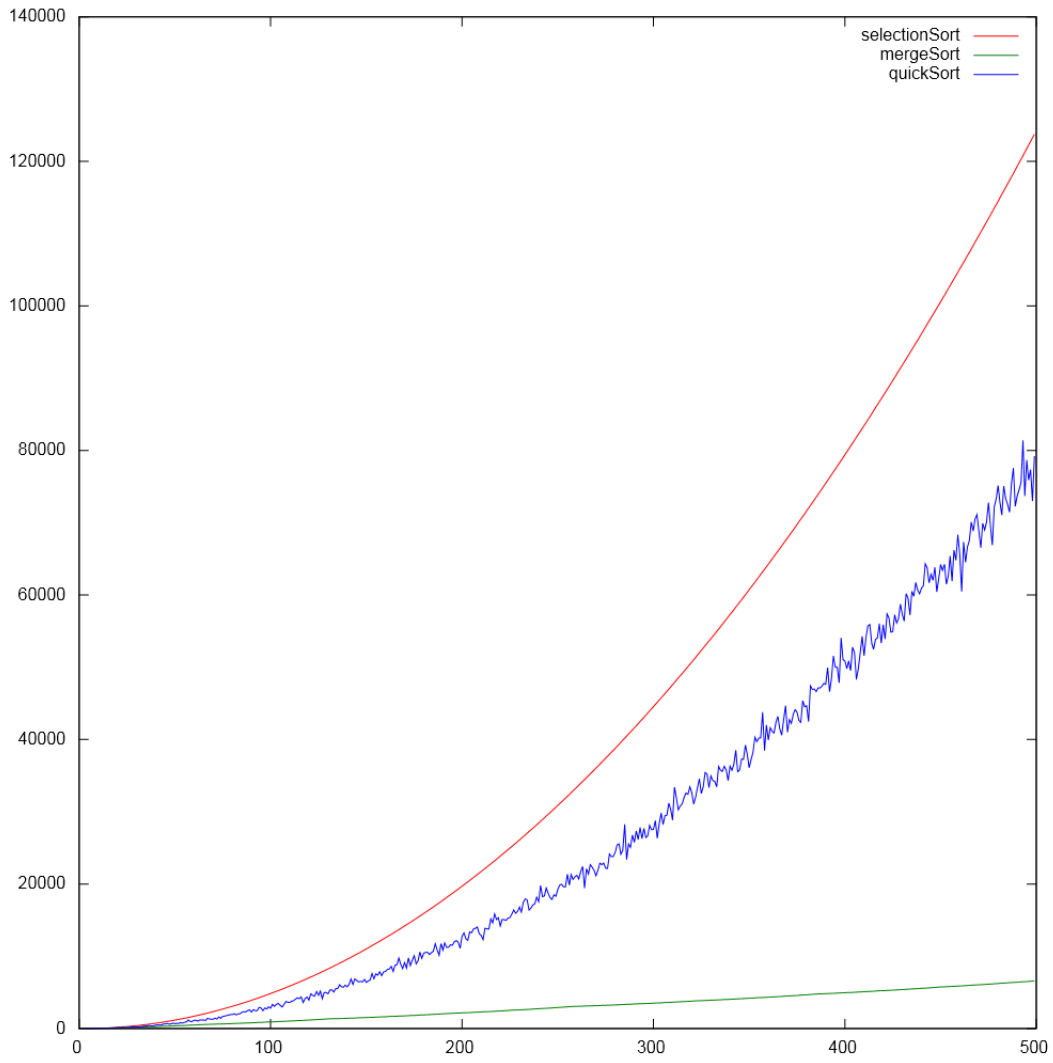
Übung 2 - Musterlösung Abgabe 3.5.2012

Aufgabe 1 – Vergleichen und Testen von Sortierverfahren 24 Punkte

a) Eine Möglichkeit der Implementation der Algorithmen befindet sich im Anhang. Das Plotten kann durch Gnuplot nach Ausführen von `sort.py` folgendermaßen geschehen:

```
gnuplot> set datafile separator ","
gnuplot> plot [0:500][0:140000] '/home/YOU/varyN.txt' using 1:2 with lines title
'selectionSort',\
>'/home/YOU/varyN.txt' using 1:3 with lines title 'mergeSort',\
>'/home/YOU/varyN.txt' using 1:4 with lines title 'quickSort'
gnuplot> set terminal postscript eps color solid
gnuplot> set output 'nVary.ps'
gnuplot> replot
```

Damit erhält man dann folgende Grafik:



Wie man sieht kommt man für `selectionSort()` gut an die quadratische Näherung – während `quickSort()` sehr unregelmäßig ist, da die Effizienz sehr stark von der Beschaffenheit des Anfangsarrays abhängt.

Das Fitting kann man so durchführen:

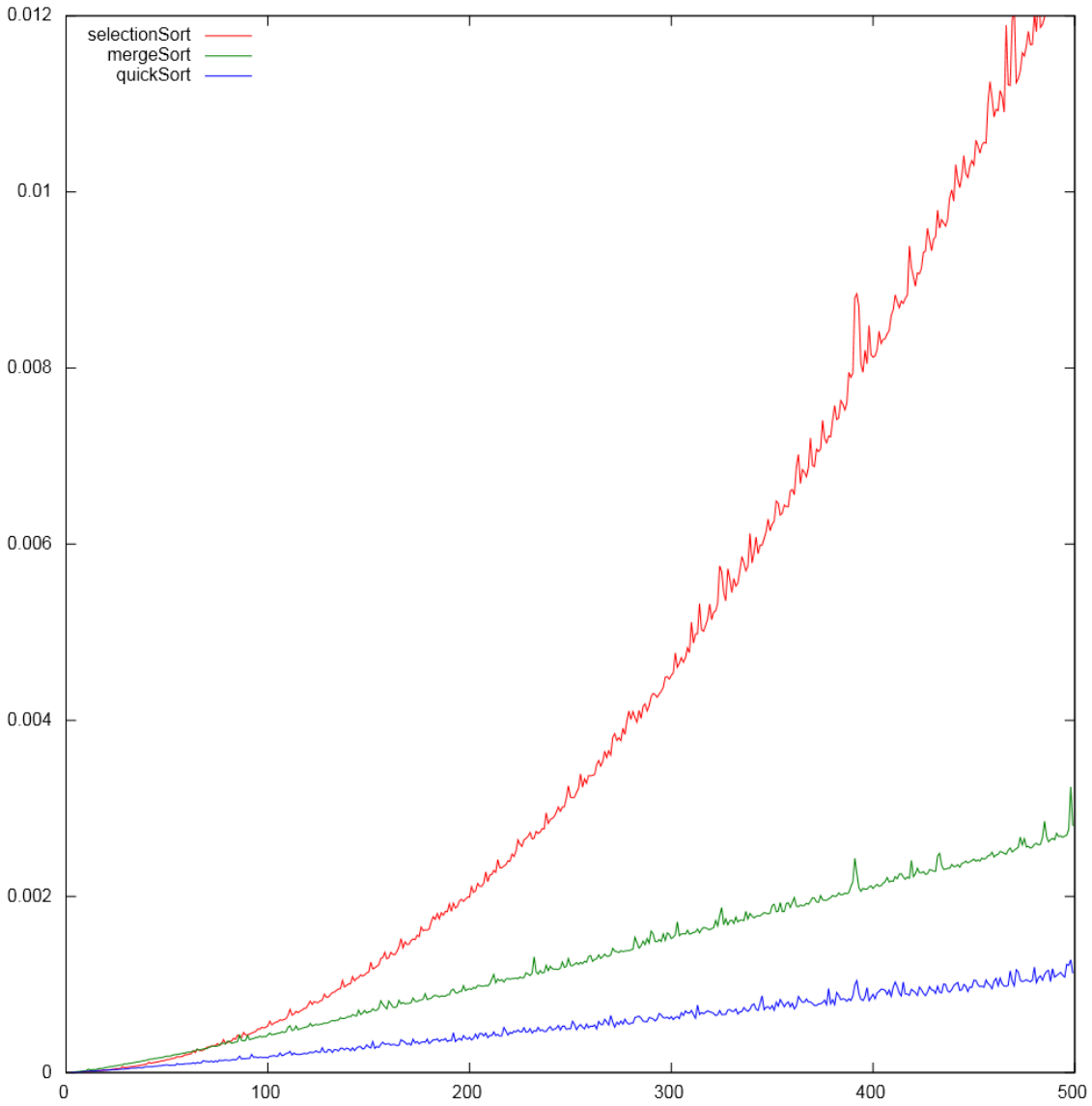
```
gnuplot> set datafile separator ","
gnuplot> f(x) = a*x**2 + b*x + c
gnuplot> g(x) = d*x*log(x)/log(2) + e*x + f
gnuplot> h(x) = g*x*log(x)/log(2) + h*x + i
gnuplot> fit f(x) '/home/YOU/varyN.txt' using 1:2 via a,b,c
gnuplot> fit g(x) '/home/YOU/varyN.txt' using 1:3 via d,e,f
gnuplot> fit h(x) '/home/YOU/varyN.txt' using 1:4 via g,h,i
```

Damit kommt man dann auf folgende Variablen:

$$a, b, c = \quad 0.5\,, \quad -1.5\,, 1$$
$$d, e, f = 1.63744\,, -1.70546\,, 8.74094$$
$$g, h, i = 82.4639\,, -602.685\,, 6760.65$$

Wie man sieht, weichen die `quickSort()` Parameter besonders stark von der Norm ab.

b) Das ist der Fall: Die funktionelle Form bleibt erhalten, es ändern sich lediglich die Konstanten, wie man an folgendem Plot sieht:

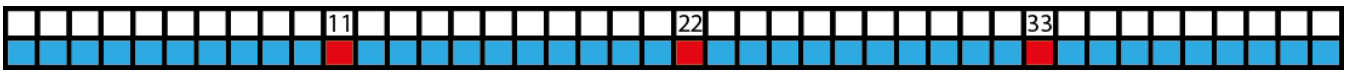


Wie von Hand gezeichnet.

Aufgabe 2 – Gewinnalgorithmus für ein Spiel 16 Punkte

a) Nach einigen Spielversuchen stellt man fest, dass es Konstellationen gibt, aus denen heraus man gegen einen denkenden Gegner nicht gewinnen kann. Durch eine kleine Analyse wird schnell klar, welche Konstellationen dies sind:

Die Begründung ist einfach: Wenn nur noch 1-10 Hölzer vorhanden sind und man selbst am Zug ist, kann man alle Hölzer und somit auch das Letzte nehmen. Liegen hingegen 11 Hölzer kann man nichts weiter tun, als 1 – 10 Hölzer zu nehmen und dem Gegner somit den Sieg zu gewähren. Ein Strategievorschlag wäre also: Zwinge den Gegner auf das „Feld" 11. Feld 11 ist von 12 – 21 zu erreichen – somit wäre mit diesen Felder als Startposition der Sieg auch gewiss. Ein weiteres ungünstiges Feld ist somit 22, da man von hier aus dem Gegner die Möglichkeit gibt, einen auf Feld 11 zu zwingen....setzt man diese Logik fort, so ergibt sich folgendes Bild:



Daraus ergibt sich die allgemeine Strategie, den Gegner schlichtweg auf eines dieser Felder zu zwingen: Ist der Gegner einmal auf einem rot gekennzeichneten Feld, kann man ihn immer wieder auf das nächste rote Feld – und somit zur Niederlage – zwingen.

b) Die allgemeine Strategie ist im Wesentlichen oben schon gegeben. Eine einfach Regel für die Wahl der Anzahl der zu ziehenden Hölzer gewinnt man, indem man sich bewusst macht, dass man den Gegner immer aufs nächste rote Feld drängen muss. Man kann also folgende Formel aufstellen:

$$Anzahl\ der\ zu\ ziehenden\ Hölzer = (position)\ mod\ (p+1)$$

„mod" entspricht hier der Modulo – Operation (http://en.wikipedia.org/wiki/Modulo_operation)

Diese Formel bietet nur sinnvolle Antworten (also im Rahmen von 1 bis p), wenn die Startposition kein Vielfaches von (p+1) ist, dann befindet man sich nämlich auf einem roten Feld und kann nur hoffen, dass der Gegner Alda I nicht gehört hat.

c) Die Strategie folgt aus den selben Überlegungen, nur wird um Eins aufgerückt: Das erste Feld, auf das man den Gegner drängen will, ist die 1. Somit hat man dann die 12, die 23, die 34 usf. Die Formel lautet dann:

$$Anzahl\ der\ zu\ ziehenden\ Hölzer = (position - 1)\ mod\ (p+1)$$

Bonusaufgabe – Dynamisches Array mit verringertem Speicherverbrauch 10 Punkte

a) Die Gesamtzahl der Leerstände setzt sich aus den noch nicht besetzten Pointern und den nicht besetzten Plätzen im zuletzt angehängten Subarray zusammen. Anschaulich gilt es die None - Einträge im ungünstigsten Fall (direkt nach der Vergrößerung der Kapazität) zu zählen. Daraus ergibt sich folgende Berechnung:

$$n_{Subarray} + n_{Pointer} = p - 1 + \frac{p}{2} \approx 1.5 * p = 1.5 * \sqrt{C} < 2 * \sqrt{C}$$

Wobei sich die Abschätzungen durch folgende Überlegungen ergeben:

$$n_{Subarray} = p - 1 \; (worste\ case: nur\ ein\ Platz\ besetzt)$$

$$n_{Zeilen\ besetzt} = \frac{C_{alt}}{p_{neu}} = \frac{(p_{old})^2}{p_{neu}} = \frac{(\frac{1}{\sqrt{2}} p_{neu})^2}{p_{neu}} = \frac{p_{neu}}{2}$$

b) Die einzelnen Formeln ergeben sich zum Beispiel aus Betrachtung der folgenden Matrix (die erste Spalte symbolisiert Pointer):

$$\begin{pmatrix} \rightarrow & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ \rightarrow & a_{21} & a_{22} & a_{23} & a_{24} & \text{None} & \text{None} \\ \text{None} & & & & & & \\ \text{None} & & & & & & \\ \text{None} & & & & & & \\ \text{None} & & & & & & \end{pmatrix}$$

Will man aufs 8. Element zugreifen, muss man also auf das 2. Element im 2. Unterarray zugreifen. Die Kapazität des Speichers ist 36, und damit die Dimension der Matrix p = 6. Immer wenn das letzte Unterarray voll ist, ist die Anzahl der Elemente somit ein Vielfaches von 6. Aus diesen einfach Beispielen kann man die geforderten Formeln ableiten:

(1)

```
row = k / p      # Integer-Division mit automatischem Abrunden
column = k % p   # Rest-Berechnung mit Modulo-Operator
element = array[row][column]
```

(2)

```
if self.size % p == 0:
      … # add new subArray
```

(3)

```
if self.size == self.capacity:
     p = max(int(round(sqrt(2)*p)), p+1) # p+1 is needed for p==1
     self.capacity = p**2
```

erzeugt: capacity = p^2 = { 1, 4, 9, 16, 36, 64, 121, …}

c) Eine mögliche Implementation befindet sich im Anhang unter `array.py`.

d) Ein Nachteil ist zum Beispiel, dass durch das bedarfsorientierte Einfügen der Zeilen die Zellen nicht mehr nacheinander im Speicher abelegt werden können. Hierdurch verteuert sich der Zugriff, weil die Gefahr von Cache-Misses größer ist. Auch die Berechnung der Zeilen- und Spaltenindizes (eine Division, eine Modulo-Berechnung) ist teurer als der einfache Indexzugriff im gewöhnlichen dynamischen Array.

Anhang – sort.py

```
import copy
from timeit import Timer
from random import randint


def selectionSort(inList):

    count = 0

    #iterate through all elements from left to right
    for i in range(len(inList)):

        #set swap index to refer to the i-th element
        k = i

        #iterate through list right of i
        for j in range(i, len(inList)):

            #find smallest element in this range
            if inList[j] < inList[k]:

                #remember it
                k = j

            #iterate count
            count += 1

        #and swap with the first element
        inList[i], inList[k] = inList[k], inList[i]

    return count


def merge(left, right):

    count = 0

    #initialize result list
    result = []

    #while both of them are not empty
    while len(left) > 0 and len(right) > 0:

        #compare the first element of each list, then pop the smaller one
        #and append it at the result list
```

```
            if left[0] > right [0]:
                  result.append(right.pop(0))
            else:
                  result.append(left.pop(0))

            #iterate count
            count += 3

      #append the rest of the remaining not empty list
      result.extend(right)
      result.extend(left)

      return result, count


def mergeSort(inList):

      count = 0

      #if its just one element or empty, return it
      if len(inList) <= 1:
            return inList, 0

      else:
            #otherwise split it in half
            left = inList[:len(inList) / 2]
            right = inList[len(inList) / 2:]

            #and do the same with the resulting two lists.
            #meanwhile, collect all counts
            leftSorted, c = mergeSort(left)
            count += c
            rightSorted, c = mergeSort(right)
            count += c

      result, c = merge(leftSorted, rightSorted)
      count += c

      #return the merged, sorted list and the accumulated sort count
      return result, count


def quickSort(inList):

      count = 0

      #if its just one element or empty, return it
      if len(inList) <= 1:
            return inList, 0

      else:
            #select the first element as the pivot element
            pivot = inList.pop(0)
```

```
        #initialize less and greater lists
        less, greater = [], []

        #append all elements lesser than pivot  to less and
        #append all elements greater than pivot to greater
        for i in inList:
            if i < pivot:
                less.append(i)
            else:
                greater.append(i)
            #iterate count
            count += 1

        #do the same to the resulting less and greater lists
        #and collect counts
        lessSorted, c = quickSort(less)
        count += c
        greaterSorted, c = quickSort(greater)
        count += c

        #return a list concatenated in the following order and the count
        result = lessSorted + [pivot] + greaterSorted
        return result, count


def checkSort(inList, outList):

    #make a copy of the out list
    outListc = copy.deepcopy(outList)

    #compare lenghts
    if len(inList) != len(outList):
        return False

    #check for all elements to be the same
    for x in inList:
        try:
            outListc.remove(x)
        except:
            return False

    #check for all elements being sorted properly
    for i in range(len(outList) - 1):
        if outList[i] > outList[i + 1]:
            return False

    #all tests passed, return True
    return True


#*****************************************************************************#
#         Apply algorithms and save the collected data to files               #
#*****************************************************************************#
```

```
testDir = '/home/YOU/'

#open file with write access
f = open(testDir + 'varyN.txt', 'w')

#check the count of sort algorithms for lists
#between length 2 and 500
for i in range(2, 500):

     #create random list
     a = [];
     for j in range(2, i):
          a.append(randint(2, i))

     #and two copies -> 3 sorting algorithms
     b = copy.copy(a)
     c = copy.copy(a)

     #create a string containing information about
     #the respective counts, write it to file
     f.write(str(i) + "," + \
               str(selectionSort(a)) + "," + \
               str(mergeSort(a)[1]) + "," + \
               str(quickSort(a)[1]) + "\n")

#close file
f.close()

#open file with write acess
f = open(testDir + 'varyT.txt', 'w')

#check the duration of sort algorithms for lists
#between length 2 and 500
for i in range(2, 500):

     #create a random list
     a = [];
     for j in range(2, i):
          a.append(randint(0, i))

     #and two copies -> 3 sorting algorithms
     b = copy.copy(a)
     c = copy.copy(a)

     #create a string containing information about
     #the respective durations, write it to file
     f.write(str(i) + ", " + \
          str(Timer("selectionSort(" + str(a) + ")", 'from __main__ import selectionSort').timeit(1))+"," + \
          str(Timer("mergeSort(" + str(b) + ")", 'from __main__ import mergeSort').timeit(1)) + ", " + \
          str(Timer("quickSort(" + str(c) + ")", 'from __main__ import quickSort').timeit(1)) + "\n");

#close file
f.close()
```

Anhang – array.py

```
from math import ceil, sqrt

class DynamicArray(object):

    def __init__(self):

        # initialize data container, size, capacity and p. It holds p^2=capacity.
        # the data structure can be visualized as a matrix
        self.data = [[None]]
        self.size = 0
        self.capacity = 1
        self.p = 1

    def append(self, item):

        # if matrix is full, extend it
        if self.size == self.capacity:

            # recalculate capacity and p, remember pOld
            pOld = self.p
            self.p = max(int(round(sqrt(2) * self.p)), p+1)
            self.capacity = self.p ** 2

            # remember old data, reallocate self.data with p "rowpointers"
            oldData = self.data
            self.data[row] = [None] * self.p

            # copy element-wise from old matrix into the new matrix
            for i in range(self.size):
                rowNew = i / self.p
                colNew = i % self.p
                rowOld = i / pOld
                colOld = i % pOld
                if self.size == self.capacity:
                    self.data[row] = [None] * self.p # allocate next row
                    self.data[rowNew][colNew] = tmp[rowOld][colOld]

        # row, column of the item in the CURRENT matrix
        row = self.size / self.p
        col = self.size % self.p

        # if row is full, create the next row
        if col == 0:
            self.data[row] = [None] * self.p

        # assign new item to its cell in the matrix
        self.data[row][col] = item

        # increment size
        self.size += 1
```

```
        # implement support for 'x = a[i]'
        def __getitem__(self, index):

            # check for valid index
            if index < 0 or index >= self.size:
                raise IndexError('DynamicArray: Invalid index')
            return self.data[index / self.p][index % self.p]

        # implement support for 'a[i] = x'
        def __setitem__(self, index, item):

            # check for valid index
            if index < 0 or index >= self.size:
                raise IndexError('DynamicArray: Invalid index')
            self.data[index / self.p][index % self.p] = item

        # implement support for 'len(a)'
        def __len__(self):
            return self.size

        # implement support for 'print a'
        def __str__(self):
            return "\n".join([str(i) for i in self.data])

# make array.py executable, add a little demonstration and test
if __name__ == "__main__":
    a = DynamicArray()
    desiredSize = 100
    print "appending", desiredSize, elements"
    for i in range(desiredSize):
        a.append(i)
        print 'Size: ', a.size
        print 'Capacity', a.capacity
    print 'a[0],a[50],a[73]: ', a[0], a[50], a[73]
    print "testing all entries:",
    assert len(a) == desiredSize
    for i in range(desiredSize):
       assert a[i] == i
    print "OK"
    a[23] = 42
    assert a[23] == 42
```