Arrays
- In MATLAB and C
- Pointer arithmetic

Sorting
- The sorting problem
- Insertion sort
- Algorithmic complexity

Divide & conquer
- Solving problems recursively
- Merge sort
- Bisection root finding

Linked list
- Search, insertion, deletion

Trees
- Binary search trees

Graphs
- Minimum spanning tree

Arrays
An array is a data structure containing a numbered (indexed) collection of items of a single data type.

In MATLAB arrays are primitive types.

In C, arrays are compound types. Furthermore, C arrays are much more limited than MATLAB’s.

/* Define, initialise, and access an array of three integers in C */
int a[3] = {10, 20, 30};
int sum = a[0] + a[1] + a[2];

/∗ Arrays of custom data types are supported too ∗/
VTOLState states[100];
for (t = 1; t < 100; t++) {
    states[t].position = states[t-1].position + states[t-1].velocity + 0.5*g;
    states[t].velocity = states[t-1].velocity + g - getThrust(states[t-1], burnRate) / states[t-1].mass;
    states[t].mass = states[t-1].mass - burnRate * escapeVelocity;
}
In C an array is represented as a sequence of records at consecutive memory addresses.

```c
/* array of five doubles */
double A[5];
/* get a pointer to the third element */
double *pt = &A[2];
```

Two (and more) dimensional arrays are simply arrays of arrays.

```c
/* A 2x5 array */
double A[2][5];
```

### Lecture 4 outline

- **Arrays**
  - In MATLAB and C
  - Pointer arithmetic

- **Sorting**
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity

- **Divide & conquer**
  - Solving problems recursively
  - Merge sort
  - Bisection root finding

- **Linked list**
  - Search, insertion, deletion

- **Trees**
  - Binary search trees

- **Graphs**
  - Minimum spanning tree

- **Static vs dynamic arrays in C**
  - This C statement defines an array `a` of five integers
    ```c
    int A[5];
    ```
  - The size is *static* because it is specified before the program is compiled. What if the size needs to be adjusted at run-time?
  - The solution is to *allocate dynamically* the required memory:
    ```c
    int arraySize = 5;
    int *A = malloc(sizeof(int) * arraySize);
    ```
  - Note that `a` is declared as a *pointer* to an `int`, not as an array. However, the array access operator `[]` can still be used. E.g. `a[1] = 2`
  - **Pointer math**: `a[n]` is the same as `(*(a + n))`
  - E.g. `a[0]` is the same as dereferencing the pointer `*(a)`
  - Under the hood, the address stored by `a` is incremented by `n * sizeof(int)` to account for the size of the pointed elements

### Sorting

**Problem**: sort an array of numbers in non-decreasing order.

- There are many algorithms to do this: bubble sort, merge sort, quick sort, ...
- We will consider three aspects:
  - Describing the algorithm formally.
  - Proving its correctness.
  - Evaluating its efficiency.
- We start from the **insertion sort algorithm**
  - Input: an array of numbers.
  - Output: the numbers sorted in non-decreasing order.
  - Algorithm: initially the sorted output array is empty. At each step, remove an element from the input array and insert it into the output array at the right place.
The insertion procedure that extends a sorted array by inserting a new element into it:

% Input: array A of size \( \geq n \) such that \( A[1] \leq \ldots \leq A[n-1] \)
% Output: permuted array such that \( A[1] \leq \ldots \leq A[n-1] \leq A[n] \)

function \( A = \text{insert}(A, n) \)
\[
\begin{align*}
i &= n \\
% the invariant is true here \\
\text{while } i > 1 \text{ and } A[i-1] > A[i] \\
\quad &\text{swap}(A[i-1], A[i]) \\
i &= i - 1 \\
% the invariant is true here \\
\end{align*}
\]
end

\( \text{Insertion sort} \)

% Input: an array \( A \) with \( n \) elements
% Output: array \( A \) such that \( A[1] \leq A[2] \leq \ldots \leq A[n] \)

function \( A = \text{insertionSort}(A, n) \)
\[
\begin{align*}
i &= 1 \\
% the invariant is true here (A) \\
\text{while } i < n \\
\quad &i = i + 1 \\
\quad &A = \text{insert}(A, i) \\
% the invariant is true here (B) \\
\end{align*}
\]
end

\( \text{Loop invariant: the first } i \text{ elements are sorted: } A[1] \leq A[2] \leq \ldots \leq A[i] \)

Proof by induction

*base case (A) \( i = 1: A[1] \) is sorted

*inductive step (B) \( i \geq 1: \) at iteration \( i \) the \( \text{insert()} \) procedure sorts \( A[1], \ldots, A[i] \) provided that \( A[1], \ldots, A[i-1] \) are sorted. The latter is given by the invariant at iteration \( i - 1. \)
Algorithmic complexity

- The time complexity of an algorithm is the maximum number of elementary operations \( f(n) \) required to process an input of size \( n \). Its space complexity is the maximum memory required.

- It often suffices to determine the order of the complexity \( g(n) \): linear \( n \), squared \( n^2 \), polynomial \( n^k \), logarithmic \( \log(n) \), exponential \( \exp(n) \), ... We say that the order of \( f(n) \) is \( g(n) \) and we write \( f(n) = O(g(n)) \) if:

\[
\exists a, n_0 : \forall n \geq n_0 : f(n) \leq ag(n)
\]

Example: insertion sort

- The size of the input is the number \( n \) of elements to sort.
- The space complexity is \( O(n) \) as the algorithm stores only the elements and a constant number of local variables.
- The time complexity of \( \text{insert}() \) is \( O(m) \) as the while loop is executed at most \( m \) times. The time complexity of \( \text{insertionSort}() \) is \( O(n^2) \) because

\[
\sum_{m=1}^{n} m = \frac{(n + 1)n}{2} = O(n^2)
\]

Divide and conquer

- **Divide and conquer** is a recursive strategy applicable to the solution of a wide variety of problems.

- The idea is to split each problem instance into two or more smaller parts, solve those, and recombine the results.

% Divide and conquer pseudocode

```
solution = solve(problem)
If problem is easy, compute solution
Else
  Subdivide problem into subproblem1, subproblem2, ...
  sol1 = solve(subproblem1), sol2 = solve(subproblem2), ...
  Get solution by combining sol1, sol2, ...
```

- Note the recursive call. Divide and conquer is naturally implemented as a recursive procedure.

- Some of the best known and most famous (and useful) algorithms are of this form, notably quicksort and the Fast Fourier Transform (FFT).

Complexity of divide and conquer

- Assume that the cost of splitting and merging a subproblem of size \( m \) is \( O(m) \) (linear) and that the cost of solving a subproblem of size \( m = 1 \) is \( O(1) \).

<table>
<thead>
<tr>
<th>O(n log2 n)</th>
<th>split &amp; merge</th>
<th>solve</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 problem of size 8</td>
<td>1 x 8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>2 problems of size 4</td>
<td>2 x 4</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>3 problems of size 2</td>
<td>4 x 2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>8 problems of size 1</td>
<td>0</td>
<td>8 x 1</td>
<td>8</td>
</tr>
</tbody>
</table>

Lecture 4 outline

- Arrays
  - In MATLAB and C
  - Pointer arithmetic
- Sorting
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- Divide & conquer
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
- Linked list
  - Search, insertion, deletion
- Trees
  - Binary search trees
- Graphs
  - Minimum spanning tree
The **merge sort** algorithm sorts an array \( A \) by divide and conquer:

**Split**: divide \( A \) into two halves \( A_1 \) and \( A_2 \).

**Merge**: iteratively remove from the beginning of the sorted \( A_1 \) and \( A_2 \) the smallest element and append it to \( A \).

**Base case**: if \( A \) has one element only it is sorted.

```
function A = mergeSort(A)
    n = length(A)
    if n == 1 then
        return A
    end
    k = floor(n / 2)
    A1 = A(1:k)
    A2 = A(k+1:end)
    A1 = mergeSort(A1)
    A2 = mergeSort(A2)
    return merge(A1, A2)
end
```

```
function A = merge(A1, A2)
    i1 = 1, i2 = 1
    m1 = length(A1), m2 = length(A1)
    while i1 <= m1 and i2 <= m2
        if A1[i1] <= A2[i2]
            A[i1+i2-1] = A1[i1], i1 = i1 + 1
        else
            A[i1+i2-1] = A2[i2], i2 = i2 + 1
        end
    end
end
```

---

**Insertion vs Merge Sort**

- The two sorting algorithms have **different complexities**:
  - insertion: \( O(n^2) \)
  - merge: \( O(n \log(n)) \)

- Plotting time vs size in loglog coordinates should give a line of slope:
  - \( 2 \) for insertion sort
  - \( \sim 1 \) for merge sort

- This is verified experimentally in the figure.

---

**Root finding**

- **Problem**: find a root of a non-linear scalar function \( f(x) \), i.e. a value of \( x \) such that \( f(x) = 0 \).

- **Assumption**: \( f(x) \) is a continuous function defined in the interval \([a, b]\); furthermore, \( f(a)f(b) < 0 \).

- The **bisection algorithm** is a divide and conquer strategy to solve this problem.

```
function bisect(f, a, b)
    m = (a + b) / 2
    if f(m) close to zero then return mu
    if f(m) * f(a) > 0
        return bisect(f, m, b)
    else
        return bisect(f, a, m)
end
```
A limitation of arrays is that inserting an element into an arbitrary position is $O(n)$. This is because existing elements must be shifted (moved in memory) in order to make space for the new one.

Linked lists solve this problem by using pointers:

In C a list data type could be defined as follows:

```c
/* List element datatype */
typedef struct ListElement_
{
    struct ListElement_*next;
    double value;
} ListElement;

/* List datatype */
typedef ListElement List;
```

The list could be defined as a pointer to its first element. It is customary to use instead a fake list element (which contains such pointers) to simplify coding functions using the list.

### Inserting an element into a linked list

To **insert** an element into a list, use pointers to create a “bypass” at cost $O(1)$.

```c
/* Insert an element in a list */
void insert(ListElement *prev, ListElement *element)
{
    element->next = prev->next;
    prev->next = element;
}
```

**Example usage**

```c
/* Create an empty list */
List list = NULL;
list->next = NULL;

/* Create an element */
ListElement *element = malloc(sizeof(ListElement));
    element->next = NULL;
    element->value = 42.0;

/* Insert at the beginning of the list */
insert(&list, element);

/* Insert after element */
insert(element, element2);
```

### Removing an element from a linked list

To **insert** an element into a list, use pointers to create a “bypass” at cost $O(1)$.

```c
/* Remove an element */
ListElement *remove(ListElement *prev)
{
    ListElement removed = prev->next;
    if (removed != NULL) {
        prev->next = removed->next;
    }
    return removed;
}
```

**Example usage**

```c
/* Remove the element after previous */
ListElement *removed;
removed = remove(previous);

/* Do not forget to release the memory if needed */
if (removed != NULL) {
    free(removed);
}
```
Lecture 4 outline

- Arrays
  - In MATLAB and C
  - Pointer arithmetic
- Sorting
  - The sorting problem
  - Insertion sort
  - Algorithmic complexity
- Divide & conquer
  - Solving problems recursively
  - Merge sort
  - Bisection root finding
- Linked list
  - Search, insertion, deletion
- Trees
  - Binary search trees
- Graphs
  - Minimum spanning tree

Binary search tree

- A binary search tree is a binary tree such that the value of each node is
  - at least as larger as the value of its left descendants
  - smaller all the values of its right descendants
- Its main purpose is to support the binary search algorithm.

Binary search algorithm

- Problem: find a node with value x in a binary search tree.
- The binary search algorithm searches for x recursively, using the binary search tree property to descend only into one branch every time.

```c
function node = binarySearch(node, x)
    if node == NULL return NULL
    if node.value == x return node
    if x > node.value
        return binarySearch(node.right, x)
    else
        return binarySearch(node.left, x)
    end
end
```

- The cost is O(h) where h is the depth of the binary tree.
- Typically h = O(log n), where n is the number of nodes in the tree. Hence the search cost is O(log n), sub-linear.
- Compare this with the O(n) cost of searching in an array or a linked list.
Lecture 4 outline

Arrays
In MATLAB and C
Pointer arithmetic
Sorting
The sorting problem
Insertion sort
Algorithmic complexity
Divide & conquer
Solving problems recursively
Merge sort
Bisection root finding

Linked list
Search, insertion, deletion
Trees
Binary search trees
Graphs
Minimum spanning tree

An (directed) graph is a set of vertices $V$ and edges $E$ connecting the edges. An undirected graph is a graph such that for each edge $(u,v)$ there is an opposite edge $(v,u)$.

% MATLAB representation
edges = [1 2 2 3 4 5 5 6 7 2 3 6 4 5 6 8 7 1 2 2 3 4 5 5 6];

An alternative representation of a graph is the adjacency matrix $A$. $A$ is a $n \times n$ matrix such that $A(u,v) = 1$ if, and only if, $(u,v) \in E$.

A spanning tree is a subset of the edges forming a tree including all the nodes.

A minimum spanning tree (MST) is a spanning tree such that the sum of the edge weights is minimal.

A famous algorithm to compute the MST is explored in the tutorial sheet.