Elementare Algorithmen und Objektorientierte Programmierung

Übung 2

Auszuarbeiten bis 25.03.21

1. Feinanalyse (6 Punkte)

Zur Berechnung der *n*-ten Potenz einer Zahl *x* (also $p = x^n$) schlägt jemand den folgenden einfachen Algorithmus vor:

```
simplePower (↓x ↓n ↑p ) {
   p = 1;
   for(i = 1; i <= n; i++) {
      p = p * x
   }
}
```

Um die Berechnung schneller zu machen, schlägt ein anderer den folgenden komplizierteren Algorithmus vor:

```
quickPower (↓x ↓n ↑p ) {
   p = 1
   while(n > 0) {
      if (n mod 2 != 0) {
         p = p * x
      }
      n = n / 2
      x = x * x
   }
}
```

a) Beschreiben Sie die beiden Verfahren zum Potenzieren. Warum "verdient" der zweite Algorithmus seinen Namen?

b) Führen Sie für beide Algorithmen eine Feinanalyse durch

c) Welcher Algorithmus ist unter welchen Bedingungen schneller

Hinweise: Verwenden Sie für die Berechnung die in der Vorlesung vorgestellten Ausführungszeiten der einzelnen Anweisungen. Die *modulo* Operation benötigt 5 Zeiteinheiten, *++* 1.8 Zeiteinheiten, *Multiplikation* 3 Zeiteinheiten

2. Grobanalyse (4 + 3 + 3 Punkte)

Entwickeln Sie die folgenden Funktionen und analysieren Sie diese infolge.

Bestimmen Sie den Best, Worst und Average-Case (sofern möglich)

`int findCharLeft(const char* str, char ch)` bestimmt die Position des ersten Zeichens von links mit dem Wert *ch* in der Zeichenkette *str*.

`int findCharRight(const char* str, char ch)` bestimmt die Position des ersten Zeichens von rechts mit dem Wert *ch* in der Zeichenkette *str*.

`int findCharRandom(const char* str, char ch)` bestimmt die Position des ersten Zeichens mit dem Wert *ch* in der Zeichenkette *str* anhand zufällig ausgewählter Positionen.

Implementieren Sie die Funktionen in C. Achten Sie auf eine möglichst effiziente Implementierung.

3. **O-Notation (4 Punkte)**

In der Vorlesung wurden die unterschiedlichen Komplexitätsklassen besprochen. Suchen Sie in „echten“ Quellen für die Komplexitätsklassen O(1), O(n), O(log(n)) und $O(n^2)$ jeweils zwei Algorithmen. Überlegen Sie sich auch eine Begründung für die Einordnung in die jeweilige Klasse.

1 Feinanalyse

a)

simplePower
Hier wird die Variable **p** einfach zu der Variable **x** fortlaufend multipliziert bis wir die gewünschte Potenz erreicht haben.

quickPower
Hier wird das ganze mit dem log2(n) verfahren gemacht. Da wird solange durch eine Schleife durch iteriert bis **n** <= 0 ist. **N** wird bei jeder Iteration halbiert und **x** mit sich selber multipliziert, daher auch das logarithmische verfahren. Die Variable **p** wird wenn **n** mod 2 != 0 ist mit **x** multipliziert.

quickPower hat seinen namen verdient weil er logarithmisch wächst. Ab einer gewissen Problemgröße ist er um einiges schneller.

b)

Feinanalyse siehe beigelegtes excel file.

c)

Ab einer Problemgröße von n >= 12 sollte **quickPower** schneller sein. Bei n < 12 ist **simplePower** schneller, da weniger Operationen pro Durchgang ausgeführt werden. **quickPower** ist bei größeren Problemen schneller da er logarithmisch wächst und somit die Konstanten nicht mehr ins Gewicht fallen.

2 Grobanalyse

findCharLeft
Hier wird von links nach rechts der string durch iteriert bis der passende Character gefunden wurde.

Testfälle:
(“abc”, ‘b’) (“abc”, ‘a’) (“abc”, ‘c’) (“abc”, ‘d’) (“”, ‘i’) (“ewniobpmwEDasdgkRLÖPQAM“, ‘k’)

Best, average und worst case im c.file beschreiben.

findCharRight
Hier wird zuerts die string länge gezählt, dannach wird von rechts nach links durch iteriert bis der passende Character gefunden wurde.

Testfälle:
(“abc”, ‘b’) (“abc”, ‘a’) (“abc”, ‘c’) (“abc”, ‘d’) (“”, ‘i’) (“ewniobpmwEDasdgkRLÖPQAM“, ‘k’)

Best, average und worst case im c.file beschreiben.

findCharRight_2
Hier wird von links nach rechts durch iteriert bis der letzte passende Character gefunden wurde.

Testfälle:
(“abc”, ‘b’) (“abc”, ‘a’) (“abc”, ‘c’) (“abc”, ‘d’) (“”, ‘i’) (“ewniobpmwEDasdgkRLÖPQAM“, ‘k’)

Best, average und worst case im c.file beschreiben.

findCharRandom
Hier wird zuerst die string länge gezählt, dannach wird der zu prüfende Index mit einer random funktion bestimmt, dies wird solange gemacht bis der gesuchte Character gefunden ist. Bei 1000 versuchen wird aber abgebrochen und eine Fehlermeldung ausgegeben.

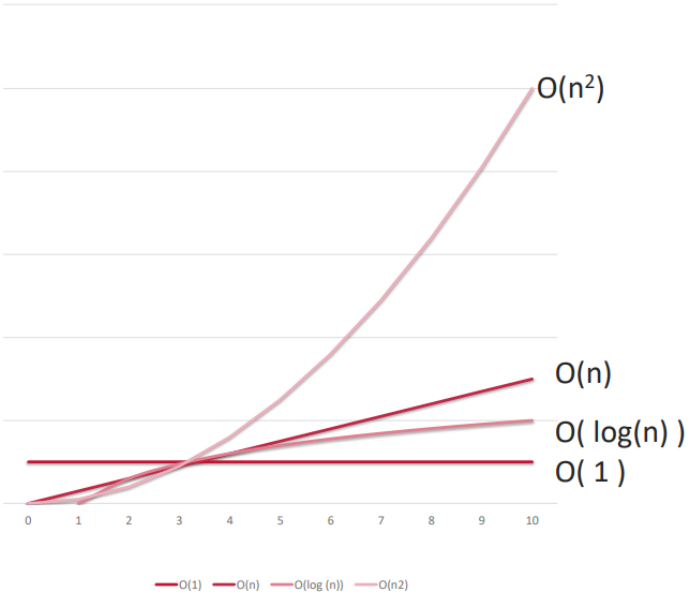
Testfälle:
(“abc”, ‘b’) (“abc”, ‘a’) (“abc”, ‘c’) (“abc”, ‘d’) (“”, ‘i’) (“ewniobpmwEDasdgkRLÖPQAM“, ‘k’)

Best, average und worst case im c.file beschreiben.

3 O-Notation

In der Vorlesung wurden die unterschiedlichen Komplexitätsklassen besprochen. Suchen Sie in „echten“ Quellen für die Komplexitätsklassen O(1), O(n), O(log(n)) und O(n²) jeweils zwei Algorithmen. Überlegen Sie sich auch eine Begründung für die Einordnung in die jeweilige Klasse.

O(1)	Ideal
O(lg n)	Sehr gut
O(n)	Zufrieden stellend
$O(n^2)$	Unerwünscht
$O(a^n)$	Katastrophal



O(1)

Mit O(1) ist eine konstante laufzeit eines Algorithmus gemeint.

Beispiele:

```
int getValueOfArr(int values[], int index) {
    return values[index];
}
```

```
void switchChar(char str[], int index1, int index2) {
    char temp = str[index1];
    str[index1] = str[index2];
    str[index2] = temp;
}
```

O(n)

Mit O(n) ist eine lineare laufzeit eines Algorithmus gemeint.

Beispiele:

```
int getLengthOfString(const char* str) {
    int i = 0;

    while (str[i] != '\0') {
        i++;
    }

    return i;
}
```

```
int arithmeticAverage(int valueCount, int values[]) {
    printf("\nvalueCount: %d\n", valueCount);

    if (valueCount > 0) {
        int min = values[0];
        int max = values[0];
        int sumOfValues = values[0];

        printf("\ncalculating min, max and sumOfValues...");
        for (int i = 1; i < valueCount; i++) {
            int currVal = values[i];

            if (currVal < min) {
                min = currVal;
            }
            else if (currVal > max) {
                max = currVal;
            }

            sumOfValues += currVal;
        }
        printf("\nfinished\n");

        printf("\nArithmetic average: %.3f", (float)sumOfValues / valueCount);
        printf("\nmin: %d", min);
        printf("\nmax: %d", max);
    }
    else {
        printf("\narray is empty!");
    }

    printf("\n************************************************************");

    return 0;
}
```

O(log(n))

Hier nimmt die Laufzeit des Algorithmus Logarithmisch zur Problemgröße zu.

Beispiele:

```c
int binarySearch(int values[],int valuesLen, int searchVal) {
    int first = 0;
    int last = valuesLen - 1;
    int mid = (first + last) / 2;

    while (first <= last) {
        if (values[mid] == searchVal) {
            printf("%d found at location %d.\n", searchVal, mid + 1);
            break;
        }
        else if (values[mid] < searchVal) {
            first = mid + 1;
        }
        else {
            last = mid - 1;
        }

        mid = (first + last) / 2;
    }

    if (first > last) {
        printf("Not found! %d isn't present in the list.\n", searchVal);
    }

    return searchVal;
}
```

```c
int quickPower(int x, int n) {
    int p = 1;

    while (n > 0) {
        if (n % 2 != 0) {
            p = p * x;
        }
        n = n / 2;
        x = x * x;
    }

    return p;
}
```

$O(n^2)$

Hier nimmt die Lauftzeut des Algorithmus zur Problemgröße Quadratisch zu.

```
void bubbleSort(int values[], int valuesLen) {
    for (int i = 1; i < valuesLen; i++) {
        for (int j = valuesLen - 1; j >= i; j--) {
            if (values[j - 1] > values[j]) {
                int x = values[j - 1];
                values[j - 1] = values[j];
                values[j] = x;
            }
        }
    }
}
```

```
void selectionSort(int values[], int valuesLen) {
    for (int i = 0; i < valuesLen - 1; i++) {
        int min = i;

        for (int j = i + 1; j < valuesLen; j++) {
            if (values[j] < values[min]) {
                min = j;
            }
        }

        int temp = values[min];
        values[min] = values[i];
        values[i] = temp;
    }
}
```