Algorithms for Data Science

CSOR W4246

Eleni Drinea

Computer Science Department

Columbia University

Tuesday, September 2, 2014

Outline

Algorithms

- **An algorithm** is a well-defined computational procedure that transforms the **input** (a set of values) into the **output** (a new set of values).
- The desired input/output relationship is specified by the statement of the **computational problem** for which the algorithm is designed.
- An algorithm is **correct** if, for every input, it **halts** with the correct output.

Efficient Algorithms

- In this course we are interested in algorithms that are **correct** and **efficient**.
- Efficiency is related to the **resources** an algorithm uses: time, space
  - *How much time/space are used?*
  - *How do they* **scale** *as the input size grows?*

We will primarily focus on efficiency in running time.

Running time

Running time of an algorithm: number of **primitive computational steps** performed.

- A line in a standard programming language (C or Java)
- For example, assigning a value to a variable, looking up an entry in an array, following a pointer, or performing an arithmetic operation on a *fixed-size* integer.

In general these fall in the following categories.

1. **Arithmetic operations**: add, subtract, multiply, divide
2. **Data movement operations:** load, store, copy
3. **Control operations:** branching, subroutine call and return

We will use pseudocode for our algorithm descriptions.

- In general one line of pseudocode could correspond to one computational step.

Our first problem: sorting

- **Input:** A list A of n integers $x_1, \ldots, x_n$.
- **Output:** A permutation $x'_1, x'_2, \ldots, x'_n$ of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \ldots \leq x'_n$

  Example: $A = \{9, 3, 2, 6, 8, 5\}$

What data structure should we use to represent a **list***?*

Our first problem: Sorting

- **Input:** A list A of n integers $x_1, \ldots, x_n$.
- **Output:** A permutation $x'_1, x'_2, \ldots, x'_n$ of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \ldots \leq x'_n$

  Example: $A = \{9, 3, 2, 6, 8, 5\}$

What data structure should we use to represent a **list***?*
An **array**.

- Collection of items of the same data type
- Random access
- "zero" indexed in C++ and Java

Insertion sort: intuition

- Start with a sorted subarray of size 1 consisting of the first element of your array.
- **Increase** the size of the sorted subarray by 1 at a time, by **inserting** the first element of A, call it key, that does not belong to the sorted subarray into the **correct** position.
  - Compare key with every element x of the sorted subarray starting from the **right**.
    - If $x > key$, move x one position to the right.
    - Otherwise ($x \leq key$), **insert** key after x.
  - Repeat until the sorted subarray has n elements.

Insertion sort pseudocode

Insertion-sort

```
for  i = 2 to A.length do
      key = A[i]
      //Insert A[i] into the sorted subarray A[1 ... i − 1]
      j = i − 1
     while j > 0 and A[j] > key do
         A[j + 1] = A[j]
         j = j − 1
     end while
      A[j + 1] =key
end for
```

Analysis of algorithms

- Correctness
- Running time
- Space

Analysis of algorithms

- Correctness: formal proof by induction
- Running time: count the number of primitive computational steps
  - This is not the same as **time** it takes to execute the algorithm.
  - We want a measure that is independent of hardware.
  - We want to know how running time **scales** with the size of the input.
- Space: typically easy to analyze

Analysis of Insertion Sort

- Correctness: show by induction that, for all $1 \leq i \leq n$, after loop i, the subarray $A[1..i]$ is sorted.
- Running time: count the number of primitive computational steps
- Space: Insertion sort sorts **in place**, that is, at most a constant number of elements of A are stored outside A at any time.

Example of induction

Fact

For all $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Proof.

- Base case: $n = 1$
- Inductive Hypothesis: Assume that the statement is true for $n \geq 1$, that is, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Inductive Step: We show that it is true for $n + 1$.
- Conclusion: It follows that the statement is true for all n since we can apply the inductive step for $n = 2, 3, \ldots$.

$\square$

Correctness of Insertion Sort

Theorem

Let $n \geq 1$ be a positive integer. For all $1 \leq i \leq n$, after the i-th loop, the subarray $A[1..i]$ is sorted.

Correctness of Insertion Sort (cont'd)

Proof.

By induction on i.

- **Base case:** $i = 1$, trivial.
- **Induction hypothesis:** assume that this statement is true for some i s.t. $1 \leq i < n$.
- **Inductive step:** Show it true for $i + 1$.
  In loop $i + 1$, element $key = A[i + 1]$ is inserted into $A[1..i]$.
  By the induction hypothesis, $A[1..i]$ is sorted.
  - key is inserted after the first element $x = A[\ell]$ for $1 \leq \ell \leq i$ s.t. $key \geq x$.
  - All elements in $A[1..i]$ greater than key are pushed one position to the right with their order preserved.

  Therefore, after loop $i + 1$, $A[1..i + 1]$ is sorted.

$\square$

Running time of Insertion Sort

for $i = 2$ to A.length **do**
  key $= A[i]$
  //Insert $A[i]$ into the sorted subarray $A[1 \ldots i-1]$
  $j = i - 1$
  while $j > 0$ and $A[j] >$ key **do**
    $A[j+1] = A[j]$
    $j = j - 1$
  end while
  $A[j+1] =$key
end for

- Let t_i be # times the line of the while loop is executed for i.
  - *How many computational steps are executed? Equivalently, what is the running time $T(n)$ of the algorithm?*
  - *Bounds on t_i?*

Running time of Insertion Sort

for $i = 2$ to A.length **do**
  key $= A[i]$
  //Insert $A[i]$ into the sorted subarray $A[1 \ldots i-1]$
  $j = i - 1$
  while $j > 0$ and $A[j] >$ key **do**
    $A[j+1] = A[j]$
    $j = j - 1$
  end while
  $A[j+1] =$key
end for

- Let t_i be # times the line of the while loop is executed for i. Then
$$T(n) = 3\sum_{i=2}^{n} t_i + 2n - 1$$
- **Best** case running time?
- **Worst** case running time?

Running time of Insertion Sort

for $i = 2$ to A.length **do**
  key $= A[i]$
  //Insert $A[i]$ into the sorted subarray $A[1 \dots i-1]$
  $j = i - 1$
  while $j > 0$ and $A[j] >$ key **do**
    $A[j+1] = A[j]$
    $j = j - 1$
  end while
  $A[j+1] =$key
end for

- Let t_i be # times the while loop is executed for value i.

$$T(n) = n + 3(n-1) + \sum_{i=2}^{n} t_i + 2\sum_{i=2}^{n}(t_i - 1) = 3\sum_{i=2}^{n} t_i + 2n - 1$$

- **Best** case running time? $5n - 4$
- **Worst** case running time? $\frac{3n^2}{2} + \frac{7n}{2} - 4$

Efficient Algorithms

- *Is insertion-sort efficient?*

Worst-case running time

- *Is insertion-sort efficient?*
  Compare to **brute force** solution:
  - At each step, generate a new permutation of the n integers.
  - If the permutation is sorted, stop and output the permutation.
    - Example: $n = 3$, $A = \{2, 5, 1\}$. Generate 3! permutations.
    - For general n, generate "at most" $n!$ permutations.

Worst-case running time

- *Is insertion-sort efficient?*
  Compare to **brute force** solution:
  - At each step, generate a new permutation of the n integers.
  - If the permutation is in non-decreasing order, stop and output the permutation.
    - Example: $n = 3$, $A = \{2, 5, 1\}$. Generate 3! permutations.
    - For general n, generate "at most" $n!$ permutations.
- **Worst-case running time:** largest possible running time of the algorithm over all inputs of a given size n.
- **Worst-case running time analysis**
  - gives well-defined computable bounds
  - average-case analysis can be tricky: how do we generate a "random" instance?
- *Is brute force solution efficient?*

Brute force approach for sorting

- Stirling's approximation formula: $n! \approx \left(\frac{n}{e}\right)^n$
  - To sort 1000 numbers, we might need go over $367^{1000} \geq 2^{7000}$ permutations!

$\Rightarrow$ Brute force solution is **not** efficient (efficiency is related to the performance of the algorithm as n grows).

- *So when is an algorithm efficient?*

Definition (Attempt 1)

An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Efficient algorithms: definition

- Caveats of the definition
  - **Scaling properties:** If the input size grows by a constant factor, we would like the running time $T(n)$ of the algorithm to increase by a constant factor as well.
- **Polynomial running time**: on input of size n, $T(n)$ is at most $c \cdot n^d$ for $c, d > 0$ constants
  - The *smaller the exponent* of the polynomial the better.

Definition

An algorithm is efficient if it has a polynomial running time.

Efficient algorithms: improved definition

Caveats

- What about huge constants in front of the leading term or large exponents?

Pros

- **Small degree polynomial** running times exist for most problems that can be solved in polynomial time.
- Conversely, problems for which no polynomial-time algorithm is known tend to be very hard in practice.
- So we can distinguish between **easy** and **hard** problems.

△ Data science: even low degree polynomial might be too slow.

Are we done with sorting?

Insertion sort is efficient.

1. *Can we do better?*
2. *And what is "better"?*
   - *E.g., is* $T(n) = n^2 + n - 4$ *worth aiming for?*

Running time in terms of # primitive steps

To discuss this, we need a coarser classification of running times for algorithms than the one obtained for insertion sort: such exact characterizations

- Are **too detailed**
- Do not reveal similarities between running times in an immediate and useful way as n grows large.
- Are often **meaningless**: pseudocode steps will **expand** by a constant factor depending on the hardware.

Coming up...

- In the next class, we will present a framework that will allow us to compare the **rate of growth** of different running times.
  - We will express the running time as a function of the **number** of primitive steps.
    - The latter is a function of the size n of the input.
  - To compare functions expressing running times, we will ignore their low-order terms altogether and focus solely on the highest-order term.
- We will present a better algorithm for sorting that uses the "divide-and-conquer" principle.

Algorithms for Data Science
CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Tuesday, September 2, 2014