Programming in Python

Algorithm complexity, optimization, dynamic programming
lecture 5

Department of Cybernetics and Artificial Intelligence
Technical University of Košice
Ing. Ján Magyar, PhD.

Algorithmization

- describing the solution as a sequence of straightforward steps
- each problem has multiple solutions – how to choose the best one?
- choosing the correct algorithm can make the difference between solving a problem and an unsolvable problem
- most problems can be mapped into a problem for which exists a classic algorithm

Algorithm analysis – number of steps

```
def exp1(a, b):
    ans = 1
    while (b > 0):
        ans *= a
        b -= 1
    return ans
```

Algorithm analysis – asymptotic notation

- defines the upper limit of algorithm complexity as the input gets ever larger
- most often we use the Big-O notation
- we ignore the constant parts of the algorithm, we consider only those parts that are dependent on the size of the input arguments
- $f(x) \in O(n^2)$

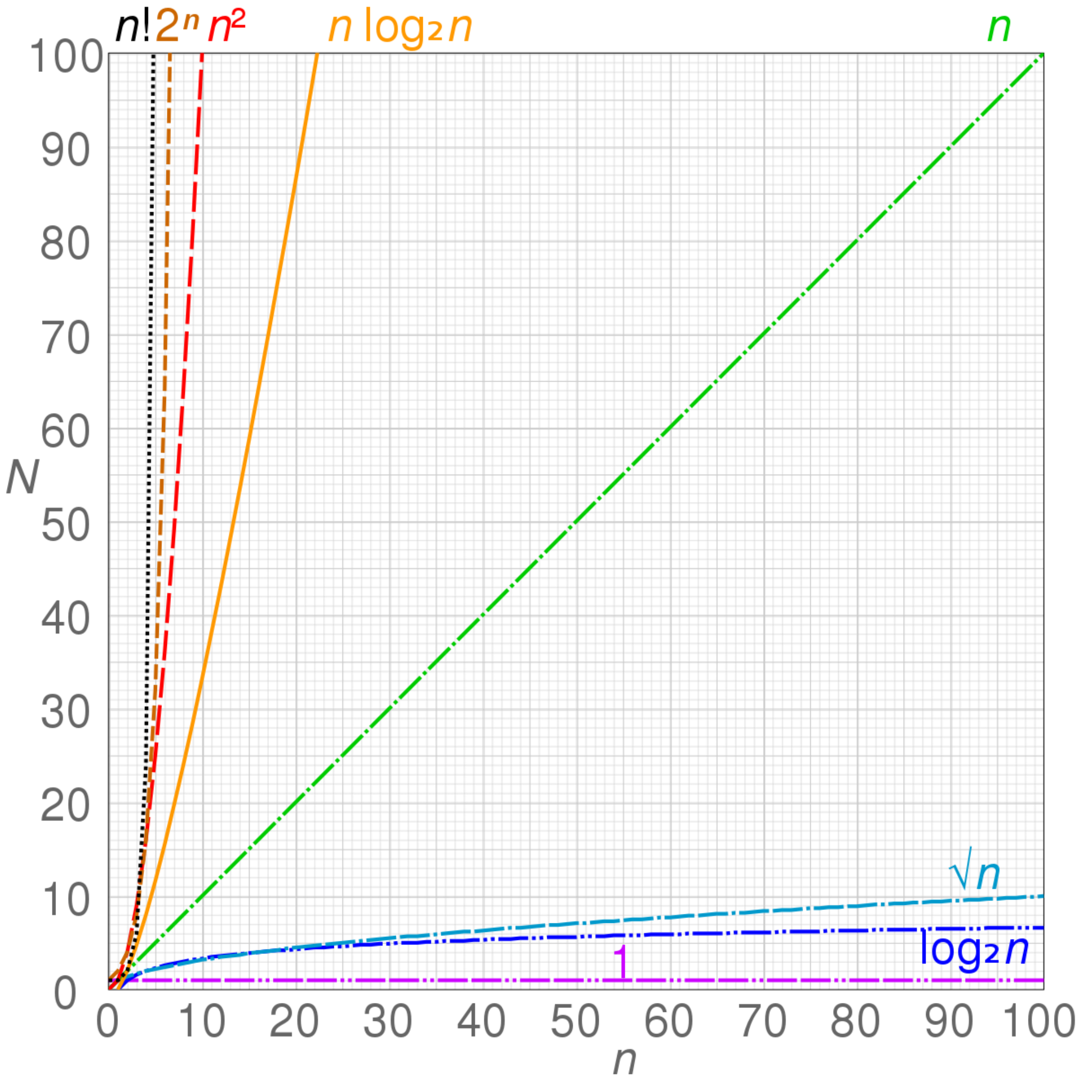
Calculating powers

```
def exp2(a, b):
    if b == 1:
        return a
    if (b % 2) == 0:
        return exp2(a * a, b / 2)
    else:
        return a * exp2(a, b - 1)
```

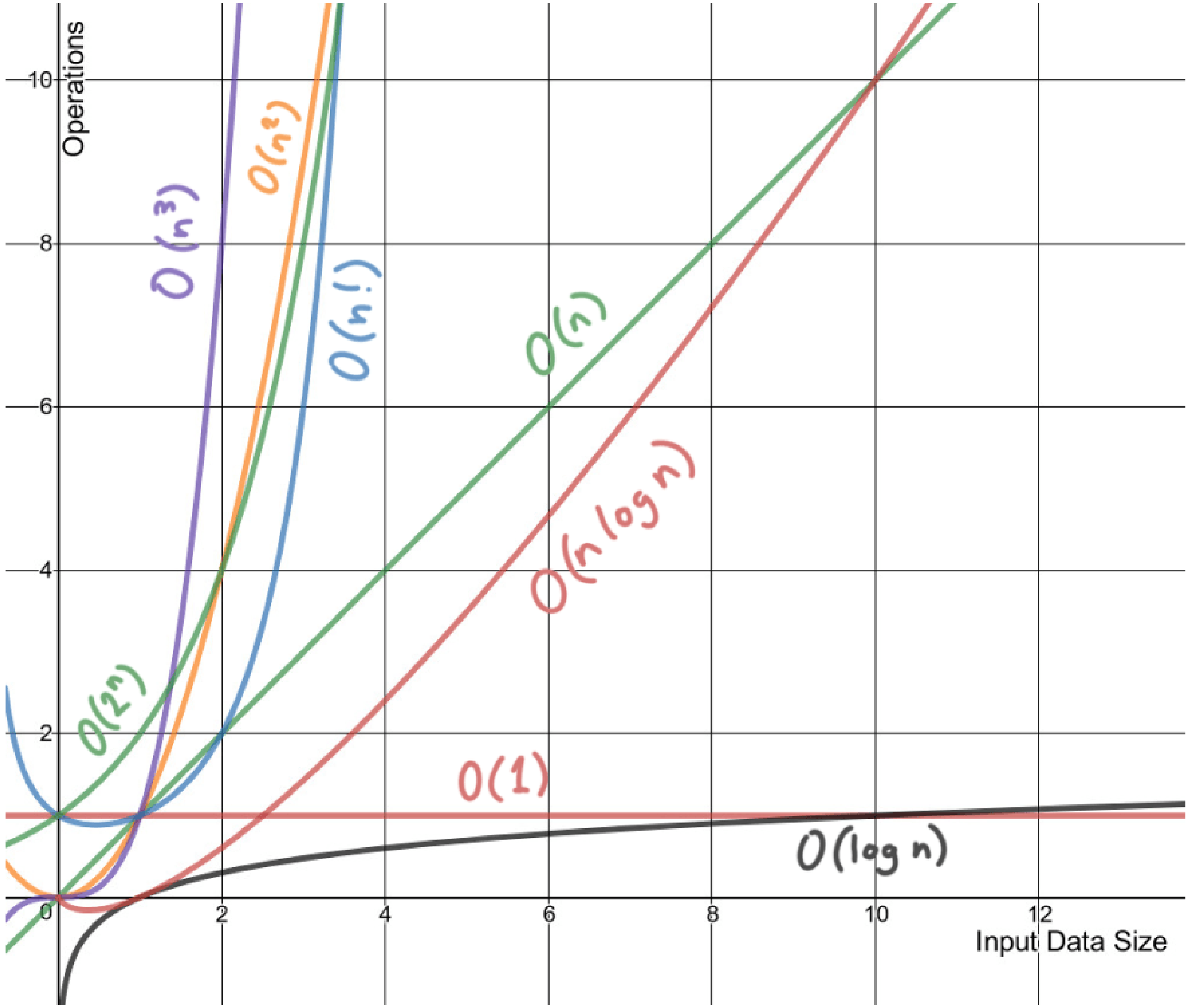
Fibonacci numbers

```
def fib(a):
    if a == 0 or a == 1:
        return 1
    else:
        return fib(a - 1) + fib(a - 2)
```

Algorithm complexity (time)



Algorithm complexity (time)



Algorithm complexity (time)

considering a frequency of 1GHz (one operation per nanosecond)

	n = 1000	n = 1,000,000,000
$O(\log n)$	10 ns	10 ms
$O(n)$	1 µs	1 s
$O(n^2)$	1 ms	16 minutes
$O(2^n)$	10^{284} years	...

Code optimization

- our goal is to increase the performance of an existing program
- part of debugging
- eliminating redundant and repeating operations

Multiple traversing over a list

```
my_list = [1240, -25, 37.24, -12, 0, 35000, 24,
17.23]

for x in range(len(my_list)):
    if my_list[x] < 0:
        my_list[x] = 0

for x in range(len(my_list)):
    my_list[x] *= 1.05

print(my_list)
```

Redundant operations

```
def is_prime(number):
    for x in range(2, number):
        if number % x == 0:
            return False
    return True


is_prime(123475862311)
```

Fibonacci numbers – again

```
def fib(a):
    if a == 0 or a == 1:
        return 1
    else:
        return fib(a - 1) + fib(a - 2)
```

Dynamic programming

- used for optimizing problems with an exponential complexity
- application
  - overarching subproblems - Fibonacci numbers
  - optimal structure - knapsack problem

Fibonacci numbers

```
steps = 0

def fib(a):
    global steps
    steps += 1
    print("Calculating fib for", a)
    if a == 0 or a == 1:
        return 1
    else:
        return fib(a - 1) + fib(a - 2)
```

Fibonacci numbers - simplified

```
def fib_smart(a, memo):
    global num_calls

    num_calls += 1
    print("fib_smart called with", a)
    if a not in memo:
        memo[a] = fib_smart(a - 2, memo) + fib_smart(a - 1, memo)
    return memo[a]

n = 10
memo = {0: 1, 1: 1}
num_calls = 0
fib_smart(n, memo)
```

Memoization

- partial results are stored in a table
- if we haven’t calculated the result for a given input, we calculate it and add it to the table
- when calling the function with the same input, we load the result from the table (table lookup)
- each function call must work with the same table

Theorem of optimal substructure

If the solutions of subproblems are locally optimal, then the overall solution will be globally optimal.

Knapsack problem

- classic problem of optimization
- a burglar has entered an apartment and wants to steal items with the highest possible value
- the burglar has only one knapsack with maximum endurance `W`
- each object in the apartment is described with the pair `(m, w)` where `m` is the object’s value and `w` its weight
- our goal is to find the set of the most valuable items the total weight of which is not larger than `W`

Greedy solution to the knapsack problem

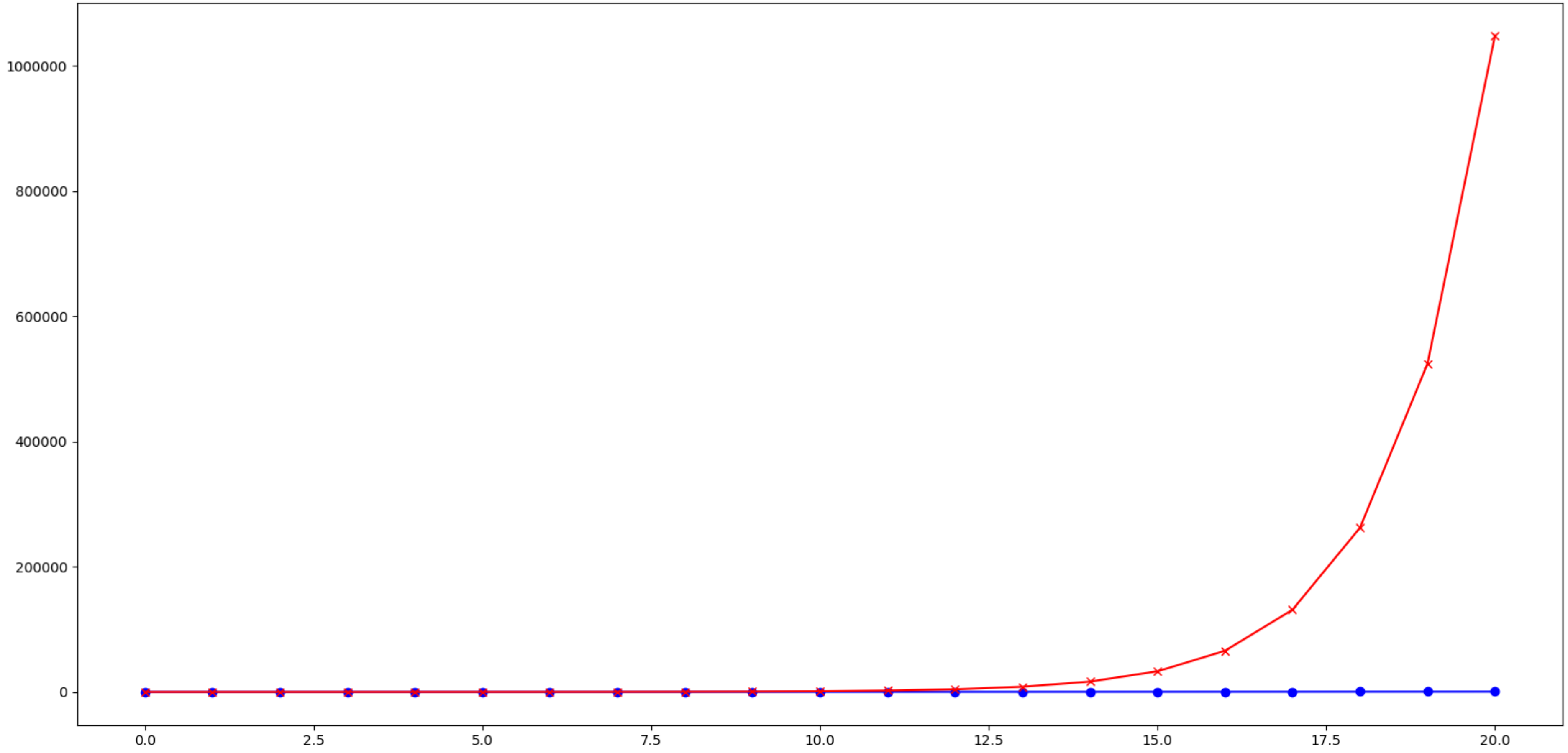
- greedy algorithms look for the best possible immediate solution
- solution: take the most valuable items until your knapsack is full
- we don’t always find the optimal solution, but usually the solution is good enough
- simple implementation, computationally cheap

example:
w = [5, 3, 2]
m = [9, 7, 8]
W = 5

Solving the knapsack problem

- intuitive – brute force
- we test all possibilities
- we represent possible solutions as vectors of n values
- each value will be 0 or 1 (we take/don’t take the item)
- we calculate the total weight for each vector
- how many possibilities are there?

Why don't we use exponential solutions?



Solving the knapsack problem

- suitable representation – decision tree
  - each node is represented as a triplet `(i, w, m)`
  - the left branch contains examples where we don’t take the item with index `i`
  - the right branch contains examples where we take the item with index `i`
  - the optimal solution is the leaf node with the highest possible value of `m`

example:
w = [5, 3, 2]
m = [9, 7, 8]
W = 5

```
def max_val(w, m, i, aW):
    global num_calls
    num_calls += 1
    if i == 0:
        if w[i] <= aW:
            return m[i]
        else:
            return 0
    without_i = max_val(w, m, i - 1, aW)
    if w[i] > aW:
        return without_i
    else:
        with_i = m[i] + max_val(w, m, i - 1, aW - w[i])
    return max(with_i, without_i)
```

```python
def fast_max_val(w, v, i, aW, m):
    global num_calls
    num_calls += 1
    try:
        return m[(i, aW)]
    except KeyError:
        if i == 0:
            if w[i] <= aW:
                m[(i, aW)] = v[i]
                return v[i]
            else:
                m[(i, aW)] = 0
                return 0
        without_i = fast_max_val(w, v, i - 1, aW, m)
        if w[i] > aW:
            m[(i, aW)] = without_i
            return without_i
        else:
            with_i = v[i] + fast_max_val(w, v, i - 1, aW - w[i], m)
        res = max(with_i, without_i)
        m[(i, aW)] = res
        return res
```

Conclusion

- algorithm complexity
- big-O notation, algorithm complexity categories
- code optimization and its goals
- dynamic programming
- memoization
- theorem of optimal substructure
- decision trees in algorithmization problems