Lecture 2: Introduction to algorithm analysis

CS303: Algorithms

January 10, 2014

1 Definition of algorithms

What is an algorithm?
Informal: What bill Clinton plays on his saxophone? - Al-Gore-Rhythm.
Formal: An algorithm is a procedure (a finite set of unambiguous instructions) for solving a problem. i.e. for obtaining a required output for any legitimate input in a finite amount of time.

Requirement:

- Finiteness (terminate after a finite number of steps)
- Definiteness (rigorously and unambiguously specified)
- Input (valid inputs are clearly specified)
- Output (can be proved to produce the correct output)
- Effectiveness (Steps are sufficiently simple and basic, i.e. codable)

Q:/ Why do we study algorithms?
A:/ **Theoretically:** The core of computer science (or the entire western civilization!)
Practically: Toolkit for you to practice (more than 50% of job interview questions at google/Microsoft are algorithm questions! Google, Amazon, are all based on good ideas of algorithms!)
Framework for designing and analyzing new algorithms for new problems (Solve problems before the end of the world!)

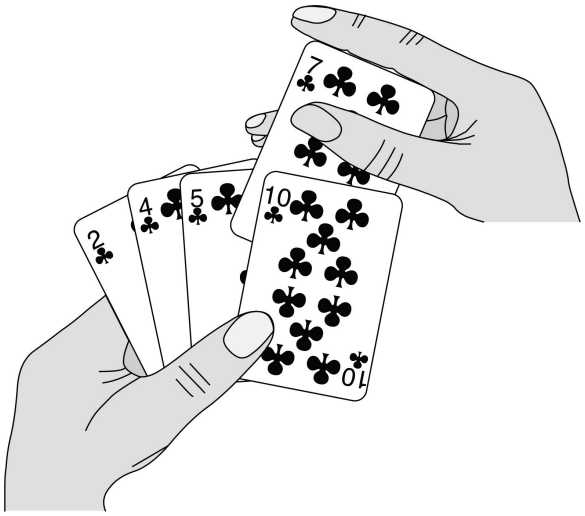
2 Introduction to algorithm analysis: sorting

Sorting is one of the most important (oldest) tasks computers do in everyday life (sorting by name, by time, etc)
Definition:

Input: a sequence of $< a_1, a_2, \ldots, a_n >$ of numbers
Ouput: permutation of the input sequence $< a'_1, a'_2, \ldots, a'_n >$, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

2.1 Insertion sort

Idea: the way we use to sort a deck of poker cards



We want to separate the original list into sorted part (in the front) and unsorted part (after the sorted part). Every time we make a move, the sorted part's length is increased by 1 and the unsorted part decreases by 1.
We usually write algorithms in pseudo-codes (fake programming language written in English)

```
Algorithm: Insertion-Sort(A,n)
for j from 2 to n
do
  key ← A[j]
  i ← j-1
  while i>0 and A[i]>key
  do
    A[i+1] ← A[i]
    i ← i-1
  end
  A[i+1] ← key
end
```

We basically shifted all the elements to the right in the sorted part until we can insert the key into the right position.

Example: sort 8 4 2 9 3 6
5 rounds of inserting

1. 4 8 2 9 3 6
2. 2 4 8 9 3 6
3. 2 4 8 9 3 6

4. 2 3 4 8 9 6
5. 2 3 4 6 8 9

2.1.1 Proof of correctness

Loop invariant (description of a part of the loop that doesn't change as loop goes on)
A[1...j-1] consists of the elements originally in $A[1 \dots j-1]$ but in sorted order.
Q:/ Why does it matter?
A:/ Help us prove the correctness of the algorithm

To show that, we must follow something that is similar to induction

1. Initialization: the loop invariant is true prior to the first iteration
2. Maintenance: if it is true before an iteration, then it is true before the next iteration
3. Termination: When the loop terminates, the loop invariant gives us what we want. $\rightarrow$ correctness of our algorithm.

For insertion sort,

1. Initialization: when j=2, then A[1] is just an element and it is sorted
2. Maintenance: if $A[1 \dots j-1]$ satisfies the loop invariant, then $A[1, \dots, j]$ is basically insert the j^{th} element into $A[1, \dots, j-1]$ to make all j element sorted
3. Termination: if $j = n+1$, Then $A[1 \dots n]$ is the subarray (i.e. the whole array)

Thus insertion sort is correct.

2.1.2 Analysis of Insertion sort

Running time:
Depend on input
input itself: e.g. if it is already sorted
Q:/ what is the best and worst case of insertion sort?
A:/ best is already sorted and worst is reversely sorted
input size: e.g. 6 elements vs. 6,000,000,000 items
Thus we will usually parameterize on the input size on different cases

Note: generally we want the upper bound (i.e. the running time is no more than that) $\rightarrow$ gives guarantee to the user (at most 3 seconds vs at least 3 seconds)

Kinds of analyses:

- Worst case: $T(n)$ as the max time of any input of size n (it is input invariant and it is a function)
- Average case: expected running time of all inputs of size n (we need to assume the distribution of inputs. $\rightarrow$ hard without probability requirement)

- Best case: min time of all inputs of size n (bogus!! no good because you can cheat. An algorithm that has a good best case may not be applicable at all. You can't compare algorithms with their best cases)

It also depends on the computer/programmer → compare them for relative speed (on the same machine with the same programmer)
This leads to the **big idea: asymptotic analysis** (why algorithm analysis is so successful)
Asymptotic analysis

- ignore machine dependent constants
- look at the growth of T(n) as n → inf