Analysis of Algorithms [Cormen Sect. 2.1]

What? Measure efficiency of an algorithm on its use of resources
e.g. Time and Space

Why?
- Designing better algorithms
- Compare two algorithms

A first example: Insertion Sort

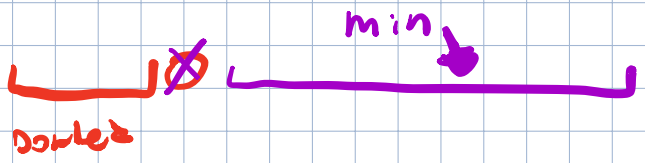
Sorting Problem

A sequence $A[1, n]$ of n elements

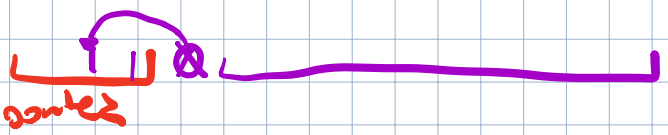
Goal: A permutation $A'[1, n]$ of $A[1, n]$ such that $A'[i] \leq A[i+1]$
$\forall\ i \in [1, n-1]$

We usually describe algorithms with pseudocode. We will implement everything in Python later on.

Given an algo



Selection Sort



Insertion Sort

Insertion Sort

Inplace : rearrange elements within A itself

Stable : Equal elements preserve their relative order

A .... 4 .... 4 .... 4 ....

A ..... 4 4 4 .....

Insertion Sort (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
```

I'm still in the array

not yet found key position

		i	j
1	3	4	2
1	3	—	4
1	—	3	4
1	2	3	4

Running Example : VisuAlgo

visualgo.net/en/sorting

Obviously in place and Stable

Correctness (and Loop invariant)

Loop invariant: It's a property that holds for a loop

Initialization: Property holds before loop starts

Maintainance: Property is preserved by loop iterations

Termination: Property holds after loop terminates (and it helps you to prove correctness)

Loop Invariant

At start of each iteration (say j), the subarray $A[1, j-1]$ consists of the elements of original $A[1, j-1]$ but sorted.

Init: $j = 2$ $A[1, 1]$ is sorted

Maint: "for" loop works by moving $A[j-1]$, $A[j-2]$, ... one position to right. If $A[1, j-2]$ was already sorted, so it is $A[1, j-1]$

Term.: $j = n+1$, $A[1, n]$ is sorted

Similar to proofs by induction.

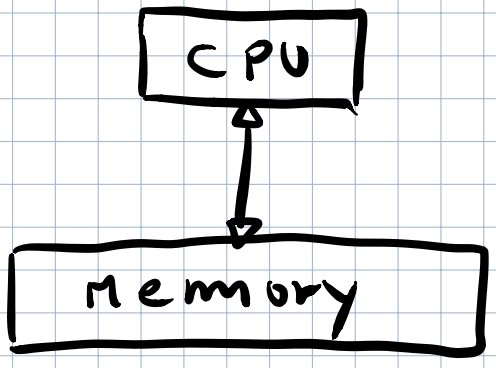
Analyzing Algorithms

Computer Model:

- Simple enough to allow math analysis,
- Precise enough to give accurate analysis

there are several models capturing different aspects of the real machine

RAM model



CPU performs operations on at most 2 values from memory and stores the result back in memory

Set operations: $+, -, *, \backslash, \sqrt{}, \log, \lceil 1, \lfloor \rfloor$...

i.e. basic operations

i.e. not sorted()

Assumption #1

Cost of any operation is (1)*

* irrealistic: $\sqrt{}$ costs much more than $+$ but accurate enough

Goal: given the input size (e.g. n) return # operations performed by the algorithm on a input of that size

Let's do it for Insertion Sort

```
Insertion Sort (A)   |A| = n                          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
    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_j = given the input A, the number of times while loop is executed for iteration j

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

too complicated!

impossible to compare two algorithms

Assumption A_2: $c_1 = c_2 = \dots = c_7 = c$

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

Focus on Best case

A is already sorted

$$t_j = 1 \quad \forall j$$

$$T(n) = cn + 3c(n-1) + c(n-1)$$
$$= 4c(n-1) + cn$$
$$\simeq 5cn = \Theta(n)$$

linear time
(or linear number of operations)

Worst case: any input giving the longest possible running time

we are pessimistic.
we want guarantees: the algorithm never takes more than this.

Worst case for IS: A is sorted in reverse order

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

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

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

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

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

$$\Theta(\,) = \Theta(n^2)$$

we have quadratic time

Order of growth:

Forget about constants and lower order terms.

Example

Express $\frac{n^3}{1000} - 100n^2 - 100n + 3$

in terms of Θ - notation

$\rightarrow \frac{n^3}{1000} - 100n^2 - 100n + 3 \overset{b}{=} \Theta(n^3)$

$\rightarrow 1000n + 100000 = \Theta(n)$