Analysis of Algorithms [Cormen 2.1]

What? Measure efficiency of an algorithm on its use of resources
e.g. time and space

Why? · Design better algorithms
· Compare two algorithms

A first example: Insertion Sort

Sorting problem

A sequence $A[1, n]$ of n elements
Goal: A permutation $A'[1, n]$ of A
such that $A'[i] \leq A'[i+1]$, $\forall i \in [1, n-1]$

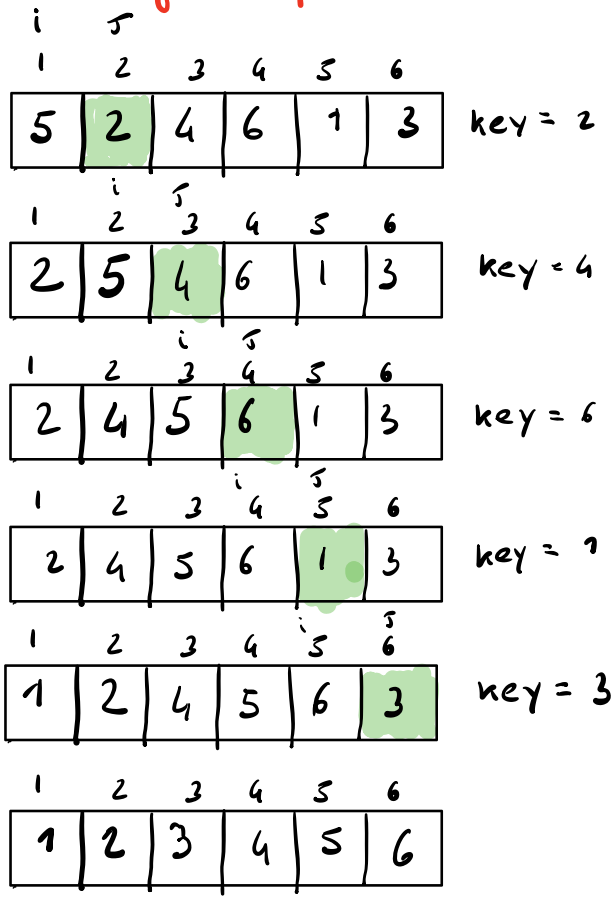
We usually describe algorithms with pseudo code.
Python's implementations in our Lab.

Insertion Sort

Inplace: rearranges elements within A itself without using extra space

```
InsertionSort(A)
    for j = 2 to A.length
        key = A[j]
        // insert key in already sorted A[1, j-1]
        i = j - 1
        while i > 0 and A[i] > key
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Running Example




```
Insertion Sort (A)
1  for j = 2 to A.length
2      key = A[j]
       // insert key in already sorted A[1, j-1]
3      i = j - 1
4      while i > 0 and A[i] > key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Note:

at iteration j

$A[1, j-1]$

is sorted

Visu Algo

visualgo.net/en/sorting

Correctness (and Loop Invariant)

Loop invariant: A property that holds at every iteration

Initialization: Property holds before the loop starts
Maintenance: Property is preserved by loop iterations
Termination: Prepety holds after the loop terminates

IS Loop Invariant

At the beggining of each iteration j, the sub array $A[1, j-1]$ consists of the elements of the original array $A[1, j-1]$ but sorted.

Initialization: When $j = 2$ $A[1, 1]$ is sorted
Maintenance: for loop works by moving $A[j-1]$, $A[j-2]$, ... $A[i]$ one position to the right if $A[j] < A[j-1], A[j-2]$ ... $A[k]$ and places $A[j]$ in position k.
Termination: $j = n+1$, $A[1, \underbrace{j-1}_{n}]$ is sorted.

Similar to proof by induction

next Question: Is the algorithm efficient?

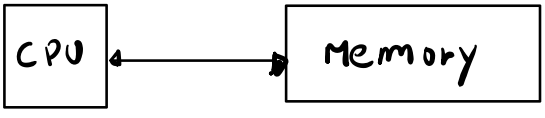
Analysing Algorithms

We need **Model of real computers**

- Simple enough to allow analysis
- Accurate enough to give meaningfull prediction

(there are a lot of different models capturing different aspects of real computers)

RAM model



CPU performs operations between 2 values in memory

Set operations : $+, -, *, \backslash, \sqrt{}, \log$
$\lceil\ \rceil, \lfloor\ \rfloor, \ldots$
AND, NOT, OR, XOR, ...
ie. basic operations only
ie. no sort() op.

Cost of any operation : (1)

- unrealistic : $\sqrt{}$ costs much more than +
  but accurate enough

Goal

Given a input size (e.g. n),
we compute # of operations performed by the algorithm (as a function $f(n)$)

Let's do it for IS

t_j be the number of times while loop is executed at j th iteration

```
Insertion Sort (A)                                  cost    times
  for j = 2 to A.length                             c1      n
      key = A[j]                                    c2      n-1
      // insert key in already sorted A[1, j-1]     0       0
      i = j - 1                                     c3      n-1
      while i > 0 and A[i] > key                    c4      Σ_{j=2}^{n} t_j
          A[i+1] = A[i]                             c5      Σ_{j=2}^{n} (t_j - 1)
          i = i - 1                                 c6      Σ_{j=2}^{n} (t_j - 1)
      A[i+1] = key                                  c7      n-1
```

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

too complicated!

Assumption #1: $c_1 = c_2 = c_3 = \dots = c_7 = c$

$$T(n) = cn + 3c(n-1) + c\sum t_j + 2c\sum (t_j - 1)$$

Best case : $t_j = 1$, $\forall j$

A is already sorted

$$T(n) = cn + 3c(n-1) + cn$$
$$\simeq 5cn$$ LINEAR TIME

Worst case :

A is sorted in decreasing order

$$\forall j, \quad t_j = j$$

$$T(n) = cn + 3c(n-1) + c\sum_{j=2}^{n} j + 2c\sum_{j=2}^{n}(j-1)$$

$$\sum_{j=2}^{n} j = \frac{(n+1)n}{2} - 1$$ GAUSS SUM

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

$$T(n) = cn + 3c(n-1) + c\frac{(n+1)(n)}{2} - 1 + 2c\frac{n(n-1)}{2}$$
$$\simeq 3c\frac{(n+1)n}{2} + 4cn$$

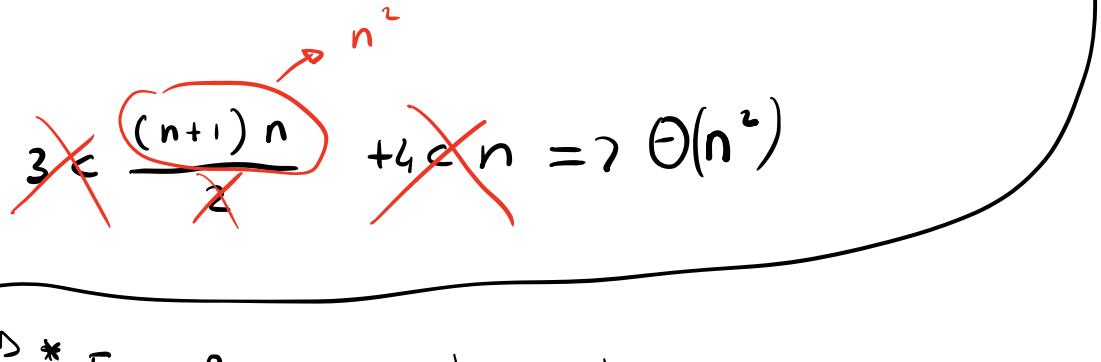
dominant term QUADRATIC TIME

Order of growth

No need to be so precise

Idea: Forget about constant factors and lower order terms!

This is enough (most of the time *) to compare efficient vs inefficient algorithms



* For large enough input size, a $\Theta(n)$ algorithm is better than a $\Theta(n^2)$ algorithm

$$T'(n) = \cancel{1.000.000}\, n \cancel{+10} \rightarrow \Theta(n)$$

$$T''(n) = \frac{n^3}{\cancel{1000}} \cancel{- 100n^2} \cancel{+n} \rightarrow \Theta(n^3)$$

Selection Sort

Idea: At each iteration i, find the minimum in $A[i, n]$. Put this element in position i.

VisuAlgo

inplace

```
Selection Sort (A)
  for i : 1 to A.length                          // ≃ cn
      min_pos = i                                // ≃ cn
      for j : i+1 to A.length                    Σ_{i=1}^{n} c(n-i)
          if A[j] < A[min_pos]
              min_pos = j
      swap (A[i], A[min_pos])                    // ≃ cn
```

cost of i^{th} iteration

$$T(n) = \sum_{i=1}^{n} c(n-i) + 3cn$$

$$\simeq c\sum_{i=1}^{n} i = \frac{c(n+1)\,n}{2} = \Theta(n^2)$$

Important: sequential vs nested parallel loops

```
for i : 1 to n :                    // cn
    s += 1
for j : 1 to n :                    // cn
    s += 1
```

$\Theta(n)$ time

```
for i : 1 to n :
    s += 1
    for j : 1 to n :
        s += 1
```

$\Theta(n^2)$ time

```
for i : 1 to n :
    s += 1
    for j : i to n :
        s += 1
```

$$T(n) = c\sum_{i=1}^{n}(n-i) = c\sum_{i=1}^{n} i = c\frac{(n+1)n}{2} = \Theta(n^2)$$

```
for i : 1 to n :
    s += 1
    for j : 1 to n/4 :
        s += 1
```

$$T(n) = n \cdot \frac{n}{4} = \Theta(n^2)$$

```
for i : 1 to n :
    s += 1
    for j : 1 to 100 :
        s += 1
```

$$T(n) = n \cdot 100 = \Theta(n)$$

From analysis IS and SS are equivalent

but
- IS best case is $\Theta(n)$ time
- SS best case is $\Theta(n^2)$ time

Let's compare those algos on random arrays with Python

Open Jupyter notebook