Algorithms

Time complexity

Lecture 8 by *Marina Barsky*

Developing Algorithms: steps

1. Formalize the problem: input and output
2. Brainstorm solution
3. Express solution: pseudocode
4. Prove correctness (outside the scope of this course)

- ➢ Estimate running time

1. Estimate space usage

How long does it take to compute?

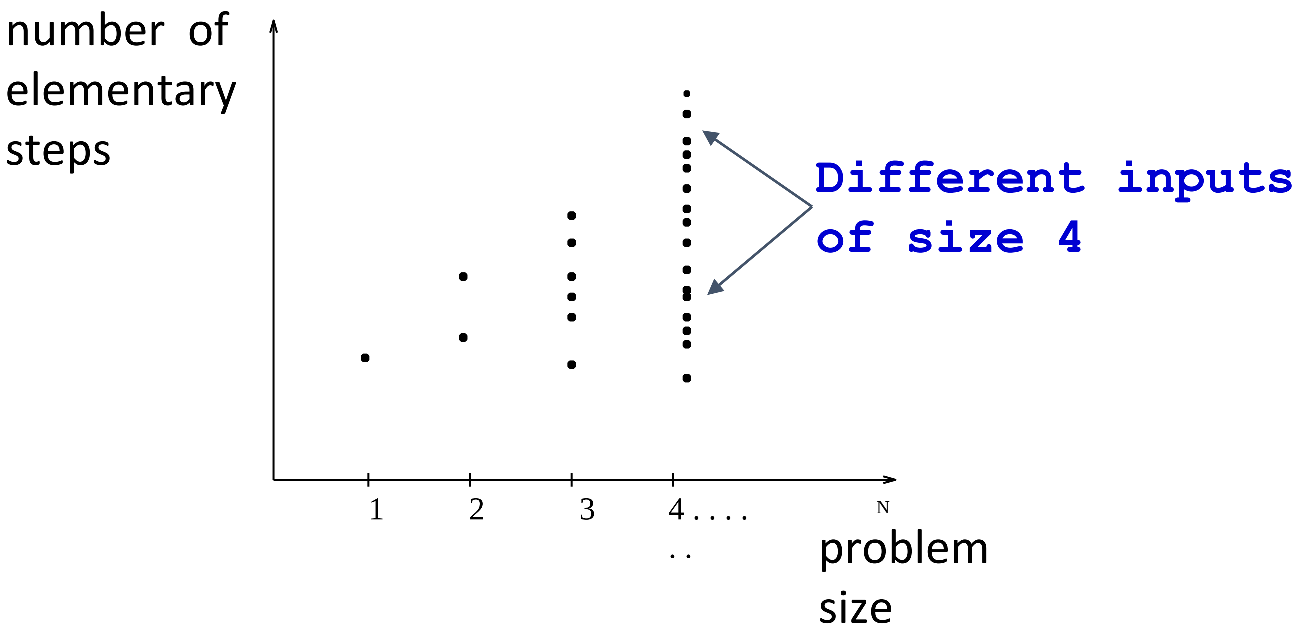
The pseudocode makes it easy to count the total number of steps as it relates to the input size *n* and the nature of the input

```
Algorithm find (array A, target)
n = length of A
for i from 0 to n-1:
      if A[i] == target:
            return i
return -1
```

- It may happen that algorithm finds `target` already on the first iteration: 1 comparison and we are done
- However, it may take *n* comparisons in case that `target` is not in A: *n* operations in total

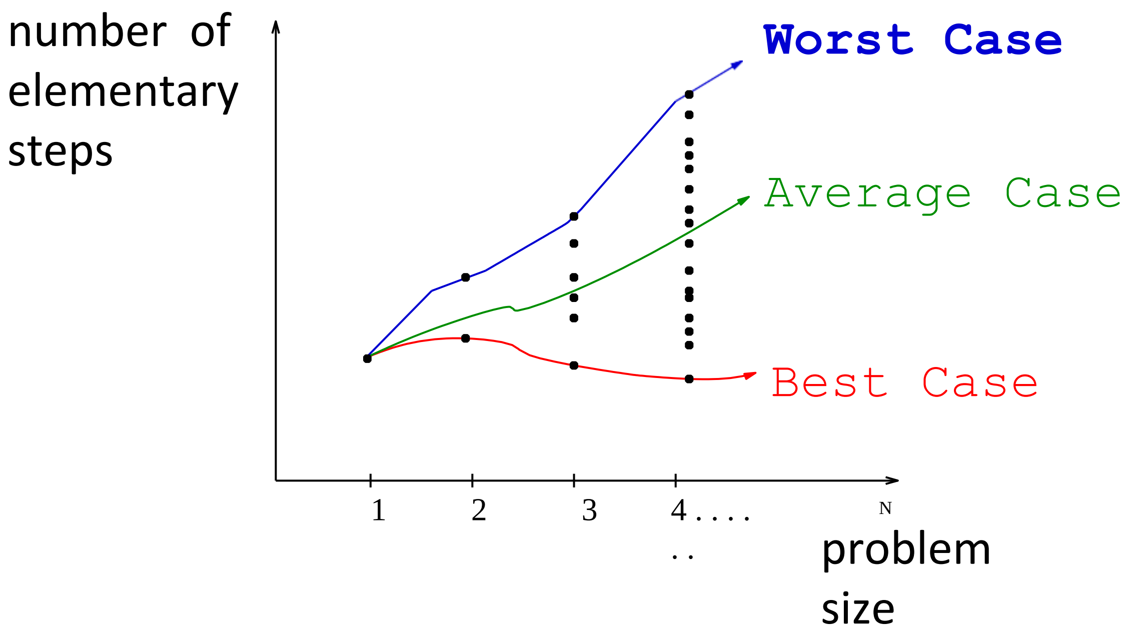
Number of operations vs. input size

- We can count number of steps for a variety of inputs and for different values of *n* and plot the results



Number of steps as function of n

- We want to discover function f(n) from the input size *n* to the total number of steps
- We also see that there is the best case and the worst case for each *n*



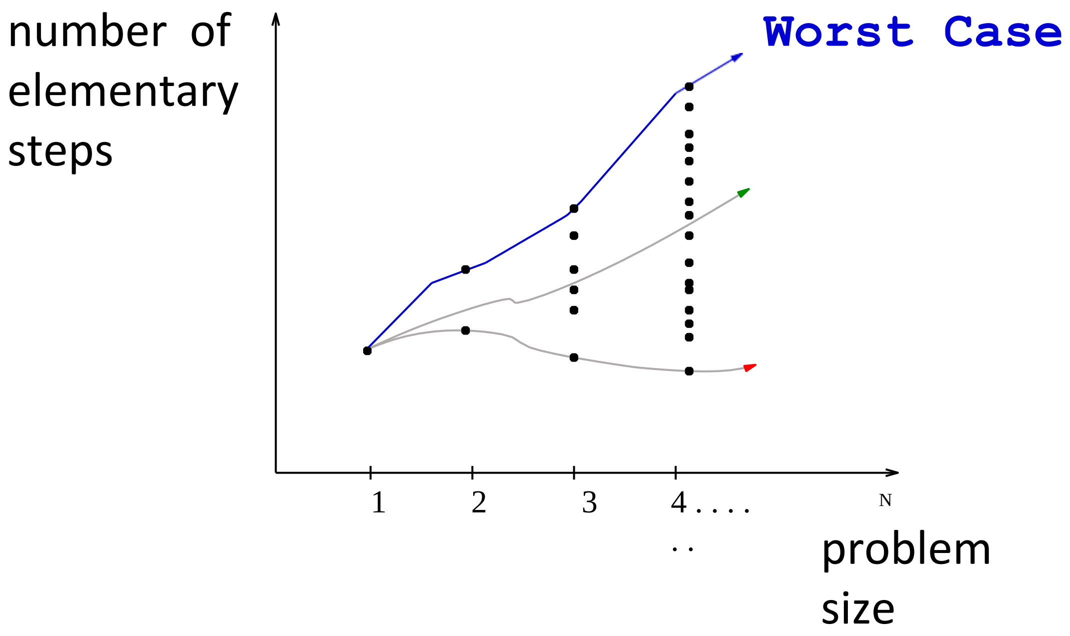
Time complexity

- The best case time complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.
- The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size *n*.
- The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

- Each of these complexities defines a **numerical function**: number of operations vs. size of the input

We are more interested in the worst case

- The nature of the input is generally not known in advance
- We concentrate on the worst-case: we want to know if it is practical to run this algorithm on large inputs of unknown nature



Counting steps: RAM model

The process of counting computer operations is greatly simplified if we accept **the RAM model of computation**:

- Access to each memory element takes a constant time (1 step)
- Each "simple" operation (+, -, =, /, if, call) takes 1 step.
- Loops and function/method calls are *not* simple operations: they depend upon the size of the data and the contents of a subroutine:
    - "sort()" is not a single-step operation
    - "max(list)" is not a single-step operation
    - " if x in list" is not a single-step operation

The RAM model is useful and accurate in the same sense as the **flat-earth model** (which ***is*** useful)!

Loops

The running time of a loop is, at most, the running time of the statements inside the loop (including if tests) multiplied by the total number of iterations.

```
m = 0
for i from 0 to n-1:  #repeat n times:
                         #2 operations –
                         #increment i, test condition
    m = m + 2            #one assignment
```

Total steps = 1 + 2n + n = 3n +1

Nested loops

Analyze from the inside out.

Total running time is the product of the sizes of all the nested loops.

```
for i from 0 to n-1:      # outer loop - 2n times
   for j from 0 to n-1:   # inner loop - 2n times
           k = k+1        # 1 time
```

Total time = $3\,n \times 2\,n = 6n^2$

Consecutive statements

Add the time complexity of each statement.

```
x = x + 1                    # 1
for i from 0 to n-1:         # 2n times
   m = m+2                   # 1 time

for i from 0 to n-1:         # 2n times
   for j from 0 to n-1:      # 2n times
      k = k+1                # 1 time
```

Total time = $1 + 3n + 2n \times 3n = 6n^2 + 3n + 1$

If-then-else statements

Operations: the test, plus either the then part or the else part: whichever is the largest.

```
if len(t) == 0:                               # test: 1
   return false                                  # then part: 1
else:                                         # else part:
   for n from 0 to len(t)-1:                     # loop: 2n
         if t[n] == p[n]:                           # if: 1 (no else)
            return false
   return true                                   # test: 1
```

Total time = 1 + (3 n + 1)= 3n + 2

Let's count! What is the closest to the total number of all steps?

```
count = 0

for i from n/2 to n:
    j = 0

    while  j <= n:
        k = 1

        while k <= n:
            count = count + 1
            k = k + 1

        j = j + 1

return count
```

A. $3n + 4n + 3n$

B. $3n \times 4n \times 3n$

C. $3n/2 + 4n + 3n$

D. $3n \times 2n \times 3n$

Logarithmic complexity

The loop takes a logarithmic number of steps if in each iteration the iteration variable is multiplied by some factor (*i* doubles in this example):

```
i = 1
while i<=n:
   i = i*2
```

- If we observe carefully, the value of *i* is doubling every time
- Initially i = 1, in next step i = 2, and in subsequent steps i = 4, 8 and so on

Logarithmic complexity

```
i = 1
while i<=n:
   i = i*2
```

- Let us assume that the loop is executing some *k* times - before i becomes > n
- At k-th step $\mathbf{2^k = n}$, and at (k + 1)-th step we come out of the loop
- Taking logarithm on both sides:

$\log(2^k) = \log n$

$k \log 2 = \log n$

$\mathbf{k = \log n}$

Logarithmic complexity

The loop takes a logarithmic number of steps if in each iteration it doubles the iteration variable:

```
i = 1
while i<=n:
   i = i*2
```

Total time = 1 + 2 log n

Logarithmic complexity

The same logic holds for the decreasing sequence as well:

```
i = n
while i >= 1:
   i = i/2
```

Total time = 1 + 2 log n