Introduction to Computer Science
Lecture 11

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today's topics

Algorithms on sequences (arrays)

- Sequence shuffling
- Sequence sorting
- Sequence searching
- Sequence merging

Algorithms and Pseudo-Code

- It is a finite sequence of rigorous well-defined instructions, whose aim is to solve a specific problem.
- Algorithms describe computations.
  - They can be implemented as programs in a programming language
  - They can also be abstractly described using pseudo-code
    - Pseudo-code is an informal notation to abstractly describe programs
      - Strong syntactic rules of programming languages are omitted.
      - Types, operators, and control structures are used in a flexible way.
      - Subtasks can be referred to as functions/procedures (straighforward tasks, or complex tasks to be further refined/implemented later on)

Pseudo-Code Example

```
isPrime(int x) -> boolean {
    assume x is positive
    if x is 1, then return false
    else {
        for each i in [2, 3, …, (x-1)] do {
            if i divides x then return false
        }
        return true
    }
}
```

Shuffling a Sequence: The Fisher-Yates algorithm

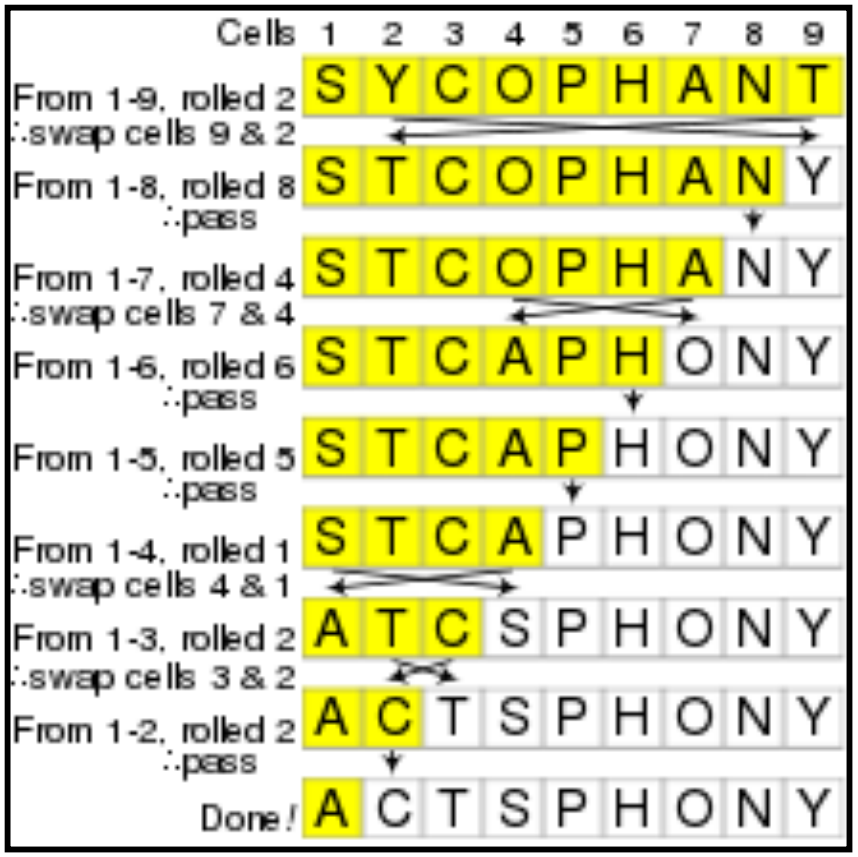
- Problem: randomly shuffle the elements of a sequence.
  - Input: a sequence s = [e1, …, en] of elements
  - Output: a random permutation of s.
- Fisher-Yates Algorithm:
  - Produces an unbiased random permutation (every possible output permutation is equally likely)
  - Shuffles sequence elements “in place” (without the need for additional space)

Shuffling a Sequence: The Fisher-Yates algorithm

```
shuffle(seq s) {
    for each i in [0..length(s)-1] do {
        choose random j in [i..length(s)-1]
        swap s[i] and s[j]
    }
}
```

Shuffling a Sequence: The Fisher-Yates algorithm

```
shuffle(seq s) {
    for i from length(s)-1 to 1 do {
        choose random j in [0..i]
        swap s[i] and s[j]
    }
}
```



Shuffling a Sequence: The Fisher-Yates algorithm

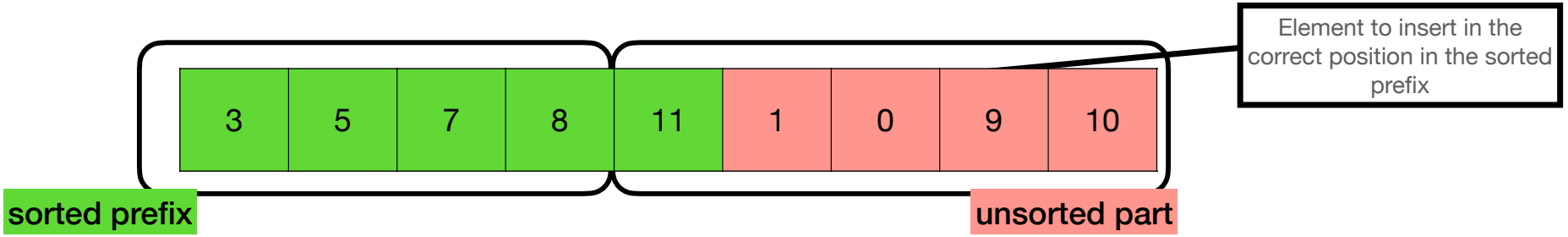
```c
void shuffle(int array[], int size) {
    srand(time(NULL));
    for (int i = 0; i < size; i++) {
        int j = (rand() % ((size-1) - i + 1)) + i ;
        int aux = array[i];
        array[i] = array[j];
        array[j] = aux;
    }
}
```

Sorting a Sequence: An Important CS Problem

- Problem: sorting a sequence of elements.
  - Input: a sequence s = [e1, …, en] of sortable elements, i.e., elements for which there exists a defined order.
  - Output: a permutation of s that is increasingly sorted.
- There exist many alternative algorithms to solve this problem, with different characteristics (efficiency, memory requirements, performance on particular data structures, performance on partially sorted sequences, …)

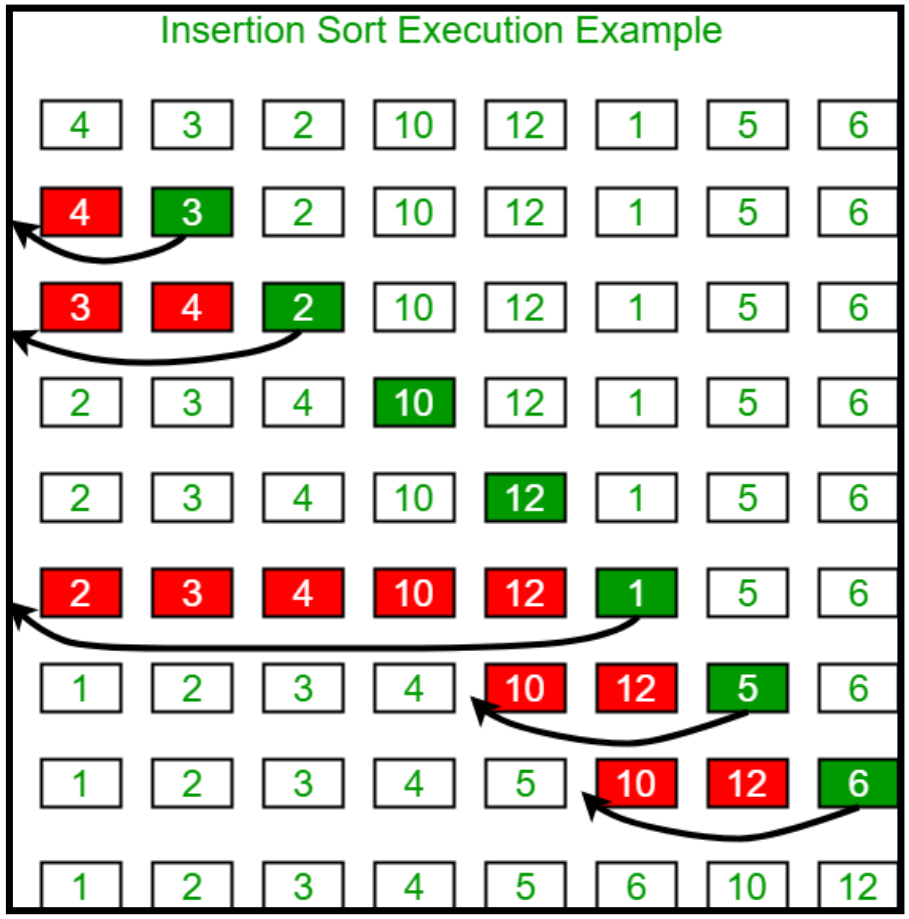
Sorting a Sequence with Insertion Sort

- Key idea:
  - Maintain a sorted prefix of the sequence.
  - At each step, insert the first element of the unsorted part into its sorted position of the sorted prefix, thus extending the sorted prefix by one.
  - When the unsorted part becomes empty, the sequence is sorted.



Sorting a Sequence with Insertion Sort

```
insertionSort(seq s) {
    for each i in [1..length(s)-1] do {
        j = i
        while (j > 0 and s[j-1]>s[j]) {
            swap s[j] and s[j-1]
        }
    }
}
```

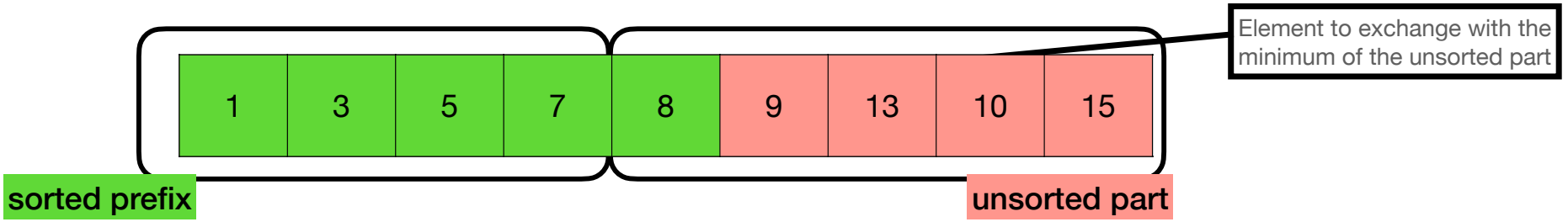


Sorting a Sequence with Insertion Sort

```
void insertionSort(int array[], int size) {
    for (int i = 1; i < size; i++) {
        int j = i;
        while (j > 0 && array[j-1] > array[j]) {
            int aux = array[j];
            array[j] = array[j-1];
            array[j-1] = aux;
        }
    }
}
```

Sorting a Sequence with Selection Sort

- Key idea:
  - Maintain a sorted prefix of the sequence, with all elements smaller than the unsorted part.
  - At each step, select the minimum of the unsorted part, and exchange it with the first element in the unsorted part, thus extending the sorted prefix by one.
  - When the unsorted part becomes empty, the sequence is sorted.



Sorting a Sequence with Selection Sort

```
selectionSort(seq s) {
    for each i in [0..length(s)-2] do {
        min_index = i
        for j from i+1 to length(s) -1 do {
            if s[j] < s[min_index] then {
                min_index = j
            }
        }
        swap s[j] and s[i]
    }
}
```

Sorting a Sequence with Selection Sort

```
void selectionSort(int array[], int size) {
    for (int i = 0; i < size-1; i++) {
        int min_index = i;
        for (int j = i+1; j < size; j++) {
            if (array[j] < array[min_index]) {
                min_index = j;
            }
        }
        int aux = array[i];
        array[i] = array[min_index];
        array[min_index] = aux;
    }
}
```

Searching in a Sequence

- Problem: given a sequence s and an element e, check if e belongs to s.
  - Input: a sequence s = [e1, …, en] of elements and an element x
  - Output: if x belongs to s, index i such that ei is x; -1 if x does not belong to s

Searching in a Sequence: Linear Search

```
linearSearch(seq s, elem x) -> int {
    x_index = -1
    i = 0
    while index == -1 and i < length(s) {
        if s[i] == x then x_index = i
        i = i + 1
   }
   return x_index
}
```

Searching in a Sorted Sequence

- Problem: given a *sorted* sequence s and an element e, check if e belongs to s.
  - Input: a sorted sequence s = [e1, …, en] of elements and an element x
  - Output: if x belongs to s, index i such that ei is x; -1 if x does not belong to s
- Linear search still works!
  - *But can we exploit the fact the s is sorted to speed up the search?*

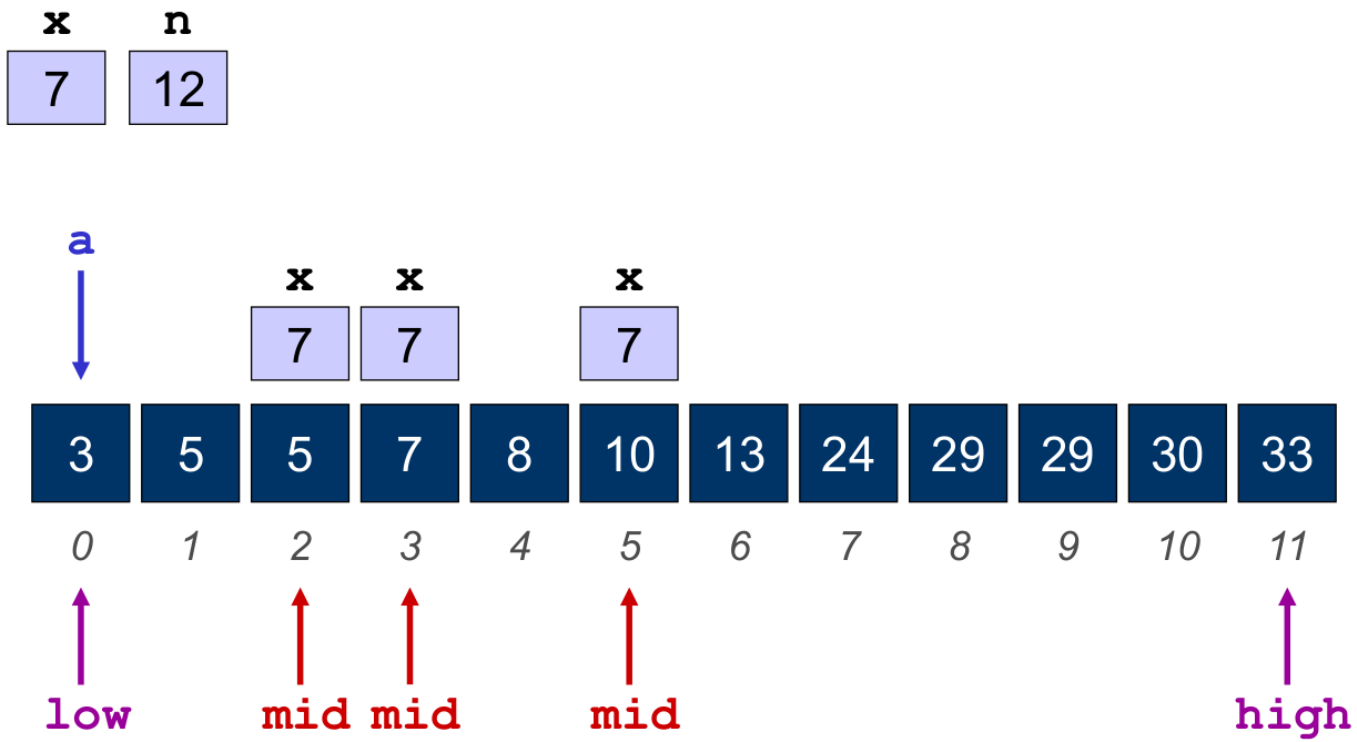
Searching in a Sorted Sequence: Binary Search

- Key idea:
  - look up the mid point of the sequence s
    - If element in mid point of s is x, return mid point
    - if element smaller than mid point of s, continue search in the first half
    - If element greater than mid point of s, continue search in the second half
    - Repeat the process until found, or subsequence to explore is empty

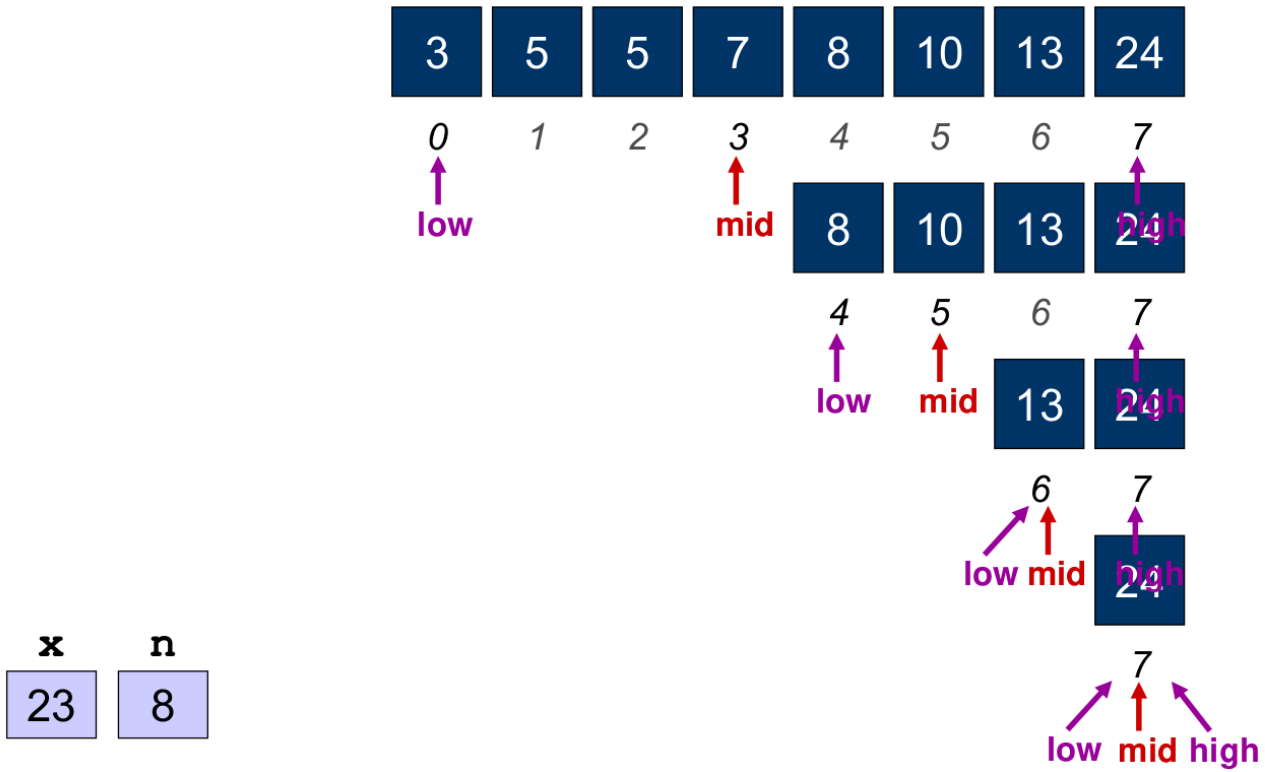
Searching in a Sorted Sequence: Binary Search

```
binarySearch(seq s, elem x) -> int {
    low = 0
    high = length(s) -1
    while low <= high {
        mid = (high-low)/2
        if s[mid] == x then return mid
        else {
            if s[mid] < x then low = mid + 1
            else high = mid - 1
        }
    }
    return -1
}
```

Searching in a Sorted Sequence: Binary Search



Searching in a Sorted Sequence: Binary Search



Merging Sorted Sequences

- Problem: given two *sorted* sequences s1 and s2, create a sorted sequence s3 that merges s1 and s2
  - Input: sorted sequences s1 and s2
  - Output: a sorted sequence s3 that is the permutation of the concatenation s1+s2

Merging Sorted Sequences

- Naïve algorithm:
  - Copy in s3 both s1 and s2 in a contiguous way (s3 is the concatenation of s1 and s2)
  - Sort s3
- *Can we take advantage of the fact that s1 and s2 are sorted to make the process more efficient?*

Merging Sorted Sequences: Linear Merge

- Key idea:
  - Maintain in s3 a sorted merge of two prefixes of s1 and s2
    - s1 and s2 have corresponding “non-treated” parts
  - At each step compare the first elements of the non-treated parts of s1 and s2.
    - Extend s3 with the smaller of the two, and reduce the untreated part of the sequence contributing the element to s3

Merging Sorted Sequences: Linear Merge

```
merge(seq s1, seq s2) -> seq {
    output = []
    i = 0
    j = 0
    while i < length(s1) and j < length(s2) {
        if s1[i] <= s2[j] then {
            output = output + s1[i]
            i = i + 1
        }
       else {
            output = output + s2[j]
            j = j + 1
        }
    }
    while i < length(s1)  {
        output = output + s1[i]
         i = i + 1
    }
    while j < length(s2)  {
        output = output + s2[j]
         j = j + 1
    }
    return output
}
```