algorithm analysis

Proving correctness (revisited)
Running time
Test cases: InsertionSort, SelectionSort

correctness

Use a **loop invariant** (instance) of the algorithm to show the correctness:

Insertion-sort: in each iteration (for a given j), the subset A[1,…,j-1] consists of numbers originally in A[1,…,j-1], but sorted in accending order

We must show three factors:

Initialization: it is true prior to the first iteration

Maintenance: if it is true before an iteration, it remains true before next iteration

Termination: when loop terminates, we get a property that holds

insertion sort

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence 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
```

Initialization: *it is true prior to the first iteration*

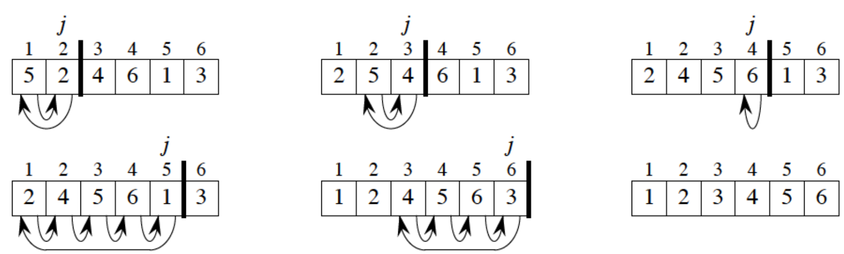
Just before the first iteration, j=2 => A[1,…,j-1] is a single element array A[1], which is sorted

Maintenance: *if it is true before an iteration, it remains true before next iteration*

We should also prove a loop invariant for the *while* loop. In such simple cases, it is sufficient to describe what it does, i.e., moving elements by one position to right until proper position for key is found.

Termination: *when loop terminates, we get a property that holds*

The outer loop ends when j > n, i.e., when j=n+1 => j-1 = n. Replacing j-1 with n in the loop invariant, the subarray A[1,…n] consists of all elements originally in A[1,…,n] but in sorted order.



proving correctness

- A counter-example is enough to prove incorrectness
- We need to prove that for all valid instances,
  - The algorithm terminates
  - Returns the desired output

Keep in mind: most of the time, the number of valid input instances can be almost infinite => exhaustive testing of correctness is impossible

- Use a sequence of logical arguments, similar to proving a theorem

Analyzing algorithms

- We want to predict the resources an algorithm requires, usually the running time
- To predict resource requirements, we need a computational model
- For all algorithms, we assume a generic RAM model of computation on a single processor
- Processor has a limited instruction-set and the program statements run <u>sequentially</u>; no concurrent operations

Analyzing algorithms

- We assume two atomic data types, integer and real, are supported and operations on such data takes unit amount of time
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling, shift left/right
  - Data movement: load, store, copy
  - Control: subroutine call/return
- We compute efficiency w.r.t the size of its input, where size is measured using binary encoding
- Two kinds of efficiency analysis: $T(n)$ and $S(n)$ are functions denoting the <u>time</u> and <u>space</u> required for running the algorithm on an n-size object

Analyzing algorithms

- Based on the input, an algorithm’s performance may vary
  - While sorting an array, if the array is almost sorted, it may be less costly (best case)
  - While searching for an element in an array, the desired object may not be present in the array (worst case)
  - Sorting 1000 numbers takes longer than sorting 3 numbers
- **best-case analysis:** the fastest the algorithm will run for a valid input
- **worst-case analysis:** the slowest the algorithm will run for a valid input
- **average-case analysis:** running time for an “average” (expected) valid input; all inputs are equally likely => estimate the expectation of running time T(n) over all possible inputs.

running time

- On a particular input, it is the number of <u>primitive operations</u> executed

- Want to define steps that are machine-independent

- A line of pseudocode may take a different amount of time than another, but each execution of the same line takes the same amount of time

- Each line consists of primitive operations

- If the line is a subroutine call, the actual call takes constant time, independently from what happens within the subroutine

- If the line specifies operations other than primitive ones, then it may take more than constant time: “sort the points w.r.t to x property”

running time calculation

General rules

- ***for* loop** at most the running time of the loop body times the number of iterations

- **nested loop** the running time of the loop body multiplied by the product of the sizes of all the loops

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)            □ n^2 times a constant cost
        k++
```

- **consecutive statement** adding the running times

```
for (i=0; i<n; i++)
    <statement>                    □ n times a constant cost
for (i=0; i<n; i++)
    for (j=0; j<n; j++)            □ n^2 times a constant cost
        <statement>
```

- ***if/else*** at most the running time of test plus the highest of the running times of the two branches

```
 if <condition>                    □ some cost
    <statement 1>
else                               □ max cost {<statement 1>, <statement 2>}
    <statement 2>
```

Example: insertion sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\qquad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

- number of "tests" in j-th iteration depends on j/A[j]
- it executes one time more than the while-loop body

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

- For j = 2, 3, …n, let t_j be the number of times that the while loop test is executed for the given j

- When a **for** or **while** loop, the test in the loop header is performed one time more than the loop body

Example: insertion sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad$ *key* $= A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$	0	$n-1$
$\quad$ $i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad$ $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad$ $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad$ $A[i+1] =$ *key*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- **Best-case:** the array is already sorted

Always find that A[i] ≤ key => all $t_j = 1$ =>

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- **Worst-case:** the array is sorted in reverse order

  Always find that A[i] > key in **while** loop test

  Have to compare key with all elements to the left of the j position (j-1 comparisons)

  while loop header becomes i=0 (one additional test) => $t_j = j$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j \text{ and } \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$$

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

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

- Let k = j – 1:

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

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n - 1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- …and it all this translates to:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

and therefore $T(n) = an^2 + bn + c$ => quadratic function of n

worst & average case analysis

- We usually focus on finding the worst-case running time for any input of size n

- This gives us a guaranteed upper bound on the running time

- There are algorithms where the worst case occurs frequently; e.g. when searching for an item that is not present.

- **Why not analyze the average case?**

worst & average case analysis

- Why not analyze the average case?

  Consider the insertion sort: on average, the key A[j] is less than half of the elements in A[1…j-1], and greater than the other half =>

  on average the while loop has to look halfway through the sorted subarray A[1…j-1] to decide where to place the key =>

$$t_j = j/2 \Rightarrow$$

  although average case running time is half of the worst case running time, it is still a quadratic function of n

order of *growth*

- Look only at the leading term of the running time formula
  =>asymptotic behavior

  For instance, insertion sort: $T(n) = an^2 + bn + c$ =>
  it grows like n^2 but it is not equal to n^2

  One algorithm is more efficient than another if its worst case running time has a smaller order of *growth*.

example

Consider sorting n numbers in ascending order, stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A and exchange it with A[2]. Continue in the same fashion for the first n-1 elements of A.

- Write a pseudocode for the algorithm (selection sort).
- What loop invariant does this algorithm maintain?
- Why does it need to run for only the first n-1 elements rather than all n elements?
- Give best case and worst case running times

j = 1

$\boxed{1}$ 5 6 3 7 2 smallest = 1

j = 2

1 $\boxed{5}$ 6 3 7 2 smallest = 2
smallest = 4
smallest = 6

j = 3

1 2 $\boxed{6}$ 3 7 5 smallest = 3
smallest = 4

j = 4

1 2 3 $\boxed{6}$ 7 5 smallest = 4
smallest = 6

j = 5

1 2 3 5 $\boxed{7}$ 6 smallest = 5
smallest = 6

1 2 3 5 6 7

SELECTION-SORT(A)

```
n = A.length
for j = 1 to n - 1
    smallest = j
    for i = j + 1 to n
        if A[i] < A[smallest]
            smallest = i
    exchange A[j] with A[smallest]
```

SELECTION-SORT(A)

for $j = 1$ **to** $n - 1$
    $smallest = j$
    for $i = j + 1$ **to** n
        if $A[i] < A[smallest]$
            $smallest = i$
    exchange $A[j]$ with $A[smallest]$

- The algorithm maintains the **loop invariant** that at the start of each iteration of the outer *for* loop, the subarray A[1…j-1] consists of the j-1 smallest elements in the arrays A[1…n], and this subarray is sorted.

- After the first n-1 elements, the subarray A[1…n-1] contains the smallest n-1 elements sorted, and therefore A[n] must be the largest element

SELECTION-SORT(A)

$n = A.length$
for $j = 1$ **to** $n - 1$
    $smallest = j$
    for $i = j + 1$ **to** n
        if $A[i] < A[smallest]$
            $smallest = i$
    exchange $A[j]$ with $A[smallest]$

c_1 n

$c2$ $n\text{-}1$

$c3$ $\sum_1^{n-1}\sum_{j+1}^{n} 1 = n^2/2+n/2$

$c4$ $\sum_1^{n-1}\sum_{j+1}^{n-1} t_j = t_j(n^2/2\text{-}n/2)$

...

- Best case: $t_j = 0$
- Worst case: $t_j = 1$

In both cases it is n^2 order of *growth*

assignment

a. Show that in both best and worst cases, the order of growth is quadratic (in a similar way as we did for insertion sort). Can you explain why (in your own words)?

b. Write a program in C++ that takes as input an array of integers of length N, and outputs the array sorted in ascending order, performing selection sort.
   - N should be user-defined (define a max value allowed)
   - `SelectionSort()` should be a routine called in `main()`

Submit:

- report with run time analysis
- C++ code