The `for` loop

- Mathematical symbols $\sum_{j=1}^{n}$ and $\prod_{j=1}^{n}$
- Count-controlled loops
- `for`

Loops

- Loops allow for repeating the execution of one command (or more)
- You might need a loop, while dealing with
  - Vectors & Matrices
  - Counters $j = 1, \ldots, n$
  - Sums $\sum_{j=1}^{n} a_j := a_1 + a_2 + \cdots + a_n$
  - Products $\prod_{j=1}^{n} a_j := a_1 \cdot a_2 \cdots a_n$
  - Expressions like, e.g., *as long as* or *until*
- Classification
  - Count-controlled loops (`for`): Repeat an action for a specific number of times
  - Condition-controlled loops: Repeat an action until some condition is satisfied

The `for` loop

- `for (init. ; cond. ; step-expr.) statement`
- Working principles of a `for` loop
  - (1) Execution of `init.` (initialization)
  - (2) Break, if the condition `cond.` is not satisfied
  - (3) Execution of `statement`
  - (4) Execution of `step-expr.`
  - (5) Go back to (2)
- `statement` is
  - a single line of code
  - more lines of code in curly brackets `{...}`, i.e., a block

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5
 6   for (j=5; j>0 ; j=j-1)
 7     printf("%d ",j);
 8
 9   printf("\n");
10 }
```

- `j=j-1` in line 6 is an assignment, no equality!
- Output:
  `5 4 3 2 1`

Read and print a vector

```
#include <stdio.h>

void scanVector(double input[], int dim) {
  int j = 0;
  for (j=0; j<dim; j=j+1) {
    input[j] = 0;
    printf("%d: ",j);
    scanf("%lf",&input[j]);
  }
}

void printVector(double output[], int dim) {
  int j = 0;
  for (j=0; j<dim; j=j+1) {
    printf("%f ",output[j]);
  }
  printf("\n");
}

main() {
  double x[5];
  scanVector(x,5);
  printVector(x,5);
}
```

- Functions must know array length
  - Passed as input parameter
- Arrays are passed via call by reference

Conventions (recall)

- Local variables are `lowercase_with_underscores`
- Global variables are `auch_underscore_hinten_`
- Constants are `UPPERCASE_WITH_UNDERSCORES`
- Functions are `firstWordLowercaseNoUnderscores`

Minimum of a vector

```
#include <stdio.h>
#define DIM 5

void scanVector(double input[], int dim) {
  int j = 0;
  for (j=0; j<dim; j=j+1) {
    input[j] = 0;
    printf("%d: ",j);
    scanf("%lf",&input[j]);
  }
}

double min(double input[], int dim) {
  int j = 0;
  double minval = input[0];
  for (j=1; j<dim; j=j+1) {
    if (input[j]<minval) {
      minval = input[j];
    }
  }
  return minval;
}

main() {
  double x[DIM];
  scanVector(x,DIM);
  printf("Minimum of the vector: %f\n", min(x,DIM));
}
```

- Note the structure (follow it for your exercises)
  - The length of the vector is a constant in `main`
    - i.e., the length cannot be changed
  - It is input parameter in `scanVector`
    - i.e., the function works for any length

Example: Sum symbol $\sum$

- Computation of the sum $S = \sum_{j=1}^{N} a_j$
  - Shorthand notation $\sum_{j=1}^{N} a_j := a_1 + a_2 + \cdots + a_N$
- Auxiliary partial sum $S_k = \sum_{j=1}^{k} a_k$
- Then, it holds that
  - $S_1 = a_1$
  - $S_2 = S_1 + a_2$
  - $S_3 = S_2 + a_3$ etc.
- Can be realized with N sums
  - Be careful: Assignment, no equality
    - $\mathtt{S} = \mathtt{a}_1$
    - $\mathtt{S} = \mathtt{S} + \mathtt{a}_2$
    - $\mathtt{S} = \mathtt{S} + \mathtt{a}_3$
    - etc.

Example: Sum symbol $\sum$

```
1 #include <stdio.h>
2
3 main() {
4   int j = 0;
5   int n = 100;
6
7   int sum = 0;
8
9   for (j=1; j<=n; j=j+1) {
10    sum = sum+j;
11  }
12
13  printf("sum_{j=1}^{%d} j = %d\n",n,sum);
14 }
```

- The program computes the sum $\sum_{j=1}^{n} j$ for $n = 100$
- Output:

```
sum_{j=1}^{100} j = 5050
```

- Be careful: Do not forget to inizialize (with zero) the variable for the result; see line 7
  - Otherwise: Wrong/random result!
- `sum += j;` is a shorthand notation for `sum = sum + j;`

Example: Product symbol $\prod$

```
1 #include <stdio.h>
2
3 main() {
4   int j = 0;
5   int n = 5;
6
7   int factorial = 1;
8
9   for (j=1; j<=n; j=j+1) {
10    factorial = factorial*j;
11  }
12
13  printf("%d! = %d\n",n,factorial);
14 }
```

- The program computes the factorial $n! = \prod_{j=1}^{n} j$ for $n = 5$
- Output:

```
5!  = 120
```

- Be careful: Do not forget to inizialize (with one) the variable for the result; see line 7
  - Otherwise: Wrong/random result!
- `factorial *= j;` is a shorthand notation for `factorial = factorial*j;`

Matrix-vector multiplication

- for loops can also be nested
  - e.g., matrix-vector multiplication
- Let $A \in \mathbb{R}^{M\times N}$ matrix, $x \in \mathbb{R}^N$ vector
- Define $b := Ax \in \mathbb{R}^M$ as $b_j = \sum_{k=0}^{N-1} A_{jk}x_k$
  - Recall: Indexing in C starts with 0
- $Ax = b$ is associated with the linear system

$$
\begin{array}{ccccccc}
A_{00}x_0 & + & A_{01}x_1 & + \dots + & A_{0,N-1}x_{N-1} & = & b_0 \\
A_{10}x_0 & + & A_{11}x_1 & + \dots + & A_{1,N-1}x_{N-1} & = & b_1 \\
A_{20}x_0 & + & A_{21}x_1 & + \dots + & A_{2,N-1}x_{N-1} & = & b_2 \\
\vdots & & \vdots & & \vdots & & \vdots \\
A_{M-1,0}x_0 & + & A_{M-1,1}x_1 & + \dots + & A_{M-1,N-1}x_{N-1} & = & b_{M-1}
\end{array}
$$

- Implementation
  - external loop runs over j
  - internal loop to compute the sum

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Be careful: Initialization `b[j] = 0`!

Matrices in column-major order

- Many math. libraries store matrices columnwise as vectors (column-major order)
  - $A \in \mathbb{R}^{M\times N}$ is stored as $a \in \mathbb{R}^{MN}$
  - $a = (A_{00}, A_{10}, \dots, A_{M-1,0},\ A_{01}, A_{11}, \dots,\ A_{M-1,N-1})$
  - A_{jk} corresponds to a_ℓ with $\ell = j + k \cdot M$
- Column-major order must be used if one wants to use those libraries
  - Usually the case with libraries written in Fortran
- Matrix-vector product
  - $b := Ax \in \mathbb{R}^M$, $b_j = \sum_{k=0}^{N-1} A_{jk}x_k$
  - with `double A[M][N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Matrix-vector product (column-major order)
  - with `double A[M*N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j+k*M]*x[k];
  }
}
```


Selection sort

- Input: Vector $x \in \mathbb{R}^n$
- Aim: Sorting x so that $x_1 \le x_2 \le \dots \le x_n$
- Algorithm (Step 1)
  - Seek x_k of $x_1, \dots, x_n$
  - Swap x_1 and x_k, i.e., x_1 is smallest entry
- Algorithm (Step 2)
  - Seek x_k of $x_2, \dots, x_n$
  - Swap x_2 and x_k, i.e., x_2 is second smallest entry
- After $n-1$ steps, x is sorted
- Note the structure (follow it for your exercises)
  - The length of the vector is a constant in `main`
    - i.e., the length cannot be changed
  - It is input parameter in `selectionSort`
    - i.e., the function works for any length


```
#include <stdio.h>
#define DIM 5

void scanVector(double input[], int dim) {
  int j = 0;
  for (j=0; j<dim; j=j+1) {
    input[j] = 0;
    printf("%d: ",j);
    scanf("%lf",&input[j]);
  }
}

void printVector(double output[], int dim) {
  int j = 0;
  for (j=0; j<dim; j=j+1) {
    printf("%f ",output[j]);
  }
  printf("\n");
}

void selectionSort(double vector[], int dim) {
  int j, k, argmin;
  double tmp;
  for (j=0; j<dim-1; j=j+1) {
    argmin = j;
    for (k=j+1; k<dim; k=k+1) {
      if (vector[argmin] > vector[k]) {
        argmin = k;
      }
    }
    if (argmin > j) {
      tmp = vector[argmin];
      vector[argmin] = vector[j];
      vector[j] = tmp;
    }
  }
}

main() {
  double x[DIM];
  scanVector(x,DIM);
  selectionSort(x,DIM);
  printVector(x,DIM);
}
```

Complexity

- Complexity of algorithms
- Landau symbol $\mathcal{O}$
- `time.h`, `clock_t`, `clock()`

Computational complexity

- Computational complexity of an algorithm
  - Amount of time, storage and/or other resources necessary to execute it
  - Comparison/evaluation of algorithms
  - Difference approaches
- Recall: An algorithm is a finite sequence of unambiguous operations which specify how to solve a problem
- Time complexity of an algorithm
  - Number of required elementary operations
    - Assignments
    - Comparisons
    - Arithmetic operations
  - Language-specific operations do not count
    - Declarations & Initializations
    - Loops, conditional statements, etc.
    - Counters
- Worst-case time complexity
  - Maximum number of operations required for inputs of given size

Example: Maximum of a vector

```
1  double maximum(double vector[], int n) {
2    int i = 0;
3    double max = 0;
4
5    max = vector[0];
6    for (i=1; i<n; i=i+1) {
7      if (vector[i] > max) {
8        max = vector[i];
9      }
10   }
11
12   return max;
13 }
```

- Complexity computation:
  - 1 assignment $\rightsquigarrow$ Line 5
  - In each step of the for loop $\rightsquigarrow$ Lines 6–10
    - 1 comparison $\rightsquigarrow$ Line 7
    - 1 assignment (worst case!) $\rightsquigarrow$ Line 8
- Loops yield sum of operations
  - i.e., for in line 6 implies $\sum_{i=1}^{n-1}$
- Altogether:

$$1 + \sum_{i=1}^{n-1} 2 = 1 + 2(n-1) = 2n - 1$$

Landau symbol $\mathcal{O}$ (= big O)

- Often only order of magnitude of complexity is interesting
- Definition: $f = \mathcal{O}(g)$ as $x \to x_0$
  - if $\limsup_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty$
  - i.e., $|f(x)| \leq C\,|g(x)|$ as $x \to x_0$
  - i.e., f grows at most like g
- Example: Maximum of a vector of length n
  - Complexity is $2n - 1 = \mathcal{O}(n)$ as $n \to \infty$
- Often 'as $n \to \infty$' is omitted
  - Standard choice (asymptotic complexity)
- In words:
  - Algorithm has linear complexity, if complexity is $\mathcal{O}(n)$ for problems of size n
    - e.g., Search for maximum of a vector
  - Algorithm has quasilinear complexity, if complexity is $\mathcal{O}(n \log n)$ for problems of size n
  - Algorithm has quadratic complexity, if complexity is $\mathcal{O}(n^2)$ for problems of size n
  - Algorithm has cubic complexity, if complexity is $\mathcal{O}(n^3)$ for problems of size n

Matrix-vector multiplication

```
1  void MVM(double A[], double x[], double b[],
2           int m, int n) {
3    int i = 0;
4    int j = 0;
5
6    for (j=0; j<m; j=j+1) {
7      b[j] = 0;
8      for (k=0; k<n; k=k+1) {
9        b[j] = b[j] + A[j+k*m]*x[k];
10     }
11   }
12 }
```

- In each step of for loop over j ⇝ Lines 6–11
  - 1 assignment ⇝ Line 7
  - In each step of for loop over k ⇝ Lines 8–10
    - 1 multiplication ⇝ Line 9
    - 1 addition ⇝ Line 9
    - 1 assignment ⇝ Line 9
- Altogether:

$$\sum_{j=0}^{m-1}\Big(1+\sum_{k=0}^{n-1}3\Big) = m + 3mn$$

- Complexity $\mathcal{O}(mn)$
  - i.e., complexity $\mathcal{O}(n^2)$ for $m = n$
  - i.e., quadratic complexity for $m = n$
- Indexing `j+k*m` in line 9 does not contribute

Search in a vector

```
int search(int vector[], int value, int n) {

  int j = 0;

  for (j=0; j<n; j=j+1) {
    if (vector[j] == value) {
      return j;
    }
  }

  return -1;
}
```

- Task:
  - Seek index `j` with `vector[j]` $=$ `value`
  - Return value `-1` if not existing
- Comparison with `==` is fine for data type `int`
  - Be careful with `double` (more details later)
- In each step of for loop over j
  - 1 comparison
- Altogether:

$$\sum_{j=0}^{n-1} 1 = n$$

- Complexity $\mathcal{O}(n)$

Binary search in sorted vector

```
int binSearch(int vector[], int value, int n) {

  int j = 0;
  int start = 0;
  int end = n-1;

  for ( ; start <= end ; ) {
    j = 0.5*(end+start);
    if (vector[j] == value) {
      return j;
    }
    else if (vector[j] > value) {
      end = j-1;
    }
    else {
      start = j+1;
    }
  }

  return -1;
}
```

- Assumption: Vector is sorted in ascending order
- Adapt ideas of bisection method
  - Consider halved vector, if `vector[j]` $\neq$ `value`
- Question: How many iterations has the algorithm?
  - Each step halves the vector
  - If n is even, $n/2^k = 1$
  - Hence, $k = 1 + \log_2 n$ steps at most
    - In each step 2 comparisons, 2 assignments, 1 multiplication, 2 additions/subtractions
- Complexity $\mathcal{O}(\log_2 n)$, i.e., logarithmic complexity
  - Sublinear complexity $\mathcal{O}(\log_2 n) \ll \mathcal{O}(n)$

Selection sort

```
void selectionSort(int vector[], int n) {
  int j = 0;
  int k = 0;
  int argmin = 0;
  double tmp = 0;

  for (j=0; j<n-1; j=j+1) {
    argmin = j;
    for (k=j+1; k<n; k=k+1) {
      if (vector[argmin] > vector[k]) {
        argmin = k;
      }
    }
    if (argmin > j) {
      tmp = vector[argmin];
      vector[argmin] = vector[j];
      vector[j] = tmp;
    }
  }
}
```

- In each step of for loop over j
  - 1 assignment
  - In each step of for loop over k
    - 1 comparison
    - 1 assignment (worst case!)
  - 1 comparison
  - 3 assignments (worst case!)
- quadratic complexity $\mathcal{O}(n^2)$, because:

$$\sum_{j=0}^{n-2}\Big(5+\sum_{k=j+1}^{n-1}2\Big) = 5(n-1) + \sum_{j=0}^{n-2}\big((n-(j+1)\big)2$$

$$= 5(n-1) + 2\sum_{k=1}^{n-1}k = 5(n-1) + 2\,\frac{n(n-1)}{2}$$

Time measurement in C

- Why time measurement?
  - Comparison of algorithms/implementations
  - Validation of theoretical considerations
- Theoretical assumptions
  - Linear complexity
    - Problem size $n \Rightarrow Cn$ operations
    - Problem size $kn \Rightarrow Ckn$ operations
    - i.e., 3× problem size ⇒ 3× execution time
  - Quadratic complexity
    - Problem size $n \Rightarrow Cn^2$ operations
    - Problem size $kn \Rightarrow Ck^2n^2$ operations
    - i.e., 3× problem size ⇒ 9× execution time
  - etc.
- E.g., program takes 1 s for $n = 1000$
  - Complexity $\mathcal{O}(n) \Rightarrow$ 10 s for $n = 10000$
  - Complexity $\mathcal{O}(n^2) \Rightarrow$ 100 s for $n = 10000$
  - Complexity $\mathcal{O}(n^3) \Rightarrow$ 1000 s for $n = 10000$
- Library `time.h`
  - Data type `clock_t` for time variables (do not forget type casting for printing)
  - Function `clock()` returns execution time since program start
  - Constant `CLOCKS_PER_SEC` for conversion: `time_variable/CLOCKS_PER_SEC` returns quantity in seconds

Example: Time measurement

```c
#include <stdio.h>
#include <time.h>

#define DIM 1000
#define VAL 500

int search(int vector[], int value, int n);
int binSearch(int vector[], int value, int n);
void selectionSort(int vector[], int n);

main() {
  clock_t t1;
  clock_t t2;
  int i = 0;
  int v[DIM];

  for(i=0; i<DIM; i=i+1) {
    printf("v[%d]=",i);
    scanf("%d",&v[i]);
  }

  t1 = clock();
  i = search(v,VAL,DIM);
  t2 = clock();

  printf("search: %f\n", (double)(t2-t1)/CLOCKS_PER_SEC);

  t1 = clock();
  selectionSort(v,DIM);
  t2 = clock();

  printf("selection sort: %f\n",
                        (double)(t2-t1)/CLOCKS_PER_SEC);

  t1 = clock();
  i = binSearch(v,VAL,DIM);
  t2 = clock();

  printf("binary search: %f\n",
         (double)(t2-t1)/CLOCKS_PER_SEC);
}
```


Runtime comparison

	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log_2 n)$
n	search	selectionSort	binSearch
1.000	0.00	0.00	0.00
2.000	0.00	0.00	0.00
4.000	0.00	0.01	0.00
8.000	0.00	0.06	0.00
16.000	0.00	0.25	0.00
32.000	0.00	1.03	0.00
64.000	0.00	4.12	0.00
128.000	0.00	16.55	0.00
256.000	0.00	64.31	0.00
512.000	0.00	257.25	0.00
1.024.000	0.00	≥ 18min	0.00
2.048.000	0.01	≥ 72min	0.00
4.096.000	0.01	≥ 4,5h	0.00
8.192.000	0.02	≥ 18h	0.00
16.384.000	0.04	≥ 3d	0.00
32.768.000	0.08	≥ 12d	0.00
65.536.000	0.15	≥ 1,5m	0.00
131.072.000	0.29	≥ 6m	0.00
262.144.000	0.60	≥ 2y	0.00
524.288.000	1.18	≥ 8y	0.00
1.048.576.000	2.53	≥ 32y	0.00

- Logarith. complexity nice, as $2^{30} > 1.048.576.000$
- Also linear complexity yields good execution time
- Quadratic complexity for large n noticeable
- Algorithms should have minimum complexity
  - One of the tasks of numerical mathematics
  - Not always possible

Comments

- Why comments?
- `//`
- `/* ... */`

Comments

- Comments are ignored by compiler
- Comments help read/understand the code
- Comments are necessary
  - to be able to understand even own code after some time
  - to help other people understand the code
- Comments are very useful during debugging
  - Commenting/uncommenting localized part of the source code 'to see what happens'
  - e.g., when dealing with syntax errors
- Important rules
  - Avoid special characters
  - Do not be mean, do not overdo
  - Usually at the beginning of the code, there is a comment with author and date of last change
    - Avoid conflicts with old versions...

Comments in C

```
1  #include <stdio.h>
2
3  main() {
4    // printf("1 ");
5    printf("2 ");
6    /*
7      printf("3");
8      printf("4");
9    */
10   printf("5");
11   printf("\n");
12 }
```

- In C, there are two types of comments:
  - Single-line comments
    - Start with // and until the end of the line
    - e.g., line 4
    - Originated from C++ syntax
  - Multi-line comments
    - Everything between /* (begin) and */ (end)
    - e.g., lines 6–9
    - /* ... */ cannot be nested (syntax error)
- Suggestion for a possible personal convention
  - Use // for *real* comments
  - Use /* ... */ for debugging
- Output:

  2 5