Umgang mit Speicherlimitationen

Bislang haben wir uns bei der Effizienzanalyse ausschliesslich auf die Laufzeitkosten fokussiert.

Ein anderes Effizienzmass ist ebenfalls sehr wichtig: der Speicherbedarf!

Wann wichtig?

- wenn Speicher knapp ist (Geräte), oder
- die Datenmengen sehr gross sind (Big Data)…

Algorithmen sollten möglichst schnell UND speichereffizient sein – aber:

Wir müssen darüber nachdenken, ob wir Geschwindigkeit über Speicherverbrauch setzen oder umgekehrt…

Big-O-Notation für Speicherbedarf

Schlüsselfrage für Zeitkomplexität war: „Wie viele Schritte braucht der Algorithmus bei N Datenelementen?“

Schlüsselfrage für Speicherbedarf: „Wie viele Speichereinheiten braucht der Algorithmus bei N Datenelementen zusätzlich?“

Einfaches Beispiel:

Funktion *MakeUppercase()* bekommt ein Array von Strings und gibt dieses Array zurück - alle Strings konvertiert in CAPS. Das Array ["fred", "anna", "oskar", "gerda"] würde konvertiert werden in ["FRED", "ANNA", "OSKAR", "GERDA"].

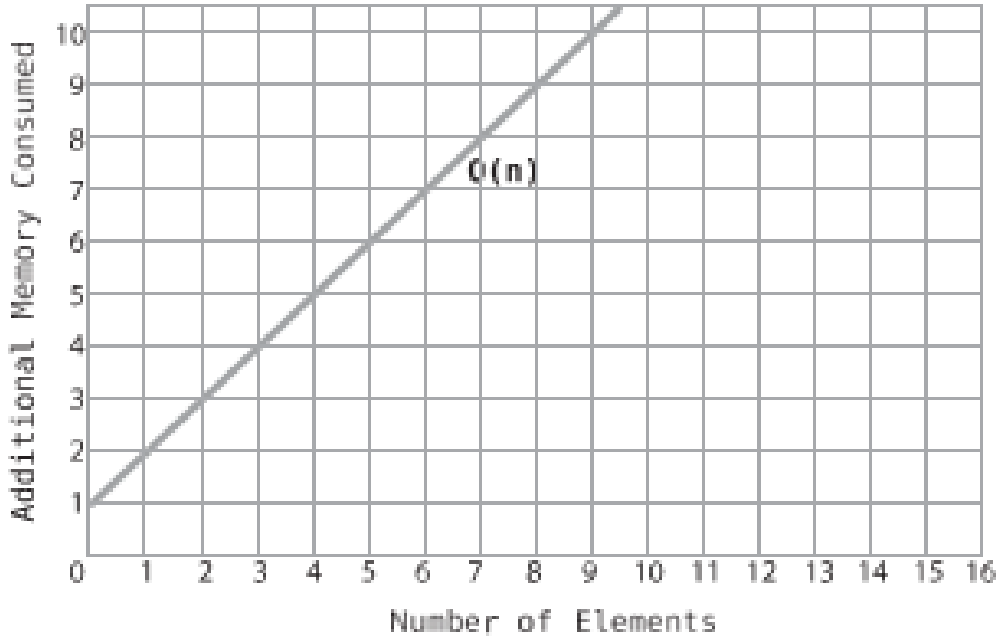
```
char * MakeUppercase (char *pArray[], int size)
{
    char *pNew=NULL;
    int shift=0;
    for (int i = 0; i < size; i++)
    {
        if (pNew) pNew=realloc(pNew, strlen(pArray[i])+1);  // existierendes Array um nächsten String verlängern
        else pNew=malloc(strlen(pArray[0])+1);              // Array mit erstem String beginnen
        for (int j=0; j<strlen(pArray[i]);j++)
        {
            *(pNew+shift) = toupper(pArray[i][j]);          // String konvertieren
            shift++;
        }
        *(pNew+shift)=0; shift++;
    }
    return pNew;
}
```

Big-O-Notation für Speicherbedarf

Wir bekommen einen Pointer pArray übergeben, und erzeugen eine komplette Kopie dieses Arrays, dessen Adresse wir zurückgeben.

-> 2 Arrays belasten den Hauptspeicher: das originale Array und die neu erzeugte Kopie.

Da der Gesamtspeicherbedarf proportional zur Menge der Eingangsdaten ist, ist er O (N).



Big-O-Notation für Speicherbedarf

Speichereffizientere Version (in place):

```
void MakeUppercase1 (char pArray[][6], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j=0; j<strlen(pArray[i]);j++)
        {
            pArray[i][j] = toupper(pArray[i][j]);
        }
    }
}
```

- Erzeugung keines neuen Arrays, keine sonstigen weiteren Datenelemente:
  Modifizierung des übergebenen Arrays *in place*.

O(1)-Algorithmus bzgl. Speicherplatz: Zusätzlich benötigter Platz konstant (=0), unabhängig von der Größe des Eingangsarrays!

Speichereffizienz ist O (1)!

Big-O-Notation für Speicherbedarf

Die Berechnung der Speichereffizienz betrachtet nur den Bedarf an zusätzlichem Speicher durch den Algorithmus!
Vergleich der beiden Implementierungen:

Version	**Zeitkomplexität**	**Speicherkomplexität**
Version #1	O(N)	O(N)
Version #2	O(N)	O(1)

Tradeoffs zwischen Zeit und Platz

Diese Funktion (in 2 Implementierungen) bekommt ein Array und bestimmt, ob es Duplikate enthält.

```
bool HasDuplicateValue (unsigned int array[], int size)
{
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < size; j++)
        {
            if(i != j && array[i] == array[j])
            {
                return true;
            }
        }
    }
    return false;
}
```

```
bool HasDuplicateValueA (unsigned int array[], int size)
{
    int existingNumbers [MAXNUM]={0};
    for(int i = 0; i < size; i++)
    {
        if (0 == existingNumbers [array[i]] )
        {
            existingNumbers [array[i]] = 1;
        }
        else    return true;
    }
    return false;
}
```

Zeit:
Version #1: Benutzt verschachtelte Schleifen: O (N^2)

Zeit:
Version #2: Benutzt ein Array und nur eine Schleife: O (N)

Trade-Offs zwischen Zeit und Platz

```
bool HasDuplicateValue (unsigned int array[], int size)
{
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < size; j++)
        {
            if(i != j && array[i] == array[j])
            {
                return true;
            }
        }
    }
    return false;
}
```

Speicherplatz:
Version #1: Kein zusätzlicher Speicher: O (1)

```
bool HasDuplicateValueA (unsigned int array[], int size)
{
    int existingNumbers [MAXNUM]={0};
    for(int i = 0; i < size; i++)
    {
        if (0 == existingNumbers [array[i]] )
        {
            existingNumbers [array[i]] = 1;
        }
        else    return true;
    }
    return false;
}
```

Speicherplatz:
Version #2: erzeugt ein Array mit max. MAXNUM Einträgen: O (M),
M = Wertebereich des Arrays

Trade-Offs zwischen Zeit und Platz

Version	Time complexity	Space complexity
Version #1	$O(N^2)$	O(1)
Version #2	O(N)	O(M)

Version #2: Optimal, wenn Geschwindigkeit nötig ist, und der maximale Wertebereich bekannt und handlebar.

Version #1: Optimal, wenn der lokale Speicher knapp oder der Wertebereich sehr gross oder unbekannt ist.

Trade-Offs zwischen Zeit und Platz

Version #3:

```
bool HasDuplicateValueS (unsigned int array[], int size)
{
    MySort(array, size);
    for(int i = 0; i < size; i++)
    {
        if (array[i] == array[i + 1])
        {
            return true;
        }
    }
    return false;
}
```

Version	Time complexity	Space complexity
Version #1	$O(N^2)$	O(1)
Version #2	O(N)	O(N)
Version #3	O(N logN + N) (schnellste Sortierung!)	O(log N) (viele Quicksort-Impl.)

-> sehr guter TradeOff zwischen Zeit und Platz!

Sortierung grosser, evtl. externer Datenmengen

Bei Sortierung externer Daten kann nicht prinzipiell die gesamte Datenmenge für direkten Zugriff und Vertauschen gleichzeitig in den Speicher geholt werden (Größe nicht limitiert), sondern es müssen Teilmengen als Zwischenschritte sortiert werden können.

Divide and Conquer (Teile, Sortiere und Erobere zurück)

Erlaubt zusätzlich Parallelisierung auf mehrere Rechenkerne:

1. Die zu sortierende Datei wird in Teile zerlegt, die jeweils intern sortiert werden können.

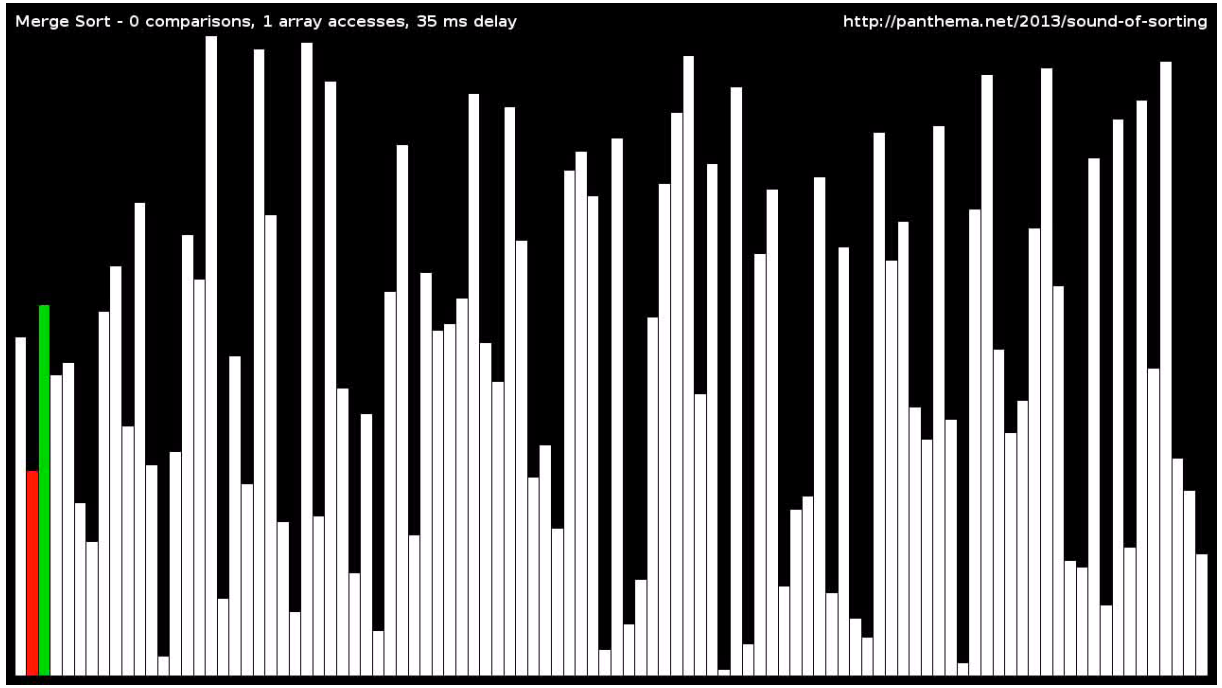
2. Die Teile werden parallel sortiert.

3. Das Ergebnis wird aus den sortierten Teilen zusammengemischt:
   Dazu ist ein Datenelement pro vorsortierter Teilmenge im Speicher ausreichend, das Kleinste wird zum Ergebnis hinzugefügt und durch seinen Nachfolger aus der Ursprungsdatei ersetzt.

MergeSort: Sortieren durch Verschmelzen

Idee: Teile den Eingangsstapel in zwei möglichst gleich große Teile. Gib jeden Teil einem Helfer mit der Bitte, auch nach dem hier beschriebenen Verfahren vorzugehen.

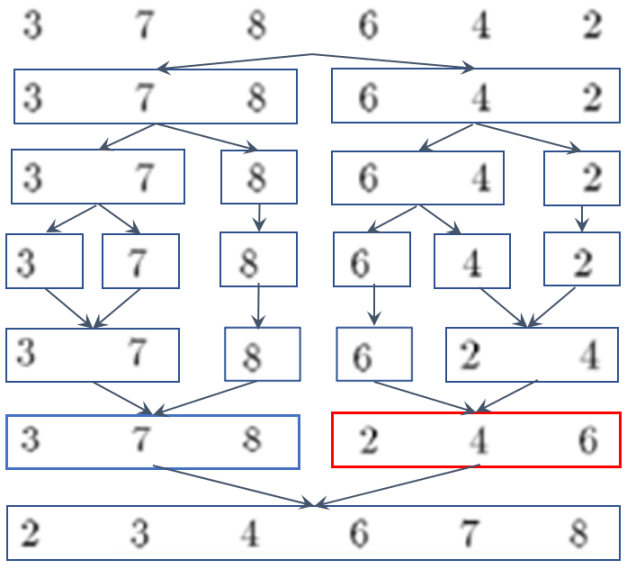
Warte, bis Dir beide sortierte Teile zurückgegeben wurden. Dann durchlaufe beide Stapel gleichzeitig von oben nach unten, und mische die Karten im Reißverschlussprinzip zu einem sortierten Gesamtstapel zusammen. Gib diesen an deinen Auftraggeber zurück.



http://panthema.net/2013/sound-of-sorting

MergeSort: Sortieren durch Verschmelzen

- Divide & Conquer, vergleichsbasiert
- Schnellstes stabiles externes Verfahren, braucht intern nur den Platz zum Mischen



Zeitkomplexität (Worst case) = O(n log(n))

Speicherbedarf: O(n) zusätzlich – nicht in place

Parallelisierbar!

DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

MergeSort Implementierung

```
void MergeSort(int array[], int size){

    if(size > 1) // Abbruchkriterium
    {
        int aLeft[size/2];   int aRight[(size + 1)/2]; // temp arrays init.
        int i;
        for(i = 0; i < size/2; ++i)     aLeft[i] = array[i];
        for(i = size/2; i < size; ++i)  aRight[i - size/2] = array[i];

        MergeSort(aLeft,size/2);          // linker Teil rekursiv bis 1 Elem.
        MergeSort(aRight,(size + 1)/2);   // rechter Teil rekursiv bis 1 Elem.

        int *pos1 = &aLeft[0];  // beide Teile mergen
        int *pos2 = &aRight[0];
        for(i = 0; i < size; ++i) // zuerst: size =2
        {
            if(*pos1 <= *pos2) // linkes Element < rechtes?
            {
                array[i] = *pos1; //  kleiner/ gleiches zuerst ins Array
                if (pos1 != &aRight[(size+1)/2 - 1])
                {// pos1 nicht verändern, wenn der größte Wert mehrmals vorkommt
                    if (pos1 == &aLeft[size/2-1]) pos1= &aRight[(size+1)/2 - 1];
                    else ++pos1;
                }
            }
            else
            {
                array[i] = *pos2; // anderes zuerst ins Array
                if (pos2 == &aRight[(size+1)/2 -1]) pos2= &aLeft[size/2 - 1];
                else  ++pos2;
             }
         }
     }
}
```

External Memory MergeSort

- Einzelne Blöcke werden im Hauptspeicher sortiert und nach dem Sortieren zurückgeschrieben (z.B. in Files)
- Die k sortierten Blöcke/ Files werden durch K-Way-Merges gemerged und die Datenelemente sofort zurückgeschrieben

Versteckte Rekursionskosten bzgl. Speicher

Einfaches Beispiel:

```
void Recurse (int n)
{
    if (n < 0)    return;
    printf ("n: %d\n",n);
    Recurse(n - 1);
}
```

Die Funktion gibt den Countdown einer Zahl auf die Konsole aus.

Geschwindigkeit: O (N)
Speicherplatz: erzeugt keine neuen Daten.

Oder doch?

Für den Wert 100 legt die Funktion die 100… 1 auf den Stack, plus jeweils die Rücksprungadresse: O (N).

Für alle Rekursionen gilt: Eine Funktion benötigt eine Platzeinheit für jeden rekursiven Aufruf.

Die Stacklimits sind sehr eng! Mein Notebook bricht den Countdown der Zahlen ab 50.000 bei 1.733 ab, und terminiert mit “RangeError: Maximum call stack size exceeded”!

Versteckte Rekursionskosten

Ansatz mit einfacher Schleife:

```
void  Loop (int n)
{
    while (n >= 0)
    {
        printf ("n: %d\n",n);
        n--;
    }
}
```

Die Funktion gibt den Countdown einer Zahl auf die Konsole aus.

Geschwindigkeit: O (N)
Platz: ~~O (N)~~ O(1)

Quicksort:

Platzbedarf: O (log N), weil es log N rekursive Aufrufe macht, und die Stacktiefe von log N benötigt!

Speicherbedarf von Sortieralgorithmen

INTERN vs. EXTERN

IN PLACE/ INTERN	EXTERN
Alle Datenelemente sind im Zugriff/ bekannt.	Nur ein Teil der Daten ist zu einem Zeitpunkt bekannt.
• Alle Daten passen in den Speicher	• Datensätze zu groß für Hauptspeicher
• Algorithmus findet auf dem Hauptspeicher statt	• Aufteilung der Daten auf externe Speichermedien

Sortierverfahren ohne Vergleiche

CountingSort: Sortieren durch Auszählen

Nichtvergleichendes Sortieren in Linearzeit (O(N))

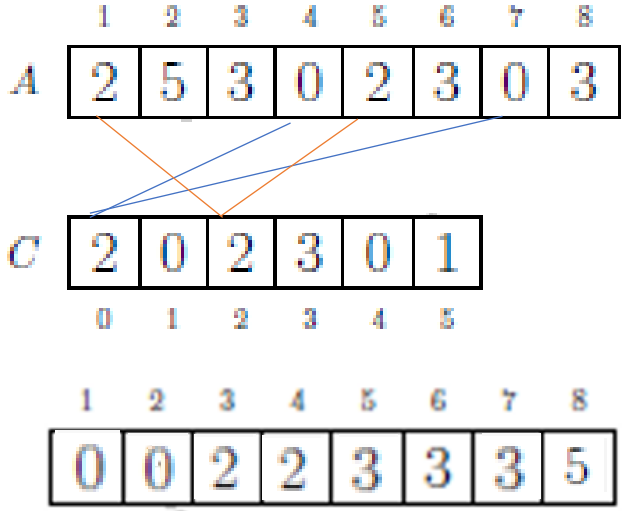
Wertebereich: natürliche Zahlen, begrenztes Intervall (Alter, Distanzen, …)

Idee: Bestimme in einem Arbeitsarray (Hashtabelle) C[0..k] für jeden Index die Anzahl der Schlüssel A, die diesem Index entsprechen. Dann plaziere direkt jedes Element, das dem nächsten Index entspricht, (in einer Schleife) in die richtige Stelle in B.

Zielgruppe: kleiner Wertebereich –> kleiner Platzbedarf (Platzbedarf nur abhängig vom Intervall), geringe Laufzeit
Nachteil: Sortierung kompositer Daten möglich, aber sehr viel langsamer durch zusätzliche Suche des n. Elements

Sortierverfahren ohne Vergleiche

CountingSort: Sortieren durch Auszählen




```
void CountingSort (int a[], int limit, int size)
{
    // initialisiertes Array anlegen
    int *arrayC=calloc(limit, sizeof(a[0]));
    for (int i = 0; i < size; i++)
    {
       arrayC [a[i]]+=1;  //C aus A
    }
    int indexA=0;
    for (int j = 0; j < limit; j++) //A aus C
    {
        while (0 != arrayC [j])
        {
            arrayC [j]--;
            a[indexA] = j;
            indexA++;
        }
    }
}
```

Zeitkomplexität: O(n + k)

Speicherbedarf: O(k)

Zusammenfassung

Wir haben gelernt, die Effizienz eines Algorithmus bzgl Zeit und Platzbedarf zu bestimmen.

Dadurch können wir unterschliedliche Verfahren und unterschiedliche Implementierungen des gleichen Algorithmus miteinander vergleichen.

Komplexität von Sortierverfahren

Verfahren	Zeitbedarf		Platzbedarf	Stabilität	Eher geeignet für Arrays	Eher geeignet für Listen	Geeignet für externe Daten
	Worst case	Average case					
BubbleSort	$O(n^2)$	$O(n^2)$	$O(1)$	ja	x		nein
SelectionSort	$O(n^2)$	$O(n^2)$	$O(1)$	(nein)	x		nein
InsertionSort	$O(n^2)$	$O(n^2)$	$O(1)$	ja		x	nein
QuickSort	$O(n^2)$	$O(n \log n)$	$O(\log n)$	(nein)	x		ja
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$	ja		x	ja
CountingSort	$O(n + k)$	$O(n + k)$	$O(n + k)$	ja	x		(ja)

Übungen in BigBlueButton

1. Beschreiben Sie die Speicherplatz-Komplexität des “Word Builder”-Algorithmus mit Hilfe von Big O.

Der Algorithmus erzeugt jede Kombination einzelner Zeichen zu Strings aus 2 Zeichen.
Bekommen wir z.B. das Array ["a", "b", "c", "d"], würden wir ein neues Array zurückgeben, das folgende Stringkombinationen enthält: [*'ab', 'ac', 'ad', 'ba', 'bc', 'bd', 'ca', 'cb', 'cd', 'da', 'db', 'dc'*]

```
char * WordBuilder (char array[], int size)
{
    char *pStrings = malloc(0);
    int index=0;
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < size; j++)
        {
            if (i != j)
            {
                pStrings=realloc(pStrings, 3);
                pStrings[index]=array[i];
                pStrings[index+1]=array[j];
                pStrings[index+2]=0;
                index+=3;
            }
        }
    }
    return pStrings;
}
```

Antworten:

A O (1)
B O (log N)
C O (N)
D O (N*logN)
E O (N^2)

Übung

2. Diese Funktion kehrt einen String um. Beschreiben Sie ihre Speicherplatz-Komplexität mit Hilfe von Big O:

```
char * Reverse (char array[], int size)
{
    char *pString = malloc(size+1);
    for (int i = size-1; i >= 0; i--)
    {
        pString[size-1-i]=array[i];
    }
    pString[size]=0;
    return pString;
}
```

Antworten:

A O (1)
B O (log N)
C O (N)
D O (N*logN)
E O (N^2)

Übung

3. Gegeben sind 3 verschiedene Implementierungen einer Funktion, die ein Array von Zahlen bekommt und ein Array der verdoppelten Zahlen zurückgibt. Für den Input [5, 4, 3, 2, 1] wäre der Ouput: [10, 8, 6, 4, 2].

```
int * doubleArray1 (int array[], int size)
{
    int * newArray;
    for (int i = 0; i < size; i++)
    {
       if (0==i) newArray = malloc(sizeof(array[0]));
       else newArray=realloc(newArray, sizeof(array[0]));
       newArray[i]=array[i]*2;
    }
    return newArray;
}
```

```
int * doubleArray2 (int array[], int size)
{
    for (int i = 0; i < size; i++)  array[i] *= 2;
    return array;
}
```

```
void doubleArray3 (int array[], int size, int index)
{
    if (index >= size)  return;
    array[index] *= 2;
    doubleArray3 (array, size, index + 1);
}
```

Die Speicherplatzkomplexität von Version #1 ist:

A O (1)
B O (log N)
C O (N)
D O (N*logN)
E O (N^2)

Übungen

3. Gegeben sind 3 verschiedene Implementierungen einer Funktion, die ein Array von Zahlen bekommt und ein Array der verdoppelten Zahlen zurückgibt. Für den Input [5, 4, 3, 2, 1] wäre der Ouput: [10, 8, 6, 4, 2].

```
int * doubleArray1 (int array[], int size)
{
    int * newArray;
    for (int i = 0; i < size; i++)
    {
       if (0==i) newArray = malloc(sizeof(array[0]));
       else newArray=realloc(newArray, sizeof(array[0]));
       newArray[i]=array[i]*2;
    }
    return newArray;
}
```

```
int * doubleArray2 (int array[], int size)
{
    for (int i = 0; i < size; i++)  array[i] *= 2;
    return array;
}
```

```
void doubleArray3 (int array[], int size, int index)
{
    if (index >= size)  return;
    array[index] *= 2;
    doubleArray3 (array, size, index + 1);
}
```

Die Speicherplatzkomplexität von Version #2 ist:

A O (1)
B O (log N)
C O (N)
D O (N*logN)
E O (N^2)

Übungen

3. Gegeben sind 3 verschiedene Implementierungen einer Funktion, die ein Array von Zahlen bekommt und ein Array der verdoppelten Zahlen zurückgibt. Für den Input [5, 4, 3, 2, 1] wäre der Ouput: [10, 8, 6, 4, 2].

```
int * doubleArray1 (int array[], int size)
{
    int * newArray;
    for (int i = 0; i < size; i++)
    {
       if (0==i) newArray = malloc(sizeof(array[0]));
       else newArray=realloc(newArray, sizeof(array[0]));
       newArray[i]=array[i]*2;
    }
    return newArray;
}
```

```
int * doubleArray2 (int array[], int size)
{
    for (int i = 0; i < size; i++)  array[i] *= 2;
    return array;
}
```

```
void doubleArray3 (int array[], int size, int index)
{
    if (index >= size)  return;
    array[index] *= 2;
    doubleArray3 (array, size, index + 1);
}
```

Die Speicherplatzkomplexität von Version #3 ist:

A O (1)
B O (log N)
C O (N)
D O (N*logN)
E O (N^2)