Ressourcen

Speicher Zeit

-59, 52, 46, 14, -50, 58, -37, -77, 34, 15

Gesucht: maximale Abschnittssumme

52 + 46 + 14 - 50 + 58 = 120

$t(n) = n \cdot (n/2 \cdot (n/4 + 1))$

```
int maxfolge1(int z[], int n) {
  int i, j, k, sum, max = -10000000;
  for (i = 0; i < n; i++)
    for (j = i; j < n; j++) {
      sum = 0;
      for (k = i; k <= j; k++)
        sum += z[k];
      if (sum > max) {
        max = sum;
      }
    }
  return max;
}
```

$t(n) = n(n^2/8 + n/2)$

$= n^3/8 + n^2/2$

$n = 10000$

$t(n) = 10^{4*3} = \underline{10^{12}}$

⇒ kubischer Algorithmus

Variante #2

```
int maxfolge1(int z[], int n) {
  int i, j,    , sum, max = -10000000;
  for(i = 0; i<n; i++)
    sum = 0;
    for(j = i; j<n; j++) {
      sum += z[j];
      if(sum > max) {
        max = sum;
      }
  return max;
}
```

$t(n) = n(n/2)$

$n = 10000 \quad t(n) = 10^{4^2} = 10^8$

$\Rightarrow \times 10000$ Schneller als Variante 1

Variante #3

```
int maxfolge1(int z[], int n) {
  int i, j,    s, gesamtmax = -10000000, endsumme = 0;
  for(i = 0; i<n; i++) {
    endsumme = ((s = endsumme + z[i]) > 0) ? s : 0;
    if(endsumme > gesamtmax)
      gesamtmax = endsumme;
    }
  return gesamtmax;
}
```

$t(n) = n$ ⇒ linear Algorithmen

$n = 10\,000$ $t(n) = 10^4$

→ $\times 10\,000$ schneller als V_2

→ $\times 100\,000\,000$ schneller als V_1

Analyse:

# V_1	# V_2	# V_3
$t(n) = n^3$	$t(n) = n^2$	$t(n) = n$
$t(n) = 10^{12}$	$t(n) = 10^8$	$t(n) = 10^4$

$\cdot 10k$ $\cdot 10k$ $\cdot 100\,000\,000$

Typische Komplexitätsfunktionen zu Algorithmen

1	konstant	Jede Anweisung eines Programms wird höchstens einmal ausgeführt. Dies ist der Idealzustand für einen Algorithmus.
log n	logarithmisch	Speicher- oder Zeitverbrauch wachsen nur mit der Problemgröße n. Die Basis des Logarithmus wird häufig 2 sein, d. h. vierfache Datenmenge verursacht doppelten Ressourcenverbrauch, 8-fache Datenmenge verursacht 3-fachen Verbrauch und 1024-fache Datenmenge 10-fachen Verbrauch.
n	linear	Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße n.
n log n	n log n	Der Ressourcenverbrauch liegt zwischen n (*linear*) und n^2 (*quadratisch*).
n^2	quadratisch	Speicher- oder Zeitverbrauch wachsen quadratisch mit der Problemgröße. Solche Algorithmen lassen sich praktisch nur für kleine Probleme anwenden.
n^3	kubisch	Speicher- oder Zeitverbrauch wachsen kubisch mit der Problemgröße. Solche Algorithmen lassen sich in der Praxis nur für sehr kleine Problemgrößen anwenden.
2^n	exponentiell	Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache. Solche Algorithmen sind praktisch kaum verwendbar.

O - Notation

$$t(n) = n\left(\frac{n^2}{8} + \frac{n}{2}\right)$$

$$t(n) = \frac{n^3}{8} + \frac{n^2}{2}$$

$\times$ $\Rightarrow$ Höchste Potenz finden

$\Rightarrow$ Streichen aller additiven u. multiplikativen Potenzen

$O(n^3)$

```
int seqsuche(int z[], int n, int zahl){
    int i;
    for(i = 0; i<n; i++){
        if(z[i] == zahl)
            return i;
    return -1;
}
```

Problemgröße: Länge des Arrays n

Kritischer Bereich

3 Fälle:

günstigster Fall: gesuchtes Element ganz vorne im Array

$t(n) = 1$

$O(1)$

ungünstigster Fall: gesuchtes Element kommt nicht vor

$t(n) = n$

$O(n)$

durchschnittlicher Fall: gesuchtes Element kommt irgendwo oder nicht vor

q... Wahrscheinlichkeit dass Element drinnen ist

$$t(n) = q \cdot \frac{n}{2} + (q+1)\,n$$

$O(n)$

Potenzen (Beispiel)

$$a^b = \underbrace{a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot \ldots}_{\times b}$$

```
int x = a;
int y = b;
int z = 1;
while (y > 0){
    if (y ungerade)
        z = z * x;
    y = y/2;
    x = x * x;
}
```

x	y	z	
4	7	1	$1 \cdot 4$
4	7	4	
16	3	4	$4 \cdot 4 = 4 \cdot (4^2)^3$ $4 \cdot 16^3$
16	3	64	
256	1	64	$64 \cdot 256^1$
256	1	16384	
65536	0	16384	

Problemgröße: Exponent y

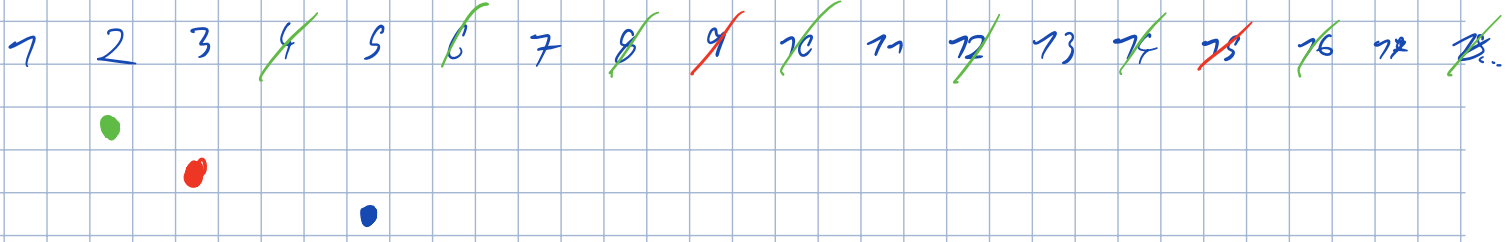
Krit. Bereich:

3 Fälle sind ident!

$$t(n) = \log_2(n)$$

$O(\log(n))$

Primzahlen (Beispiel)



Eingabe: Zahl(n)

```
for (i = 1; i <= n; i++)          // prim(0) wird zunächst nicht benutzt
    prim[i] = i;                  // zunächst alle Zahlen erstmal Primzahlen
for (i = 2; i <= n/2; i++)        <- n/2
    if (prim[i])                  // entspricht if(prim[i] != 0)
        for (j = 2*i; j <= n; j = j+i)   <- n/4 ... k.q.n
            prim[j] = 0;
```

Ausgabe: Die Primzahlen von 1 bis a sind

```
for (i = 2; i <= n; i++)
    if (prim[i])
        Ausgabe: i
```

3 Fälle: alle 3 Fälle sind gleich

Problemgröße: n

$$t(n) = T_1 + T_2 + T_3$$
$$= n + \frac{n}{2} \cdot \frac{n}{4} + n$$
$$= n + n^2 \frac{k \cdot q}{2} + n$$

$\rightarrow O(n^2)$

SORTIEREN

Im folgenden Arbeitsauftrag erarbeiten Sie sich folgende drei Sortieralgorithmen: ***Bubble-Sort****,* ***Insert-Sort*** *und* ***Select-Sort****. Zu jedem Algorithmus ist die* ***Funktionsweise*** *zu recherchieren, eine* ***Implementierung*** *zu erstellen und dessen* ***Laufzeit zu analysieren****.*
Die Laufzeitanalyse beinhaltet dabei funktionierende Implementierung/Musterlösung. Gehen Sie daher schrittweise vor und versuchen Sie die Implementierung tatsächlich selbst zu erstellen!

Vergleichen Sie Ihre Lösungen im Anschluss im Team!

BUBBLE-SORT

FUNKTIONSWEISE

Recherchieren Sie die Funktionsweise des Bubble-Sort Algorithmus im Internet und beschreiben Sie seine Funktionsweise mit eigenen Worten.

Beim Bubblesort - Algorithmus werden immer 2 Werte (der vorherige & aktuelle) verglichen, wobei der höhere Wert immer bis zum Ende mitgenommen wird.

Max 300 Zeichen

IMPLEMENTIERUNG

Implementieren Sie den Bubble-Sort Algorithmus in Java und testen Sie ihn mit folgendem Code-Stück:

```
int[] a = {19,96,30,11,73,2,99,72,69,73};
bubble_sort(a);
```

Die Implementierung soll folgenden Output erzeugen:

```
------ vor bubble_sort------------------
 19 96 30 11 73  2 99 72 69 73
 1. Durchlauf:   2 96 30 19 73 11 99 72 69 73
 2. Durchlauf:   2 11 96 30 73 19 99 72 69 73
 3. Durchlauf:   2 11 19 96 73 30 99 72 69 73
 4. Durchlauf:   2 11 19 30 96 73 99 72 69 73
 5. Durchlauf:   2 11 19 30 69 96 99 73 72 73
 6. Durchlauf:   2 11 19 30 69 72 99 96 73 73
 7. Durchlauf:   2 11 19 30 69 72 73 99 96 73
 8. Durchlauf:   2 11 19 30 69 72 73 73 99 96
 9. Durchlauf:   2 11 19 30 69 72 73 73 96 99
------ nach bubble_sort-----------------
  2 11 19 30 69 72 73 73 96 99
```

LAUFZEITKOMPLEXITÄT

Überlegen Sie sich die Laufzeitkomplexität des Bubble-Sort Algorithmus!
Geben Sie die Laufzeitkomplexität in der O-Notation für den günstigsten, durchschnittlichen, ungünstigsten Fall an! Falls alle drei Fälle ident sind/nicht unterschieden werden können, begründen Sie!
Begründen Sie außerdem Ihr Ergebnis! **Eine Antwort (z.B. O(n³)) alleine ist nicht ausreichend!**

***Hinweis**: Begründungen können Herleitungen/Erklärungen/Berechnungen der Anzahl der Operationen abhängig von der Problemgröße sein. Beispiele unterstützen Erklärungen.*

```
public static void bubble_sort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] > a[j]) {
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

n, n/2

$t(n) = n \frac{n}{2} = \frac{n^2}{2}$

$O(n^2)$

Problemgröße a.length = n

günstig = ungünstig = durchschn. Fall
weil wir nie vorzeitig die Schleife beenden.

Laufzeitanalyse

INSERT-SORT

FUNKTIONSWEISE

Recherchieren Sie die Funktionsweise des Insert-Sort Algorithmus im Internet und beschreiben Sie seine Funktionsweise mit eigenen Worten.

Ersten Wert wird als erster Wert des neuen Stapels genommen. Jeder nächste Wert wird in den neuen Stapel sortiert eingefügt.

Max 300 Zeichen

IMPLEMENTIERUNG

Implementieren Sie den Insert-Sort Algorithmus in Java und testen Sie ihn mit folgendem Code-Stück:

```
int[] a = {61, 96, 24, 45, 87, 29, 4, 14, 31, 31};
insert_sort(a);
```

Die Implementierung soll folgenden Output erzeugen:

```
——— vor insert_sort———————————————
 61 96 24 45 87 29  4 14 31 31

 1. Durchlauf:  61 96 24 45 87 29  4 14 31 31
 2. Durchlauf:  24 61 96 45 87 29  4 14 31 31
 3. Durchlauf:  24 45 61 96 87 29  4 14 31 31
 4. Durchlauf:  24 45 61 87 96 29  4 14 31 31
 5. Durchlauf:  24 29 45 61 87 96  4 14 31 31
 6. Durchlauf:   4 24 29 45 61 87 96 14 31 31
 7. Durchlauf:   4 14 24 29 45 61 87 96 31 31
 8. Durchlauf:   4 14 24 29 31 45 61 87 96 31
 9. Durchlauf:   4 14 24 29 31 31 45 61 87 96
——— nach insert_sort———————————————
  4 14 24 29 31 31 45 61 87 96
```

LAUFZEITKOMPLEXITÄT

Überlegen Sie sich die Laufzeitkomplexität des Insert-Sort Algorithmus!
Geben Sie die Laufzeitkomplexität in der O-Notation für den günstigsten, durchschnittlichen, ungünstigsten Fall an! Falls alle drei Fälle ident sind/nicht unterschieden werden können, begründen Sie!
Begründen Sie außerdem Ihr Ergebnis! **Eine Antwort (z.B. $O(n^3)$) alleine ist nicht ausreichend!**

***Hinweis**: Begründungen können Herleitungen/Erklärungen/Berechnungen der Anzahl der Operationen abhängig von der Problemgröße sein. Beispiele unterstützen Erklärungen.*

```java
public static void insert_sort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0 && a[j] < a[j-1]; j--) {
            int t = a[j];
            a[j] = a[j - 1];
            a[j - 1] = t;
        }
    }
}
```

Problemgröße: a.length = n

Günstigster Fall:

sortierte Liste

① n Iterationen

② 1 Iteration

$t(n) = n \cdot 1 = n$

$O(n)$

Ungünstigster Fall: ⇒ Werte genau verdreht

① n Iterationen

② n/2 Iterationen

$t(n) = n \cdot \frac{n}{2} = \frac{n^2}{2}$

$O(n^2)$

Durchschnittlicher Fall:

gerichtet

① n Iterationen

② $\frac{n}{4}$ Iterationen

$t(n) = n \frac{n}{4} = \frac{n^2}{4}$

$O(n^2)$

Laufzeitanalyse

SELECT-SORT

FUNKTIONSWEISE

Recherchieren Sie die Funktionsweise des Selection-Sort Algorithmus im Internet und beschreiben Sie seine Funktionsweise mit eigenen Worten.

Wählt am Anfang den ersten Wert des Arrays aus und vergleicht ihn mit allen anderen Werten des Arrays. Den kleinsten Wert tauscht er nun mit dem ausgewählten Wert aus.
Danach wählt er den nächsten Wert aus und das ganze beginnt von vorne.

Max 300 Zeichen

IMPLEMENTIERUNG

Implementieren Sie den Select-Sort Algorithmus in Java und testen Sie ihn mit folgendem Code-Stück:

```
int[] a = {24,74,24,86,59,96,13,76,7,39};
Sorting.select_sort(a);
```

Die Implementierung soll folgenden Output erzeugen:

```
——— vor select_sort———————————
 24 74 24 86 59 96 13 76  7 39
 1. Durchlauf:   7 74 24 86 59 96 13 76 24 39
 2. Durchlauf:   7 13 24 86 59 96 74 76 24 39
 3. Durchlauf:   7 13 24 86 59 96 74 76 24 39
 4. Durchlauf:   7 13 24 24 59 96 74 76 86 39
 5. Durchlauf:   7 13 24 24 39 96 74 76 86 59
 6. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 7. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 8. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 9. Durchlauf:   7 13 24 24 39 59 74 76 86 96
——— nach select_sort———————————
  7 13 24 24 39 59 74 76 86 96
```

LAUFZEITKOMPLEXITÄT

Überlegen Sie sich die Laufzeitkomplexität des Insert-Sort Algorithmus!
Geben Sie die Laufzeitkomplexität in der O-Notation für den günstigsten, durchschnittlichen, ungünstigsten Fall an! Falls alle drei Fälle ident sind/nicht unterschieden werden können, begründen Sie!
Begründen Sie außerdem Ihr Ergebnis! **Eine Antwort (z.B. $O(n^3)$) alleine ist nicht ausreichend!**

***Hinweis**: Begründungen können Herleitungen/Erklärungen/Berechnungen der Anzahl der Operationen abhängig von der Problemgröße sein. Beispiele unterstützen Erklärungen.*

```java
public static void select_sort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int h = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[h] > a[j]) {
                h = j;
            }
        }
        if (h != i) {
            int t = a[h];
            a[h] = a[i];
            a[i] = t;
        }
        System.out.println(Arrays.toString(a));
    }
}
```

n

n/2

1

Problemgröße: a.length = n

$$t(n) = n\left(\frac{n}{2} + 1\right)$$

$$= \frac{n^2}{2} + n$$

$$O(n^2)$$

Laufzeitanalyse