

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

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 $A[i] > key$

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = key$

Running Example

i

j

1 2 3 4 5 6

5	2	4	6	1	3
---	---	---	---	---	---

key = 2

i

j

1 2 3 4 5 6

2	5	4	6	1	3
---	---	---	---	---	---

key = 4

i

j

1 2 3 4 5 6

2	4	5	6	1	3
---	---	---	---	---	---

key = 6

i

j

1 2 3 4 5 6

2	4	5	6	1	3
---	---	---	---	---	---

key = 7

i

j

1 2 3 4 5 6

1	2	4	5	6	3
---	---	---	---	---	---

key = 3

i

j

1 2 3 4 5 6

1	2	3	4	5	6
---	---	---	---	---	---

Insertion Sort (A)

```

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

```

Note :

a + iteration j

$A[1, j-1]$

is sorted

Visual 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: Property holds after the loop terminates

IS Loop Invariant

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

Initialization: When $j=1$ $A[1, 1]$ is sorted

Maintenance: For loop works by moving
 $A[j-1], A[j-2], \dots, A[k]$ one position
to the right if $A[s] < A[j-1], A[j-2], \dots, A[k]$. and places $A[j]$ in
position k .

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

Similar to proof by induction

next Question: Is the algorithm efficient?

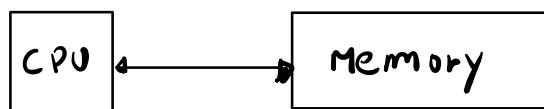
Analyzing Algorithms

We need Model of real computers

- Simple enough to allow analysis
- Accurate enough to give meaningful 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 : +, -, *, \, $\sqrt{}$, log
 \cap , \cup , ...
 AND, NOT, OR, XOR, ...
 i.e. basic operations only
 i.e. 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 $p(n)$)

Let's do it for IS

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

Insertion Sort (A)

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 $A[i] > \text{key}$

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = \text{key}$

cost	times
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	
C_7	$n-1$

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4 \cdot \sum_{j=2}^n t_j \\ + C_5 \sum_{j=2}^n (t_j - 1) + C_6 \sum_{j=2}^n (t_j - 1) + C_7(n-1)$$

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

$$T(n) = cn + 3c(n-1) + cn \\ \approx 5cn \quad \text{LINEAR TIME}$$

Worst case :

A is sorted in decreasing order

$$\forall j, t_j = j$$

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

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

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

$$T(n) = cn + 3c(n-1) + c \frac{(n+1)(n)}{2} - 1 \\ + 2c \frac{n(n-1)}{2} \\ \approx \cancel{3c \frac{(n+1)n}{2}} + 4cn \quad \text{QUADRATIC TIME}$$

dominant term

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

$$\cancel{3} \cancel{\frac{(n+1)n}{2}} + \cancel{4n} \Rightarrow \Theta(n^2)$$

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

$$T(n) = \cancel{1,000,000} n + \cancel{10} \rightarrow \Theta(n)$$

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

Selection Sort

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

Visualgo

Put this element in position i .

Inplace

Selection Sort (A)

for $i : 7$ to $A.length$

$\min_pos = i$

$\approx cn$

$\approx cn$

for $j : i+1$ to $A.length$

$\text{cost } \tilde{c}$ $\left(\begin{array}{l} \text{if } A[j] < A[\min_pos] \\ \min_pos = j \end{array} \right) \sum_{i=1}^n c(n-i)$

swap ($A[i], A[\min_pos]$) $\approx cn$

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

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

Important: sequential vs nested parallel loops

for $i : 1 \text{ to } n :$
 $\Delta += 1$ //cn

for $j : 1 \text{ to } n :$
 $\Delta += 1$ //cn

$\Theta(n)$ time

for $i : 1 \text{ to } n :$
 $\Delta += 1$

for $j : 1 \text{ to } n :$
 $\Delta += 1$

$\Theta(n^2)$ time

for $i : 1 \text{ to } n :$
 $\Delta += 1$

for $j : \underline{i} \text{ to } n :$
 $\Delta += 1$

for $i : 1 \text{ to } n :$
 $\Delta += 1$

for $j : 1 \text{ to } \frac{n}{4} :$
 $\Delta += 1$

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

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

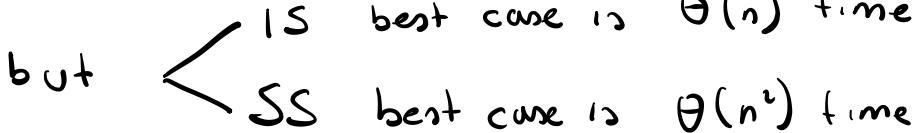
for $i : 1 \text{ to } n :$

$\Delta += 1$

for $j : 1 \text{ to } 100 :$
 $\Delta += 1$

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

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