
    }
  }

  return out;
}
```

Laufzeitkomplexität

```java
private static int f5(int n) {
  int out = 0;

  for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++){
      out++;
    }
  }

  return out;
}
```

Laufzeitkomplexität

- Wir haben ein Computer, welcher ein Problem mit $O(f(n))$ der Grösse M in der Zeit t lösen kann.

- Was ist die lösbare Problemgrösse, falls wir k-Mal so viel Zeit zur Verfügung haben?

- Beispiel:
  - Der Computer kann ein $O(n^3)$ Problem kann in 10s einen Input der Grösse 1'000 verarbeiten

  - Wie gross kann der Input sein, wenn man 20s zur Verfügung hat?

Laufzeitkomplexität

- Komplexität $O(n^3)$
- Bisherige Inputgrösse 1'000
- Mehr Zeit: $\times 2$

1. Der Computer macht ca $1000^3 = 10^9$ Operationen in 10s
2. In 20s Zeit kann er also $2 \cdot 10^9$ Operationen machen
3. Die neu bewältigbare Problemgrösse ist somit $\sqrt[3]{2 \cdot 10^9} \approx 1260$

Laufzeitkomplexität

- Komplexität $O(f(n))$
- Bisherige Inputgrösse M
- Mehr Zeit: $\times k$

1. Der Computer macht ca $f(M)$ Operationen
2. Mit der zusätzlichen Zeit kann er also $k \cdot f(M)$ Operationen machen
3. Die neu bewältigbare Problemgrösse ist somit $f^{-1}(k \cdot f(M))$

Vorbesprechung

Sortieren mit Suchbäumen

- Wie können binäre Suchbäume zum Sortieren von Listen verwendet werden?

- Welche Listen eignen sich am besten zum Sortieren mit Suchbäumen?
  1. Aufsteigend vorsortiert
  2. Absteigend vorsortiert
  3. Gut durchmischt

- Was ist die Laufzeitkomplexität im:
  - besten Fall
  - durchschnittlichen (gut durchmischten) Fall
  - schlechtesten Fall

Komplexitätsanalyse und O-Notation

- Was ist die Komplexität dieser Codefragmente?
- Überlegt euch wie häufig a++ ausgeführt wird

```
1  // Fragment 1
2  for (int i=1; i<n; i++) a++;
3
4  // Fragment 2
5  for (int i=0; i<2*n; i++) a++;
6  for (int j=0; j<n; j++) a++;
7
8  // Fragment 3
9  for (int i=0; i<n; i++)
10    for (int j=0; j<n; j++)
11        a++;
12
13 // Fragment 4
14 for (int i=0; i<n; i++)
15     for (int j=0; j<i; j++)
16         a++;
17
18 // Fragment 5
19 while(n >= 1) n = n/2;
20
21 // Fragment 6
22 for (int i=0; i<n; i++)
23     for (int j=0; j<n*n; j++)
24         for (int k=0; k<j; k++)
25             a++;
```

Komplexität

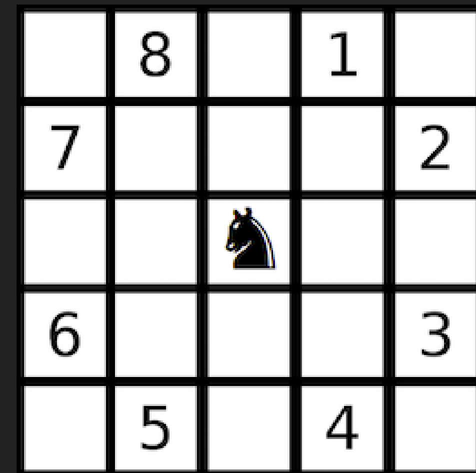
- Ihr habt ein Computer, welcher bisher ein Problem der Grösse M lösen konnte.
- Wie gross kann M' sein, wenn der Computer neu dreimal so schnell arbeitet.
- Wie vorher bereits gelöst, nun einfach mit schnellerem Computer anstatt mit mehr Zeit.

Komplexität

- Ihr habt ein Computer, welcher bisher ein Problem der Grösse M lösen konnte.
- Wie gross kann M' sein, wenn der Computer neu dreimal so schnell arbeitet.
- Wie vorher bereits gelöst, nun einfach mit schnellerem Computer anstatt mit mehr Zeit.

Ein Springer auf dem Schachbrett

- Im Schach bewegt sich ein Springer entweder um zwei Felder horizontal und ein Feld vertikal oder um ein Feld horizontal und zwei Felder vertikal.
- Dieses Sprungmuster führt zu ein paar interessanten Fragen, denen Ihr in dieser Aufgabe nachgehen sollt.



Ein Springer auf dem Schachbrett

- Positionen auf dem Schachbrett werden dabei als Objekte vom Typ Position gespiechert.

- Ihr könnt zwei Positionen addieren:

```
Position pos1plus2 = pos1.add(pos2);
```

- Es macht Sinn, alle möglichen Züge vom Springer in einer Liste zu speichern:

```
possibleMoves = new ArrayList<Position>(8);
possibleMoves.add(new Position(1, 2));
possibleMoves.add(new Position(2, 1));
// … usw
```

Ein Springer auf dem Schachbrett

Aufgabe 1: getReachableSet

- Die Funktion getReachableSet soll eine Menge von Felder zurückgeben, welche ausgehend von einer Startposition *innerhalb* von einer gegebenen maximalen Anzahl von Schritten erreicht werden können.

- Helferfunktion:

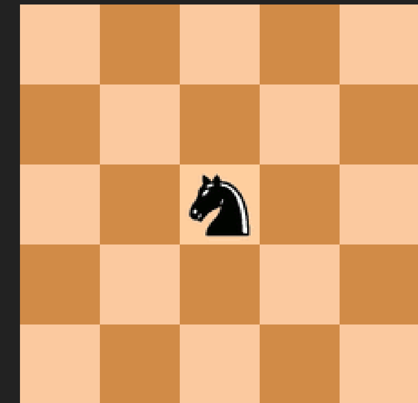
```
visit(Positoin pos, int maxDepth, int depth, ArrayList<Position> visited)
```

1. Fügt falls nötig pos visited hinzu
2. Prüft ob die maximale Tiefe erreicht ist
3. Probiert alle gültigen Züge aus und ruft visit rekursiv auf

Ein Springer auf dem Schachbrett

Aufgabe 2: completePath

- Implementiert die Funktion findCompletePath, welche zu einer gegebenen Startposition einen Pfad über das Schachbrett zurück gibt
- Bei diesem Pfad sollen alle Felder genau einmal besucht werden
- Verwendet Backtracking, ähnlich wie beim Rucksackproblem



Viel Spass!