Analyzing Algorithms

Prepared by Hyrum D. Carroll
(based on slides from Suk Jin Lee)

Basic principles

- Analyzing an algorithm means ability to predict resources that the algorithm requires.
- *Computational (running) time* is the most important factor that we want to measure.
- To evaluate resource requirements and to predict running time, we need a universal and independent computational model: computational time is a universal measure and it should not depend on a particular computer.

Random-access machine model

- *Random-access machine* (RAM) is a virtual computer with the following properties:
  - Instructions are executed one after another, with no concurrent operations.
  - The instruction's set coincides with the commonly found one in real computers:
    - Arithmetic: add/sub, mul/div, remainder, shift left/right.
    - Control: conditional/unconditional branching, subroutine call and return.
    - Data movement: load, store, copy.
    - Each instruction takes a constant amount of time.

Random-access machine model

- *Random-access machine* (RAM) is a virtual computer with the following properties:
  - The RAM model uses integer and floating-point types of numeric data.
  - We don't worry about precision.
  - Assume a limit of the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$ ($\lg n$ is a commonly used shorthand for $\log_2 n$).

Analyzing an algorithm's running time

- *The time taken by an algorithm depends on the input:*
  - Sorting 1000 numbers takes longer than sorting 3 numbers.
  - A given sorting algorithm may even take different amounts of time on two inputs of the same size: it takes less time to sort *n* elements when they are already sorted than when they are sorted in reverse order.

  e.g. $\boldsymbol{A} = [1, 2, 3, 4, 5, 6]$ vs $\boldsymbol{B} = [6, 5, 4, 3, 2, 1]$

Analyzing an algorithm's running time

- *Input size depends on the problem being considered*:
  - Usually, the number of items in the input. Like the size *n* of the array being sorted.
  - Could be something else: if multiplying two integers, could be the total number of bits in the two integers.
  - Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Analyzing an algorithm's running time

- Finally, the ***running time*** is the number of primitive operations (steps or pseudocode lines) executed on a particular input.

Analyzing an algorithm's running time: Fundamentals

- Steps of an algorithm to be machine-independent.
- Each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i.

```
$i \leftarrow i - 1$                // $c_i$
$A[i + 1] \leftarrow$ key     // $c_{i+1}$
```

Analyzing an algorithm’s running time: Fundamentals

- The line consists only of primitive operations.
  - If the line is a function (subroutine) call, then the actual call takes constant time, but the execution of the function being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Examples: “sort the points by x-coordinate”, “sort an array”, etc.

Analysis of Insertion sort

Insertion Sort 1$^{st\text{-}a}$ algorithm

```
for $j$ ← 2 to length[$A$]
  do { key ← $A[j]$
      // Insert $A[j]$ into the sorted sequence $A[1…j–1]$
      $i$ ← $j$ – 1
      while ($i$ > 0) and ($A[i]$ > key)
          do { $A[i+1]$ ← $A[i]$
               $i$ ← $i$ – 1
               $A[i + 1]$ ← key
          }
      }
```

Analysis of Insertion sort

- Assume that the i^{th} line takes time c_i, which is a constant.
- For $j = 2, 3, \ldots, n$, where n = length[A], let t_j denote the number of times that the **`while`** loop test is executed for that value of j.
- Note that when a **`for`** or **`while`** loop exits in the usual way – due to the test in the loop header – the test is executed one time more than the loop body.
- Assume that comments are not executable statements.

The running time

- The running time = Σ_j (cost of the j^{th} statement) $\times$
  $\times$ (number of times statement is executed)=
  $= \boldsymbol{\Sigma\, c_j\, s_j}$

Insertion Sort 1st algorithm: the Running time

Statement	***Running Time***
InsertionSort(*A*, *n*)	
for $j \leftarrow 2$ to n	$c_1 n$
do { **key** $\leftarrow A[j]$	$c_2(n-1)$
// Insert $A[j]$ into the sorted sequence $A[1\ldots j-1]$	0
$i \leftarrow j-1$	$c_4(n-1)$
while $(i > 0)$ and $(A[i] > \text{key})$	$c_5\, T$
do { $A[i+1] \leftarrow A[i]$	$c_6\,(T(n-1))$
$i \leftarrow i-1$	$c_7\,(T(n-1))$
$A[i+1] \leftarrow \text{key}$	$c_8\,(T(n-1))$
}	
}	

$T = t_2 + t_3 + \ldots + t_n$, where t_j is the number of **while** expression evaluations for the j^{th} **for** loop iteration

Insertion Sort 1st algorithm: the Running time

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

What can ***T*(*n*)** be?

***Best case*: inner loop body never executed (the array is already sorted)**

$t_j = 1$ ➔ ***T*(*n*)** is a linear function

***Worst case*: inner loop body executed for all previous elements**

$t_j = j$ ➔ ***T*(*n*)** is a quadratic function

Average case

???

Insertion Sort 1st algorithm: the Running time – Best case

- ***Best case***: inner loop body never executed
  (the array is already sorted)

$$\boldsymbol{T(n)} = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + \underbrace{+ c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8 \sum_{j=2}^{n}(t_j - 1)}_{0} =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1)$$

- ***T*(*n*)** is a linear function

Insertion Sort 1st algorithm: the Running time – Worst case

- ***Worst case***: inner loop body executed for all previous elements (the array is initially sorted in the reverse order)

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

$$+c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8 \sum_{j=2}^{n} (j-1) =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \frac{n(2+n+1)}{2} +$$

$$+c_6 \frac{(n-1)(1+n-1)}{2} + c_7 \frac{(n-1)(1+n-1)}{2} + c_8 \frac{(n-1)(1+n-1)}{2}$$

  - $T(n)$ is a quadratic function

Insertion Sort 1st-a algorithm: the Running time

```
for $j$ ← 2 to length[$A$]
  do { key ← $A[j]$
       // Insert $A[j]$ into the sorted sequence $A[1…j–1]$
       $i$ ← $j$ – 1
       while ($i$ > 0) and ($A[i]$ > key)
           do { A[$i$+1] ← A[$i$]
                $i$ ← $i$ – 1
           }
       $A[i+1]$ ← key
       }
```

Insertion Sort 1st-a algorithm: the Running time

Statement	*Running Time*
InsertionSort(A, n)	
for $j \leftarrow 2$ to n	$c_1 n$
do { **key** $\leftarrow A[j]$	$c_2(n-1)$
// Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$	0
$i \leftarrow j-1$	$c_4(n-1)$
while $(i > 0)$ and $(A[i] >$ key$)$	$c_5\ T$
do { $A[i+1] \leftarrow A[i]$	$c_6\ (T(n-1))$
$i \leftarrow i-1$	$c_7\ (T(n-1))$
}	
$A[i+1] \leftarrow$ key	$c_8\ (n-1)$
}	

$T = t_2 + t_3 + \ldots + t_n$, where t_j is the number of **while** expression evaluations for the j^{th} **for** loop iteration

Insertion Sort 1st-a algorithm: the Running time

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

What can ***T*(*n*)** be?

Best case -- inner loop body never executed (the array is already sorted)

$t_j = 1$ ➔ ***T*(*n*)** is a linear function

Worst case -- inner loop body executed for all previous elements

$t_j = j$ ➔ ***T*(*n*)** is a quadratic function

Average case

???

Insertion Sort 1st-a algorithm: the Running time – Best case

- ***Best case***: inner loop body never executed (the array is already sorted)

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

$$\underbrace{+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1)}_{0} + c_8(n-1) =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

- $T(n)$ is a linear function

Insertion Sort 1st-a algorithm: the Running time – Worst case

- ***Worst case***: inner loop body executed for all previous elements (the array is initially sorted in the reverse order)

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

$$+c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1) \ =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \frac{n(2+n+1)}{2} +$$

$$+c_6 \frac{(n-1)(1+n-1)}{2} + c_7 \frac{(n-1)(1+n-1)}{2} + c_8(n-1)$$

- $\boldsymbol{T(n)}$ is a quadratic function

Running time: What is more important to analyze?

- Best case?
- Worst case?
- Average case?
- Some of them?
- All?

Running time: the worst case is the most interesting!

- The worst-case running time gives a guaranteed upper bound for any input.
- For many algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for empty items may be frequent.

Running time: Average case

- The average case is interesting and important, because it gives a closer estimation of the realistic running time.
- However, its consideration usually requires more efforts (algebraic transformations, etc.).
- On the other hand, it is often roughly as “bad” as the worst case.
- Hence, it is often enough to consider the worst case.

Running time

- The worst case is the most interesting.
- The average case is interesting, but often is as “bad” as the worst case and may be estimated by the worst case.
- The best case is the least interesting.

Activity

- Reorder the following efficiencies from smallest to largest:
  a) $n \log n$
  b) $n + n^2 + n^3 + n^4$
  c) 101
  d) n^3
  e) $n^5 \log n$