University of Central Lancashire
UCLan

CO2412
Computational Thinking

Algorithm design strategies: Overview
Dynamic programming
Static and dynamic arrays
Python lists and the Tutorial 1.1 problem

Where opportunity creates success

Algorithm design strategies: Overview

CO2412

9th November 2021

Design strategy: Brute force

Initial analysis: What is the **space of all conceivable solutions** to the problem, for the given input? Check all parameters/options and devise an scheme that iterates over all the permitted values and their combinations.

Brute-force design strategy: Evaluate all potential solutions, one by one.

Strengths of the strategy: The code is easy to write, and it is easy to prove that it is correct. Beyond the initial analysis, not much needs to be figured out.

Weakness of the strategy: Reusing information might reduce the relevant number of candidate solutions. This is not done; instead, all are tried out.

While there are some problems that can be addressed in this way, most cannot; the space of solutions that need to be enumerated usually grows too fast.

Example problem: Maximum sublist sum

Specification of a function solving the *maximum sublist sum* problem

Precondition (of the function), *i.e.*, initial execution state: One argument is passed to the function, namely, a list of floating-point and/or integer numbers.

Postcondition (of the function), *i.e.*, final execution state: The function returns a sublist, *i.e.*, a contiguous part of the original list, such that the sum over all elements of the sublist is as large as possible.

Example: The list given by

x = [–147, **72, –49, 40, 46, 35, 26**, –69, 21, –5, –52, –40, 6, –133, 36]

has the maximum sublist x[1: 7] = [72, –49, 40, 46, 35, 26] with the sum 170.

Brute-force maximum-sublist-sum algorithm

```
def brute_force_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    for i in range(len(x)):
        for j in range(i+1, len(x)+1):
            sublist_sum = 0
            for k in range(i, j):
                sublist_sum += x[k]
            if sublist_sum > max_sublist_sum:
                left_idx = i
                right_idx = j
                max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Summary:

- Try out all possible sublists, with index i running over all possible left limits and j over all right limits (greater than i)
- Evaluate the sum of the elements of each sublist
- Keep track of the maximum; finally, return the maximum sublist

Design strategy: Greedy algorithms

Greedy algorithms are based on the idea of making the **best local improvement** (*i.e.*, the best immediately visible small change) to a partial solution. They consider **one candidate solution** only and build it up gradually.

Image source: City College Norwich

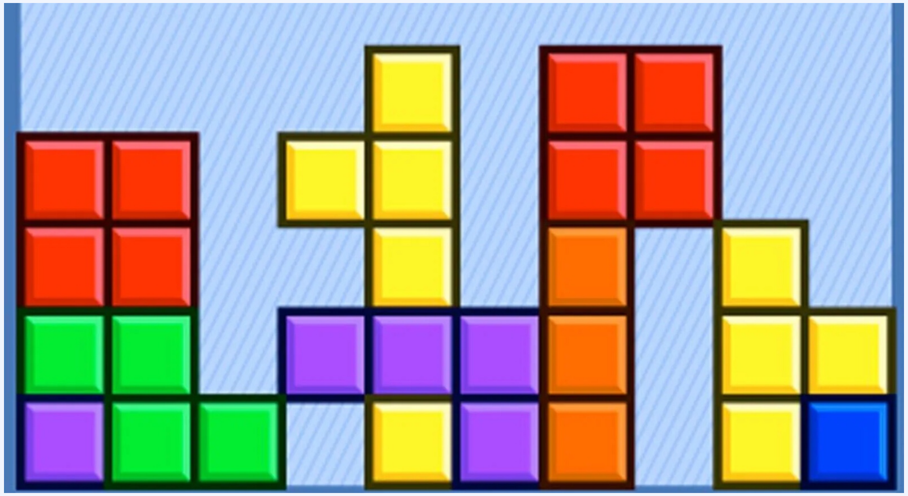


Image source: BBC

Strength: Systematic and easy to implement.

Weakness: It does not solve all problems correctly; but even then, it might return an acceptable suboptimal result or an approximation to the solution.

Selection sort (greedy)

```
def selection_sort(x):
    for i in range(len(x)):
        ● all before index i is sorted
        min_idx = i
        for j in range(i+1, len(x)):
            if x[j] < x[min_idx]:
                min_idx = j
        ● min_idx is the index of the smallest
          element from the unsorted part
        next_element = x.pop(min_idx)
        x.insert(i, next_element)
        ● all until (including) index i is sorted
    ● the whole list is sorted
```

Test list: [8, 58, 25, 48, 19, 39, 76, 6, 11, 86, 75]

Sorted part of the list:
[6]
Unsorted part of the list:
[8, 58, 25, 48, 19, 39, 76, 11, 86, 75]

Sorted part of the list:
[6, 8]
Unsorted part of the list:
[58, 25, 48, 19, 39, 76, 11, 86, 75]

...

Sorted part of the list:
[6, 8, 11, 19, 25, 39, 48, 58, 75]
Unsorted part of the list:
[76, 86]

Sorted part of the list:
[6, 8, 11, 19, 25, 39, 48, 58, 75, 76]
Unsorted part of the list:
[86]

Sorted part of the list:
[6, 8, 11, 19, 25, 39, 48, 58, 75, 76, 86]
Unsorted part of the list:
[]

Design strategy: Divide and conquer

Decomposition breaks a problem down into smaller subtasks. For the two decomposition techniques discussed today, divide-and-conquer and dynamic programming, subtasks are **subproblems**: Smaller versions of the problem.

The solution of a subproblem then is a **partial solution** for the whole problem.

First **divide**,

then **conquer**,

finally **combine** the results.

In divide-and-conquer, subproblems do not overlap (or they are assumed not to overlap). Each subproblem occurs once, and hence, **each partial solution is used only once**. It need not be stored anywhere beyond its single use.

Mergesort: Divide and conquer

List: [20, 22, 4, 89, 110, 52, 60, 79, 58, 9, 87]

sublist_size = 1

Merging x[0:1] = [20] with x[1:2] = [22]
Merged to x[0:2] = [20, 22]

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 110 52 60 79 58 9 87

Merging x[2:3] = [4] with x[3:4] = [89]
Merged to x[2:4] = [4, 89]

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 110 52 60 79 58 9 87

Merging x[4:5] = [110] with x[5:6] = [52]
Merged to x[4:6] = [52, 110]

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 52 110 60 79 58 9 87

Merging x[6:7] = [60] with x[7:8] = [79]
Merged to x[6:8] = [60, 79]

20 22 4 89 52 110 60 79 58 9 87
20 22 4 89 52 110 60 79 58 9 87

Merging x[8:9] = [58] with x[9:10] = [9]
Merged to x[8:10] = [9, 58]

20 22 4 89 52 110 60 79 58 9 87
20 22 4 89 52 110 60 79 9 58 87

(Nothing to be done for x[10].)

20 22 4 89 52 110 60 79 9 58 87

Mergesort: Divide and conquer

sublist_size = 2

Merging x[0:2] = [20, 22] with x[2:4] = [4, 89]	20 22 4 89 52 110 60 79 9 58 87
Merged to x[0:4] = [4, 20, 22, 89]	4 20 22 89 52 110 60 79 9 58 87
Merging x[4:6] = [52, 110] with x[6:8] = [60, 79]	4 20 22 89 52 110 60 79 9 58 87
Merged to x[4:8] = [52, 60, 79, 110]	4 20 22 89 52 60 79 110 9 58 87
Merging x[8:10] = [9, 58] with x[10:11] = [87]	4 20 22 89 52 60 79 110 9 58 87
Merged to x[8:11] = [9, 58, 87]	4 20 22 89 52 60 79 110 9 58 87

sublist_size = 4

Merging x[0:4] = [4, …] with x[4:8] = [52, …]	4 20 22 89 52 60 79 110 9 58 87
Merged to x[0:8] = [4, …, 110]	4 20 22 52 60 79 89 100 9 58 87
(Nothing to be done for x[8:11].)	4 20 22 52 60 79 89 100 9 58 87

sublist_size = 8

Merging x[0:8] = [4, …] with x[8:11] = [9, …]	4 20 22 52 60 79 89 100 9 58 87
Merged to x[0:11] = [4, …, 110]	4 9 20 22 52 58 60 79 87 89 100

Design strategies: Overview

We have seen many algorithm design elements in use so far, including:

- Case distinctions
- Recursive function calls
- Nested loops
- Dynamic data structures (lists, dictionaries, *etc.*)

Design strategies concern algorithm and code development at a more abstract level than that of its implementation. They are established approaches for designing algorithms; they all have their own strengths and weaknesses.

- **Brute force:** Check all possible solutions, determine the right/best one.
- **Greedy algorithms:** Build the solution step by step until it is complete.
- Decomposition by **divide-and-conquer** or by **dynamic programming**.

Design strategies: Overview

Brute force	Easy to implement and to verify
Greedy algorithms	Easy to implement, often very efficient
Decomposition techniques	Powerful by reduction to subproblems

Design strategies concern algorithm and code development at a more abstract level than that of its implementation. They are established approaches for designing algorithms; they all have their own strengths and weaknesses.

- **Brute force:** Check all possible solutions, determine the right/best one.
- **Greedy algorithms:** Build the solution step by step until it is complete.
- Decomposition by **divide-and-conquer** or by **dynamic programming**.

Design strategies: Overview

Brute force	Easy to implement and to verify	Scales with size of solution space, often forbiddingly expensive
Greedy algorithms	Easy to implement, often very efficient	Not all problems are accessible to this kind of approach
Decomposition techniques	Powerful by reduction to subproblems	Requires a thorough analysis of the problem and its subproblems

Design strategies concern algorithm and code development at a more abstract level than that of its implementation. They are established approaches for designing algorithms; they all have their own strengths and weaknesses.

- **Brute force:** Check all possible solutions, determine the right/best one.
- **Greedy algorithms:** Build the solution step by step until it is complete.
- Decomposition by **divide-and-conquer** or by **dynamic programming**.

Dynamic programming

University of Central Lancashire
UCLan

CO2412

9th November 2021

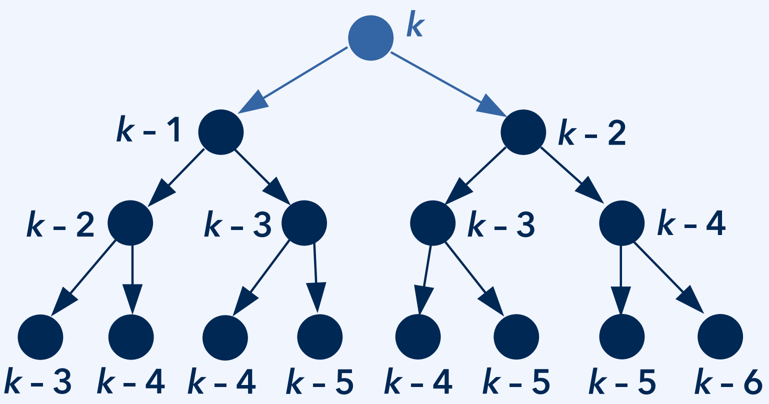
Divide-and-conquer: Limitations

Where the decomposition of a problem into smaller parts leads to mutually **overlapping subproblems**, divide-and-conquer (*e.g.*, implemented by multiple recursion) might recompute the same partial solution many times.

Fibonacci sequence

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}, \text{ for } k > 1$$

0, 1, 1, 2, 3, 5, 8, 13, …

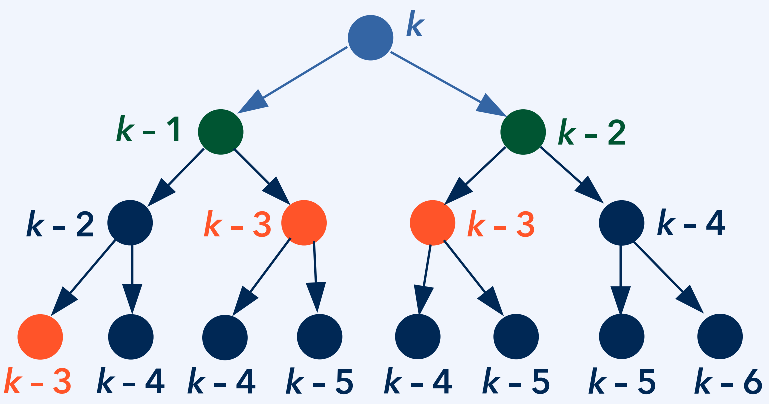


Fibonacci sequence: First attempt at decomposition.

Divide-and-conquer: Limitations

Where the decomposition of a problem into smaller parts leads to mutually **overlapping subproblems**, divide-and-conquer (*e.g.*, implemented by multiple recursion) might recompute the same partial solution many times.

The subproblems from the two branches (for $k - 1$ and $k - 2$) overlap: They both contain the $k - 3$ subproblem.



Fibonacci sequence: First attempt at decomposition.

Fibonacci sequence

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}, \text{ for } k > 1$$

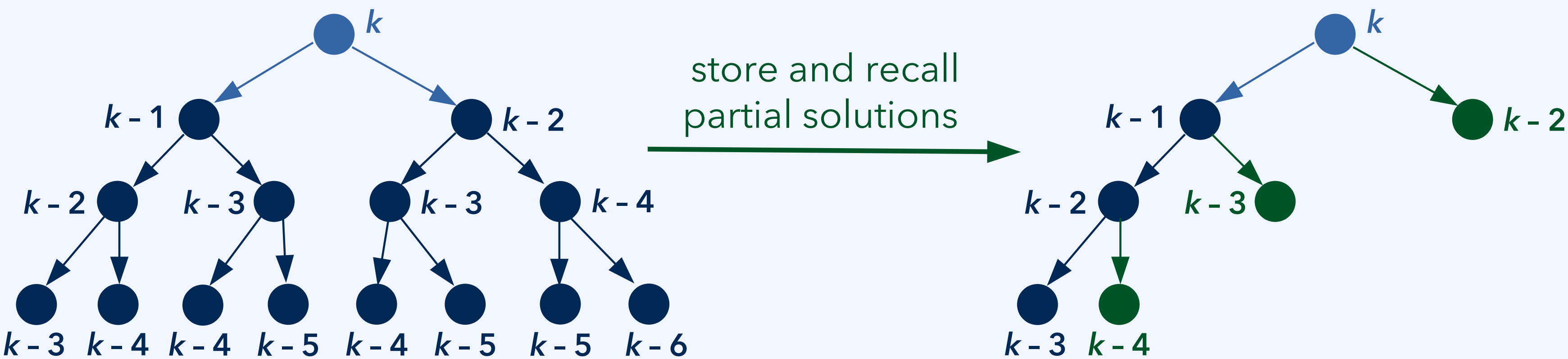
0, 1, 1, 2, 3, 5, 8, 13, …

Design strategy: Dynamic programming

University of Central Lancashire UCLan

Where the decomposition of a problem into smaller parts leads to mutually **overlapping subproblems**, divide-and-conquer (*e.g.*, implemented by multiple recursion) might recompute the same partial solution many times.

To improve the decomposition efficiency in such cases, it can help to store and recall **partial solutions**. This strategy is called **dynamic programming**.



Fibonacci sequence: First attempt at decomposition.

Fibonacci sequence: O(n) time solution by dynamic programming.

Kadane's algorithm: Dynamic programming

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0

    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum

    return x[left_idx: right_idx]
```

First, initialize the **best overall sublist** x[left_idx: right_idx] and the left boundary for the **best current sublist** x[i: j]; implicitly, initially, j = 0

For each $0 \leq j < n$,

- determine the best sublist x[i: j+1] with boundary j+1
- update information on the best sublist found so far

return the **best overall sublist**

Kadane's algorithm for the maximum sublist sum

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0
    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Remark

Kadane's algorithm is a result of design by dynamic programming.

A partial solution is stored and and recalled, and the subproblems of the maximum sublist problem are overlapping.

Divide-and-conquer vs. dynamic programming

Divide and conquer:

- The partial solutions (to subproblems) **do not need to be remembered**.
- Each partial solution is **used only once**, when it is combined with one or multiple other partial solutions in a single specific way.

Dynamic programming:

- Partial solutions are **stored and recalled** when required.
- Therefore, the same partial solution can be **used multiple times**, and it can be combined with other partial solutions in a variety of ways.

Divide-and-conquer vs. dynamic programming

Divide and conquer:

- **Subproblems do not overlap**, there is a genuine split into subproblems.
- The partial solutions (to subproblems) **do not need to be remembered**.
- Each partial solution is **used only once**, when it is combined with one or multiple other partial solutions in a single specific way.

Dynamic programming:

- Partial solutions are **stored and recalled** when required.
- There is the option (and expectation) that **subproblems overlap**.
- Therefore, the same partial solution can be **used multiple times**, and it can be combined with other partial solutions in a variety of ways.

Algorithm design strategies: Overview

Brute force	Easy to implement and to verify	Scales with size of solution space, often forbiddingly expensive
Greedy algorithms	Easy to implement, often very efficient	Not all problems are accessible to this kind of approach
Decomposition techniques	Powerful by reduction to subproblems	Requires a thorough analysis of the problem and its subproblems

Design strategies concern algorithm and code development at a more abstract level than that of its implementation. They are established approaches for designing algorithms; they all have their own strengths and weaknesses.

- **Brute force:** Check all possible solutions, determine the right/best one.
- **Greedy algorithms:** Build the solution step by step until it is complete.
- Decomposition by **divide-and-conquer** or by **dynamic programming**.

University of Central Lancashire
UCLan

Static and dynamic arrays

CO2412

9th November 2021

Static arrays

An array contains a sequence of elements of the same type, arranged **contiguously in memory**.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

In C/C++, the type of an array such as int[] is the same as the corresponding pointer type int*, *i.e.*, the array actually is a pointer. Its value is an address at which an integer is stored, namely, the memory address of the first element.

Static arrays

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. **It is most common to work with static arrays.**

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

In C/C++, the type of an array such as int[] is the same as the corresponding pointer type int*, *i.e.*, the array actually is a pointer. Its value is an address at which an integer is stored, namely, the memory address of the first element.

Remark

Static data structures can only change their content, *i.e.*, the values of their elements. Once they are allocated, **their size and structure cannot change**.

Static arrays

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. This supports fast access using **pointer arithmetics**.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

x = &(x[0]) **x + 4 = &(x[4])**

In C/C++, the type of an array such as int[] is the same as the corresponding pointer type int*, *i.e.*, the array actually is a pointer. Its value is an address at which an integer is stored, namely, the memory address of the first element.

Above, *x would evaluate to 34, and so would x[0].
The expression *(x + 4) would evaluate to 4, and so would x[4].

Static arrays

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. This supports fast access using **pointer arithmetics**.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

x = &(x[0]) x + 3 = &(x[3]) x + 6 = &(x[6])

In C/C++, the type of an array such as int[] is the same as the corresponding pointer type int*, *i.e.*, the array actually is a pointer. Its value is an address at which an integer is stored, namely, the memory address of the first element.

This is highly efficient since when x[i] is accessed, the compiler transforms this into accessing the memory address x + sizeof(int) * i.

Static arrays

Remark

In Python, numpy can be used to create an array, e.g., with x = np.array([34, 1, 7, 12, 3, 4, 7, 12]).

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
34	1	7	12	3	4	7	12

x = &(x[0])   x + 3 = &(x[3])   x + 6 = &(x[6])

Efficiency analysis:

Read/write access to an array element: O(1) time.
Deleting an element from the array: Impossible.
Extending the array by an element: Impossible.

Dynamic arrays

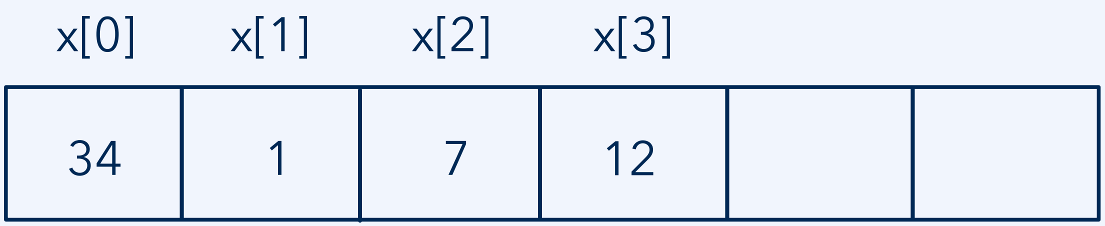
Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimziation level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future.

Dynamic arrays

Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimziation level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.



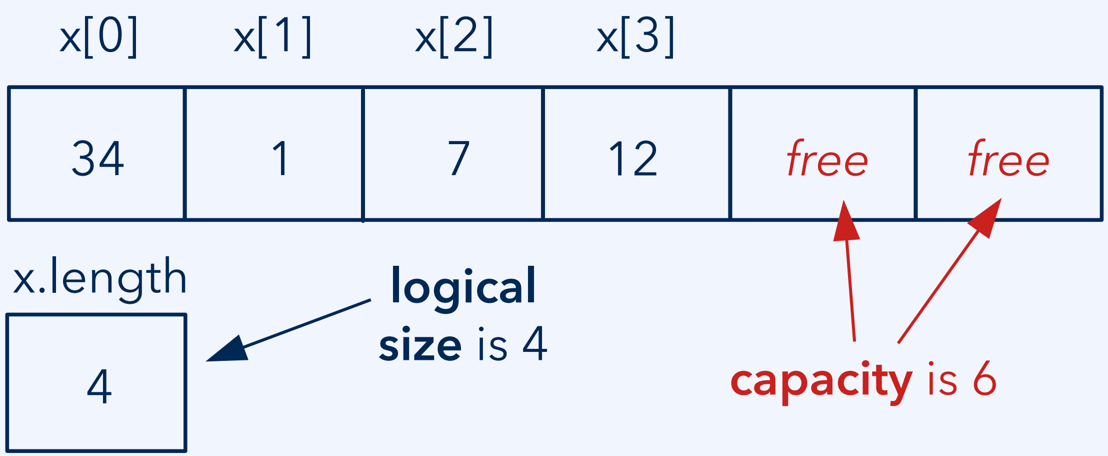
x = [34, 1, 7, 12]

Note: More memory is allocated than strictly necessary.
Like before, the elements are contiguously arranged in memory.

Dynamic arrays

Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimziation level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.



x = [34, 1, 7, 12]

x.append(3)
x.append(4)

Dynamic arrays

Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimziation level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
34	1	7	12	3	4

x.length
6

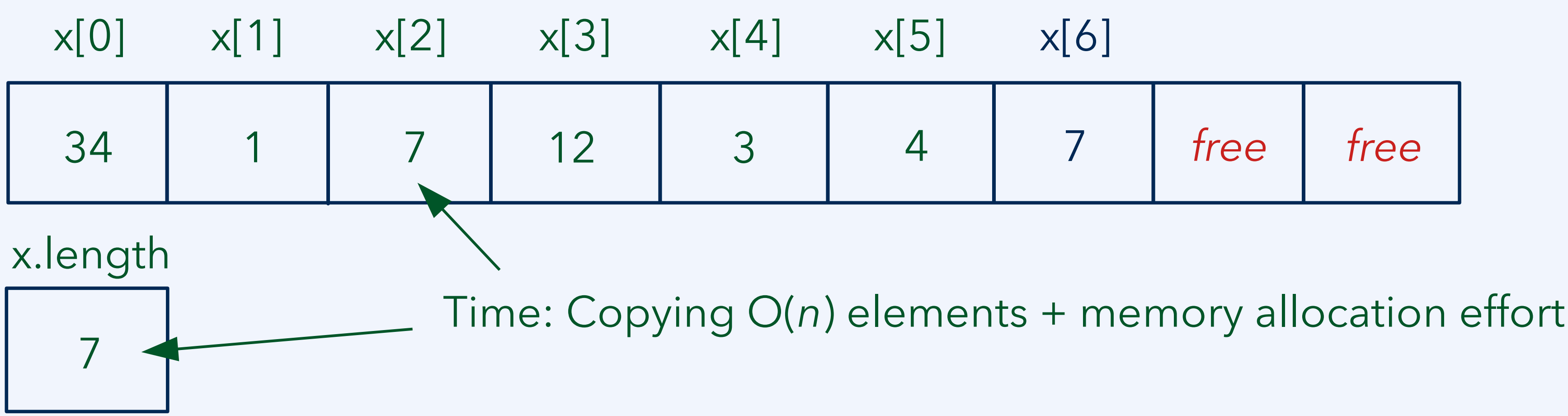
x = [34, 1, 7, 12]

x.append(3)
x.append(4)
x.append(7)

Dynamic arrays

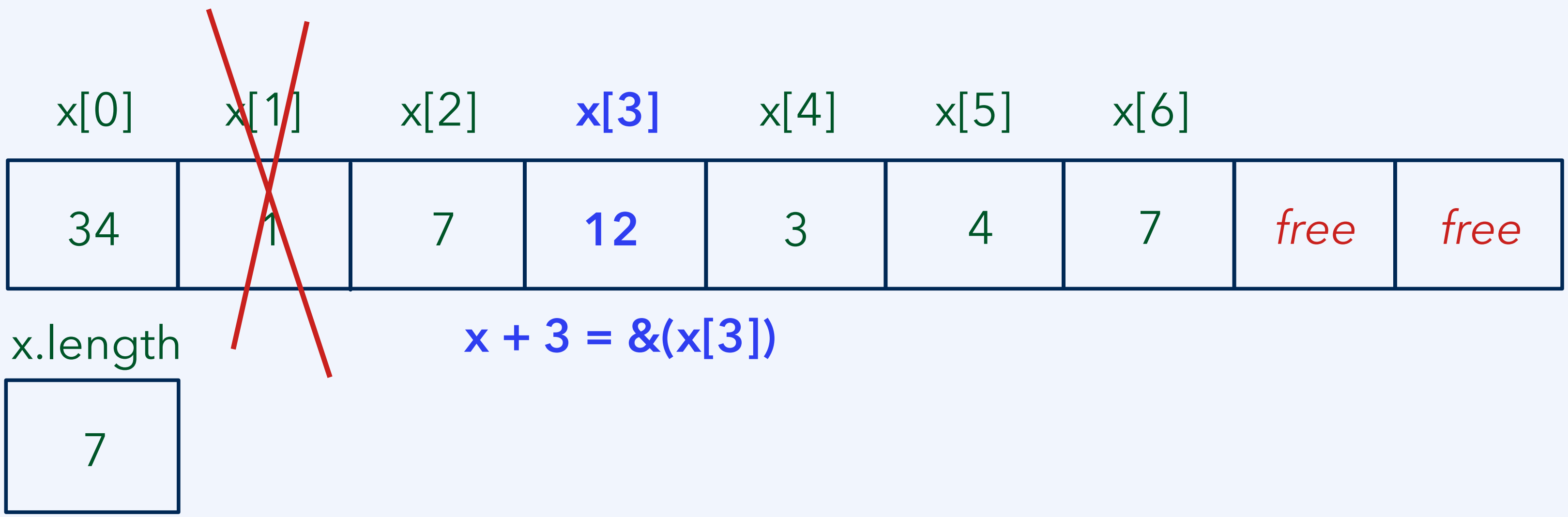
Conventional arrays are **static data structures**. Their size in memory is constant, and memory needs to be allocated only once, *e.g.*, at declaration time. (Details depend on programming language, compiler, flags/optimziation level, *etc.*).

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.



Dynamic arrays: Efficiency analysis

- **Read/write access to an array element: O(1) time.**
  Address of the i-th element computable by pointer arithmetics.
- **Deleting an element from the array?**



Dynamic arrays: Efficiency analysis

- Read/write access to an array element: O(1) time.
  Address of the i-th element computable by pointer arithmetics.
- Deleting an element from the array: **O(1) at the end, O(*n*) elsewhere.**
  All the elements with greater indices need to be shifted.
- **Extending the array by one element?**

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]			
34	7	12	3	4	7	*free*	*free*	*free*

x.length
6

Dynamic arrays: Efficiency analysis

- Read/write access to an array element: O(1) time.
  Address of the i-th element computable by pointer arithmetics.
- Deleting an element from the array: O(1) at the end, O(*n*) elsewhere.
  All the elements with greater indices need to be shifted.
- **Extending the array by one element:** O(1) at the end, if there is capacity.
  O(*n*) elsewhere, or if the capacity of the dynamic array is exhausted.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	**x[6]**		
34	7	12	3	4	7	**12**	*free*	*free*

x.length
7

Python lists and the Tutorial 1.1 problem

CO2412

9th November 2021

Python lists

Lists in Python are implemented as dynamic arrays. Their elements behave in the same way as Python variables do in general: **For elementary data types such as numbers, they contain the value**.

```
In [1]:
x = list(range(7))
y = x[2: 4]
y[0] = 7

print("x = ", x)
print ("y = ", y)
```

```
x =  [0, 1, 2, 3, 4, 5, 6]
y =  [7, 3]
```

When a sublist x[i: j] is created from x, **all the sublist elements are copied**.

Python lists

Lists in Python are implemented as dynamic arrays. Their elements behave in the same way as Python variables do in general: For elementary data types such as numbers, they contain the value.

```
In [1]:  x = list(range(7))
         y = x[2: 4]
         y[0] = 7

         print("x = ", x)
         print ("y = ", y)

x =  [0, 1, 2, 3, 4, 5, 6]
y =  [7, 3]
```

This takes O(j – i) time and space; in typical cases, that is O(*n*).

When a sublist x[i: j] is created from x, **all the sublist elements are copied**.

Python lists

Lists in Python are implemented as dynamic arrays. Their elements behave in the same way as Python variables do in general: For elementary data types such as numbers, they contain the value, **otherwise they are object references**.

```
In [1]:
x = list(range(7))
y = x[2: 4]
y[0] = 7

print("x = ", x)
print ("y = ", y)

x =  [0, 1, 2, 3, 4, 5, 6]
y =  [7, 3]
```

```
In [2]:
x = [[i] for i in range(7)]
y = x[2: 4]
y[0] = [7]

print("x = ", x)
print ("y = ", y)

x =  [[0], [1], [2], [3], [4], [5], [6]]
y =  [[7], [3]]
```

When a sublist x[i: j] is created from x, **all the sublist elements are copied**.

Python lists

Lists in Python are implemented as dynamic arrays. Their elements behave in the same way as Python variables do in general: For elementary data types such as numbers, they contain the value, **otherwise they are object references**.

In [1]:
```python
x = list(range(7))
y = x[2: 4]
y[0] = 7

print("x = ", x)
print ("y = ", y)
```

```
x =  [0, 1, 2, 3, 4, 5, 6]
y =  [7, 3]
```

In [3]:
```python
x = [[i] for i in range(7)]
y = x[2: 4]
y[0].pop()
y[0].append(7)

print("x = ", x)
print ("y = ", y)
```

```
x =  [[0], [1], [7], [3], [4], [5], [6]]
y =  [[7], [3]]
```

When a sublist x[i: j] is created from x, **all the sublist elements are copied**.

Revisited: List operations used in selection sort

```
def selection_sort(x):
    for i in range(len(x)):

        min_idx = i
        for j in range(i+1, len(x)):
            if x[j] < x[min_idx]:
                min_idx = j

        next_element = x.pop(min_idx)
        x.insert(i, next_element)
```

[6, 8, 58, 25, 48, 19, 39, 76, 11, 86, 75]

pop(8), which returns 11

[6, 8, 58, 25, 48, 19, 39, 76, 86, 75]

insert(2, 11)

[6, 8, 11, 58, 25, 48, 19, 39, 76, 86, 75]

removes the element at index next_element from the list, and returns its value

the index of all the following elements decreases by 1

inserts next_element at list index i

the index of all the following elements increases by 1

Revisited: List operations used in selection sort

[6, 8, 58, 25, 48, 19, 39, 76, 11, 86, 75]

pop(8), which returns 11

[6, 8, 58, 25, 48, 19, 39, 76, 86, 75]

insert(2, 11)

[6, 8, 11, 58, 25, 48, 19, 39, 76, 86, 75]

```
def selection_sort(x):
    for i in range(len(x)):

        min_idx = i
        for j in range(i+1, len(x)):
            if x[j] < x[min_idx]:
                min_idx = j

        next_element = x.pop(min_idx)
        x.insert(i, next_element)
```

x.pop(min_idx) removes the element at index next_element from the list, and returns its value

the index of all the following elements decreases by 1

x.insert(i, next_element) inserts next_element at list index i

the index of all the following elements increases by 1

These operations both take O(*n*) time except at the very end of the list.

Revisited: Time efficiency of Kadane's algorithm

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0
    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Input size *n* given by len(x)

O(1) instructions

loop executed O(*n*) times

- O(1) instructions
- O(1) optional instructions
- O(1) optional instructions

???

O(*n*) time efficiency

Revisited: Time efficiency of Kadane's algorithm

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0
    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Input size *n* given by len(x)

O(1) instructions

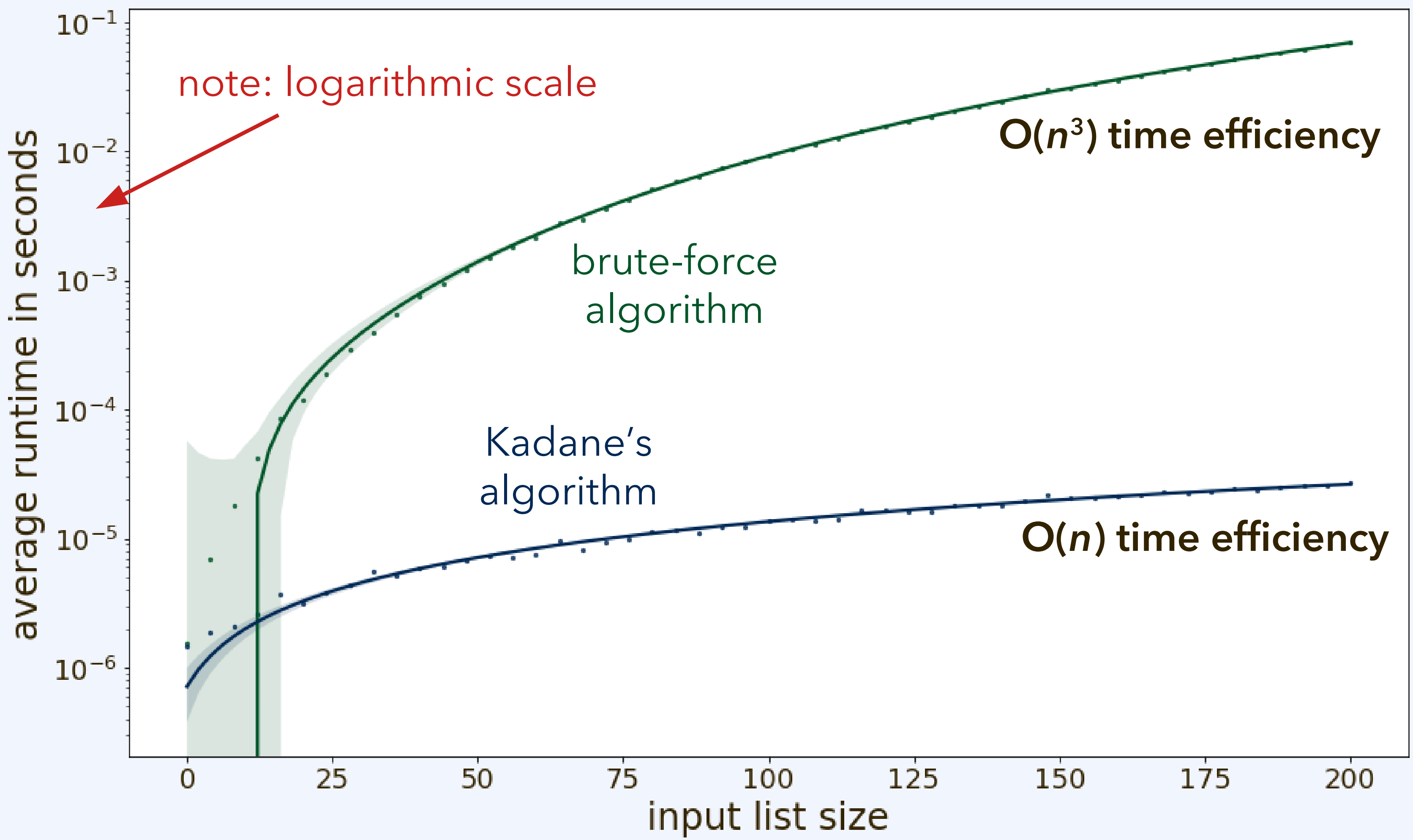
loop executed O(*n*) times

- **O(1) instructions**
- O(1) optional instructions
- O(1) optional instructions

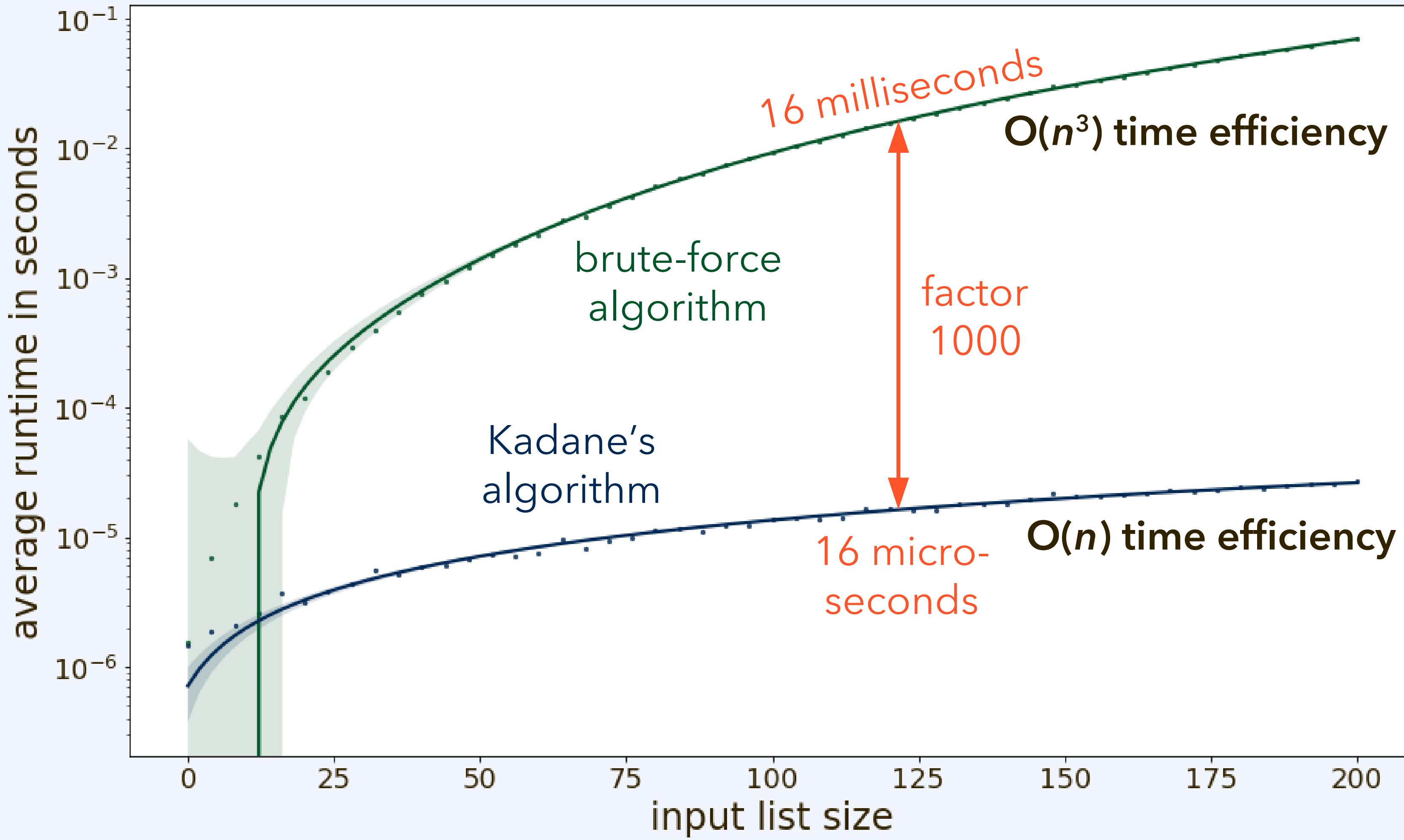
O(*n*) instructions

O(*n*) time efficiency

Maximum sublist sum algorithms: Performance



Maximum sublist sum algorithms: Performance



Space efficiency of Kadane's algorithm *in Python*

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0
    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Input size *n* given by len(x)

Five new variables

One loop index

???

Space efficiency of Kadane's algorithm *in Python*

```
def kadane_sublist(x):
    left_idx, right_idx = 0, 0
    max_sublist_sum = 0
    i = 0
    sublist_sum = 0
    for j in range(len(x)):
        sublist_sum += x[j]
        if sublist_sum < 0:
            i = j+1
            sublist_sum = 0
        elif sublist_sum > max_sublist_sum:
            left_idx, right_idx = i, j+1
            max_sublist_sum = sublist_sum
    return x[left_idx: right_idx]
```

Input size *n* given by len(x)

Five new variables

One loop index

> **Remark**
>
> Using data structures other than Python lists, this might be done in O(1) space.

New list with O(*n*) elements

O(*n*) space efficiency

Tutorial 1.1 problem: Return the maximum

fastest iterative code

```
def max_iterative(listA):
    current_max_val = listA[0]
    for i in listA:
        if i > current_max_val:
            current_max_val = i
    return current_max_val
```

(by Chris Pickup)

fastest recursive code

```
def largestRecur(list, n):

    if n == 1:
        return list[n-1]
    else:
        previous = largestRecur(list, n-1)
        current = list[n-1]
        if previous > current:
            return previous
        else:
            return current
```

(by Sam Hardy)

Tutorial 1.1 problem: Return the maximum

fastest iterative code

```
def max_iterative(listA):
    current_max_val = listA[0]
    for i in listA:
        if i > current_max_val:
            current_max_val = i
    return current_max_val
```

(by Chris Pickup)

fastest recursive code

```
def largestRecur(list, n):

    if n == 1:
        return list[n-1]
    else:
        previous = largestRecur(list, n-1)
        current = list[n-1]
        if previous > current:
            return previous
        else:
            return current
```

(by Sam Hardy)

Both implementations run in O(n) time. The iterative code is more efficient by a factor 7.

Tutorial 1.1 problem: Return the maximum

$O(n^2)$ recursive code

```
def largestRecur(list):
    n = len(list)
    if n == 1:
        return list[n-1]
    else:
        previous = largestRecur(list[0: n-1])
        current = list[n-1]
        if previous > current:
            return previous
        else:
            return current
```

$O(n)$ recursive code

```
def largestRecur(list, n):

    if n == 1:
        return list[n-1]
    else:
        previous = largestRecur(list, n-1)
        current = list[n-1]
        if previous > current:
            return previous
        else:
            return current
```

(by Sam Hardy)

Sublist creation takes $O(n)$ time (and space)!

University of
Central Lancashire
UCLan

CO2412 Computational Thinking

Algorithm design strategies: Overview
Dynamic programming
Static and dynamic arrays
Python lists and the Tutorial 1.1 problem

Where opportunity creates success