Correctness and efficiency

Serge Kruk

September 6, 2022

In this class (most of the time)

- We do not care about the language.
- We do not care about micro-optimization.
  - Which is why the book uses pseudo-code and I use python and Lisp.
- We will quantify efficiency for large instances only.
  - Our aim is to compare algorithms.
  - We focus on the worst-case behaviour.
  - Only if two algorithms are asymptotically identical might we go in more details.

The four questions permeating this class (Never forget this!)

Given a problem:

- Design (or find) an algorithm.
- Prove that it is correct (and solves the problem).
- Discuss its efficiency (contrast with other algorithms).
- It is the best that can be done? (A very hard question in general).

Recall the algorithm (Discuss C and E)

- Problem: Given an array of integers, sort them in increasing order.
- Proposed solution:

```
def bsn(a=[]):
    n = len(a)
    for i in range(n-1):
        for j in range(i+1,n):
            if a[i] > a[j]:
                a[i],a[j] = a[j],a[i]
    return a
```

Correctness

- Note that the algorithm will terminate.

Correctness

- Note that the algorithm will terminate.
- Loop invariant: Entering the for loop on `i` with value `k`, array a[0..k-1] contains the k smallest elements of the set in sorted order.

Correctness

- Note that the algorithm will terminate.
- Loop invariant: Entering the for loop on `i` with value `k`, array a[0..k-1] contains the k smallest elements of the set in sorted order.
  - Vacuously true the first time.

Correctness

- Note that the algorithm will terminate.
- Loop invariant: Entering the for loop on `i` with value `k`, array a[0..k-1] contains the k smallest elements of the set in sorted order.
  - Vacuously true the first time.
  - Assume true for some value `k`

Correctness

- Note that the algorithm will terminate.
- Loop invariant: Entering the for loop on `i` with value `k`, array a[0..k-1] contains the k smallest elements of the set in sorted order.
  - Vacuously true the first time.
  - Assume true for some value `k`
    - The second loop will compare to `a[k]` every element from `a[k+1]` to the `a[n-1]` and move the smallest one of those to `a[k]`. Therefore at the end `a[k]` is the smallest element of `a[k..n-1]`. Therefore exiting the `i` loop `a[0..k]` contains the `k+1` smallest elements in sorted order.

Correctness

- Note that the algorithm will terminate.
- Loop invariant: Entering the for loop on `i` with value `k`, array a[0..k-1] contains the k smallest elements of the set in sorted order.
  - Vacuously true the first time.
  - Assume true for some value `k`
    - The second loop will compare to `a[k]` every element from `a[k+1]` to the `a[n-1]` and move the smallest one of those to `a[k]`. Therefore at the end `a[k]` is the smallest element of `a[k..n-1]`. Therefore exiting the `i` loop `a[0..k]` contains the `k+1` smallest elements in sorted order.
  - Since the first loop ends with `i` entering at value `n-2`, it exits with `a[0..n-1]` containing the `n` elements (the whole array) sorted. □

Correctness elements to consider

- Termination is essential but often trivial to ascertain.

Correctness elements to consider

- Termination is essential but often trivial to ascertain.
- Finding the good loop invariant is crucial for looping algorithms.

Correctness elements to consider

- Termination is essential but often trivial to ascertain.
- Finding the good loop invariant is crucial for looping algorithms.
  - If you design the algo, then the LI is the idea you have, made formal.

Correctness elements to consider

- Termination is essential but often trivial to ascertain.
- Finding the good loop invariant is crucial for looping algorithms.
  - If you design the algo, then the LI is the idea you have, made formal.
  - If you are given the algo, then you have to read carefully and understand the code. (Reading code regularly is essential!)

Efficiency

- Which line(s) of code are expensive/representative of work done?
  - The function call?
  - The `len` call?
  - The for loops?
  - The `if` statement?
  - The `swap` statement?

Computing runtime the hard way

- We count comparisons.
- We could count swaps, and only discuss worst case.

i	j	Comp
0	1,2,3,4,. . . ,n-1	n-1
1	2,3,4,5,. . . ,n-1	n-2
2	3,4,5,6,. . . ,n-1	n-3
. . .		
n-3	n-2,n-1	2
n-2	n-1	1
	Total:	?

Adding up the last column:

$$1 + 2 + 3 + \ldots + n - 1 = \frac{(n-1)n}{2}$$

Interlude: Summation notation

$$\sum_{i=1}^{i=5} 1 = 1 + 1 + 1 + 1 + 1 = 5$$

$$\sum_{i=1}^{i=5} i = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{i=1}^{i=n} i = 1 + 2 + 3 + 4 + 5 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=a}^{i=b} i = a + (a+1) + (a+2) + \ldots + (b-1) + b = \frac{b(b+1)}{2} - \frac{(a-1)a}{2}$$

Computing runtime the easy way

```
def bs(a=[]):
    n = len(a)
    for i in range(n-1):
        for j in range(i+1,n):
            if a[i] > a[j]:
                a[i],a[j] = a[j],a[i]
    return a
```

Let $T(n)$ be the runtime of `bs` when the input is an array of length n.

$$T(n) = \sum_{i=0}^{i=n-2} \sum_{j=i+1}^{n-1} 1$$

Explain each number in the above.

Almost mechanically

The runtime of algorithm `bs`

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{i=n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{i=n-2} (n-1) - (i+1) + 1 \\
&= \sum_{i=0}^{i=n-2} n - i - 1 \\
&= \sum_{i=0}^{i=n-2} n - 1 - \sum_{i=0}^{i=n-2} i \\
&= (n-1)(n-2+1) - (n-2)(n-1)/2 \\
&= \frac{(n-1)n}{2}
\end{aligned}
$$

Summarized claim of efficiency

We say that algorithm `bs`, for an array of size n,

- has a running time proportional to $(n-1)n/2$.
- has a space requirement proportional to n.

These are powerful statements, independent of CPU and of language used (mostly).

Another example of runtime analysis

```
def sq_no_mult_i(n):
  n2 = 0
  for i in range(n):
    n2 += n
  return n2
```

Let us define $T(n)$ for the number of addition done by the algorithm for an input integer n.

$$\begin{aligned} T(n) &= \sum_{i=0}^{i=n-1} 1 \\ &= (n-1)+1 \\ &= n \end{aligned}$$

Let us do this for recursive code

```
def sq_no_mult_r(n, count=-1, nsquared=0):
  if count == -1:
    count = n
  if count == 0:
    return nsquared
  return sq_no_mult_r(n, count-1, nsquared+n)
```

Let us define $T(count)$ as the runtime of the algorithm with input $count = n$. Then if we count additions

$$T(0) = 0$$
$$T(count) = 1 + T(count - 1)$$

Almost mechanically

The recursion is then solved.

$$
\begin{aligned}
T(n) &= 1 + T(n-1) \\
&= 1 + 1 + T(n-2) \\
&= 1 + 1 + 1 + T(n-3) \\
&= 1 + 1 + 1 + 1 + T(n-4) \\
&= \vdots \\
&= 1 + 1 + 1 + \ldots + 1 + T(0) \\
&= n
\end{aligned}
$$

which corresponds to our iterative algorithm runtime.

Summary of runtime analysis

Recipe

- Decide carefully what you are going to count.
- Define T(some parameter(s)) as the runtime given input.
- For iterative algorithm, write down your sums and simplify.
- For recursive algorithm, write down your recursion and simplify.

Interlude: Can you prove that

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

Proof (direct)

$$S_k = \sum_{i=1}^{i=k} i = k(k+1)/2$$

$$\begin{aligned}
S_k &= 1 + 2 + 3 + \ldots + k - 1 + k \\
2S_k &= 1 + 2 + 3 + \ldots + k - 1 + k + k + k - 1 + k - 2 + \ldots + 3 + 2 + 1 \\
&= 1 + 2 + 3 + \ldots + k - 1 + k \\
&+ k + k - 1 + k - 2 + \ldots + 3 + 2 + 1 \\
&= (k+1) + (k+1) + (k+1) + \ldots + (k+1) + (k+1) \\
&= k(k+1) \\
S_k &= \frac{k(k+1)}{2}
\end{aligned}$$

Proof (by induction)

- Base case

$$S_1 = 1 \cdot 2/2 = 1$$

- Inductive step. Assume true for S_{k-1} and consider S_k

$$\begin{aligned} S_k &= 1 + 2 + 3 + \ldots k - 1 + k \\ &= S_{k-1} + k \\ &= \frac{(k-1)k}{2} + k \\ &= k(\frac{k-1}{2} + 1) \\ &= k\frac{(k+1)}{2} \\ &= \frac{k(k+1)}{2} \quad \square \end{aligned}$$

Discuss

Do you notice any similarity between the proof of correctness of `bs` and of the triangular numbers?

Notation (informal introduction)

Most of the time, we will summarize

$$\frac{n(n+1)}{2}$$

as

$$\Theta(n^2)$$

Two reasons:

- The term n^2 is the leading term of $n(n+1)$.
- We only care about large n.

Efficiency (take two)

Discuss the efficiency of the following:

```
def bs0 (a):
    s,n = True, len(a)
    while s:
        s = False
        for i in range(n-1,0,-1):
            if a[i] < a[i-1]:
                a[i],a[i-1] = a[i-1],a[i]
                s = True
    return a
```

Efficiency

In the algorithm above, given an array of size n,

- The worst case is proportional to n^2
- The best case is proportional to n

The best case occurs when the array is already sorted. The worst case occurs when there is at least one swap per inner loop.
How often do you think the best case happens in practice?

Efficiency

- Worst case
- Best case
- Average case (very hard in general)

Algorithm is correct

Consider the outer loop. It will loop until no swaps are done. No swaps indicates all elements in place. Therefore, the array is sorted. So if the algorithm finishes, it produces a sorted array.
The number of adjacent elements at the right of a[k] that are smaller than a[k] is either zero or is decreased at every inner loop. Therefore, the algorithm must terminate.

Review (Discuss)

What are the differences between an array and a linked list?

Interlude: Array vs Linked list

- Array uses less storage
- Random access in an array is faster $\Theta(1)$ vs $\Theta(n)$
- Arrays are more costly to grow (impossible in some languages)

Discuss the algorithms we have seen (bs and bsO) wrt the following problems

- Given an array of integers, sort the integers in increasing order.

Discuss the algorithms we have seen (bs and bsO) wrt the following problems

- Given an array of integers, sort the integers in increasing order.
- Given a linked list of integers, sort the integers in increasing order.

A pitch for Lisp

This code will handle arrays as well as linked lists. No change.

```
(defun bs (sequence &optional (compare #'<))
  (loop with sorted = nil until sorted do
    (setf sorted t)
    (loop for a below (1- (length sequence)) do
        (unless (funcall compare (elt sequence a)
                                 (elt sequence (1+ a)))
          (rotatef (elt sequence a)
                   (elt sequence (1+ a)))
          (setf sorted nil)))))
```

Class problem

Develop the following

The integer root function: iroot(k, n) returns the largest integer x such that x^k does not exceed n, assuming k and n are both positive integers. (Of course, you must remain in the integer realm all the time; you are not allowed to simply call a floating point library and do `floor(sqrt(k,n))`.) For example:

- iroot(3, 125) equals 5, because 5^3 equals 125

As usual, code, argument of correctness and runtime.

Class problem

Develop the following

The integer root function: iroot(k, n) returns the largest integer x such that x^k does not exceed n, assuming k and n are both positive integers. (Of course, you must remain in the integer realm all the time; you are not allowed to simply call a floating point library and do `floor(sqrt(k,n))`.) For example:

- iroot(3, 125) equals 5, because 5^3 equals 125
- iroot(3, 126) equals 5 also

As usual, code, argument of correctness and runtime.

Class problem

Develop the following

The integer root function: iroot(k, n) returns the largest integer x such that x^k does not exceed n, assuming k and n are both positive integers. (Of course, you must remain in the integer realm all the time; you are not allowed to simply call a floating point library and do `floor(sqrt(k,n))`.) For example:

- iroot(3, 125) equals 5, because 5^3 equals 125
- iroot(3, 126) equals 5 also
- iroot(3, 124) is 4, because 5^3 is greater than 124.

As usual, code, argument of correctness and runtime.