Introduction, Insertion Sort, and Loop Invariance

Joe Song

January 23, 2020

The lecture notes are mostly based on Chapter 2 of Cormen, Leiserson, Rivest, and Stein. Introduction to Algorithms. 3rd Ed. 2009. MIT Press. Cambridge, Massachusetts.

Contents

1 Motivation

1. Understanding the complexity of algorithms is used to be a more theoretical branch of computer science
2. Big data applications beg for efficient algorithms!
3. Job interviews now frequently test on
   - algorithmic details
   - implementations

1.1 Example. Big Data Sources

Name/Symbol				Short Scale	Adoption Year
yotta / Y	1000^8	10^{24}	1,000,000,000,000,000,000,000,000	septillion	1991
zetta / Z	1000^7	10^{21}	1,000,000,000,000,000,000,000	sextillion	1991
exa / E	1000^6	10^{18}	1,000,000,000,000,000,000	quintillion	1975
peta / P	1000^5	10^{15}	1,000,000,000,000,000	quadrillion	1975
tera / T	1000^4	10^{12}	1,000,000,000,000	trillion	1960
giga / G	1000^3	10^9	1,000,000,000	billion	1960

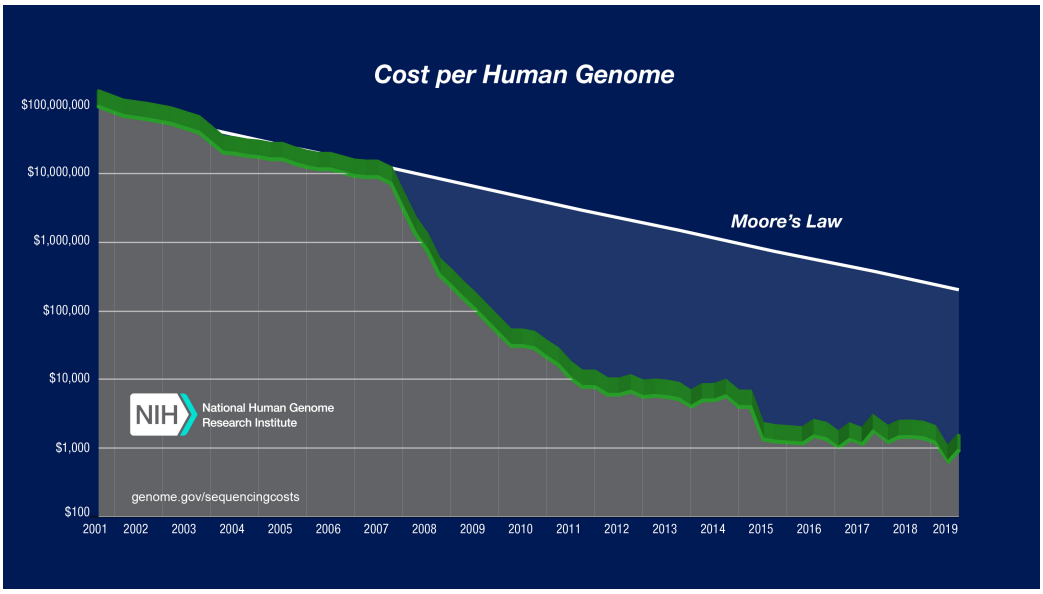
Source: Wikipedia

Data Phase	Astronomy	Twitter	YouTube	Genomics
Acquisition	25 zetta-bytes/year	0.5–15 billion tweets/year	500–900 million hours/year	1 zetta-bases/year
Storage	1 EB/year	1–17 PB/year	1–2 EB/year	2–40 EB/year
Analysis	In situ data reduction	Topic and sentiment mining	Limited requirements	Heterogeneous data and analysis
	Real-time processing	Metadata analysis		Variant calling, ~2 trillion central processing unit (CPU) hours
	Massive volumes			All-pairs genome alignments, ~10,000 trillion CPU hours
Distribution	Dedicated lines from antennae to server (600 TB/s)	Small units of distribution	Major component of modern user's bandwidth (10 MB/s)	Many small (10 MB/s) and fewer massive (10 TB/s) data movement

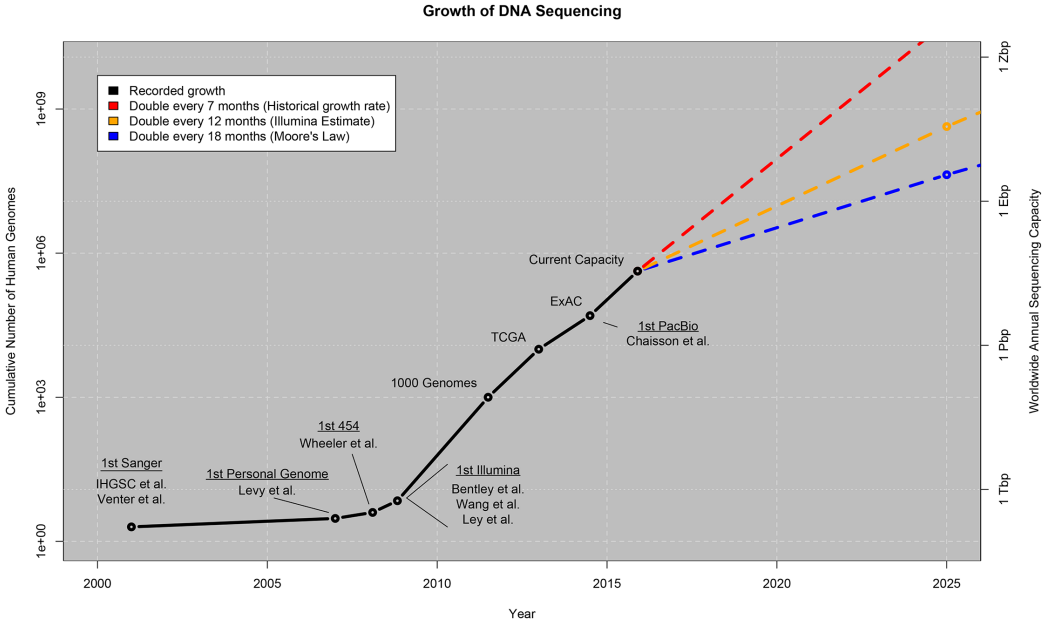
doi:10.1371/journal.pbio.1002195.t001

References: Stephens et al., (2015) Big Data: Astronomical or Genomical? PLoS Computational Biology. DOI: 10.1371/journal.pbio.1002195

1.2 Next Generation Sequencing (NGS)



http://www.genome.gov/sequencingcosts/



http://en.wikipedia.org/wiki/European_Nucleotide_Archive

Reference: Richter BG, Sexton DP (2009) Managing and Analyzing Next-Generation Sequence Data. PLoS Comput Biol 5(6): e1000369. doi:10.1371/journal.pcbi.1000369

Some of the techniques and manufacturers include

- sequencing by synthesis as used by the Solexa Genome Analyzer II (GAII) by Illumina,
- sequencing by ligation as used by the ABI SOLiD sequencer and by the polony sequencing technique developed by the Church Lab at Harvard Medical School,
- sequencing by hybridization as used by Affymetrix, and
- single molecule sequencing as used by Helicos, VisiGen, and Pacific Biosciences.

2 Insertion-Sort

- Demonstrate the idea of Insertion-Sort by an example

  $A = \langle 5, 2, 4, 6, 1, 3 \rangle$

- Derive the pseudo-code of Insertion-Sort

  Assume the array index starts with 1.

```
INSERTION-SORT(A)
1  for j = 2 to length[A]
2      key = A[j]
3      // Insert A[j] to the sorted sequence A[1 .. j − 1]
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

2.1 The proof technique of loop invariant

Come up with a good loop invariant statement.

Initialization: The loop-invariant is true prior to the first iteration.

Maintenance: If loop-invariant is true before each iteration, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives a useful property that helps to show the algorithm is correct.

2.2 Correctness proof by loop invariant

Apply loop invariant to the proof of Insertion-Sort. Best done by illustrating using a diagram.

What is the logic of correctness proof of loops? Similar to mathematical induction.

2.2.1 Correctness of the outer for-loop

Loop invariant: "At the start of j-th **for** loop, elements in $A[1 \ldots j-1]$ are sorted."

Initialization: $A[1]$ is sorted (array with only one element)

Maintenance: Before each loop j, $A[1\,..\,j-1]$ is sorted.

After each loop j, $A[1\,..\,j]$ is sorted because key is inserted to the right position (assuming the inner loop is correct).

Termination: Let $n = length[A]$.

When the loop terminates, $j = n + 1$. According to loop-invariant, $A[1\,..\,j-1]$ is sorted, which is exactly $A[1\,..\,n]$.

2.2.2 Correctness of the inner while-loop

After the **while**-loop, the sub array $A[1\,..\,j]$ will be sorted.

Loop invariant: At the start of each **while** iteration i ($i \leq j-2$):

1. $A[i+2\,..\,j]$ contains the largest $j-i-1$ elements in the original $A[1\,..\,j-1]$,

2. $A[i+2\,..\,j]$ is sorted,

3. $A[i+2] > key$.

Initialization: The base will start after the first loop:

- Before the first loop: $i = j-1$,

1	2	$\cdots$	$j-2$	$j-1$	j
$A[1] <$	$A[2] <$	$\cdots$	$A[j-2] <$	$A[j-1]$	$A[j] = key$
				i	

- After the first loop: $A[j] = A[j-1] > key$.

  1. Since $A[1\,..\,j-1]$ was already sorted and $A[j-1] > key$, $A[j\,..\,j]$ now contains the largest element in the original $A[1\,..\,j-1]$.

2. $A[i+2\,..\,j] = A[j\,..\,j]$ is always sorted with a single element.

3. Evidently $A[j] > key$.

After $i = i-1$, $i = j-2$. Thus, we have $A[i+2] = A[j] > key$.

- Before the second loop:

1	2	$\cdots$	$j-2$	$j-1$	j
$A[1]$	$A[2]$	$\cdots$	$A[j-2]$	$key < A[j-1] =$	$= A[j] > key$
			i		

Maintenance:

If $A[i+2\,..\,j]$ is sorted and if the loop does not terminate, i.e., $(i > 0, A[i] > key)$, then

$$key < A[i] < A[i+2]$$

The 2nd "<" is due to the fact that $A[i+2\,..\,j]$ contains the largest elements in the original array $A[1\,..\,j-1]$.

After line 6, $A[i+1\,..\,j]$ is sorted and contains the largest $j-i$ elements in the original array $A[1\,..\,j-1]$ and $A[i+1] > key$

1	2	$\cdots$	i	$i+1$	$i+2$	$\cdots$	j
$A[1]$	$A[2]$	$\cdots$	$A[i]$	$key < A[i+1] <$	$A[i+2] <$	$\cdots <$	$A[j]$
			i				

After line 7 $i = i-1$, we have $A[i+2\,..\,j]$ contains the largest $j-i-1$ elements in the original array $A[1\,..\,j-1]$ and $A[i+2] > key$.

We have just shown that the loop-invariant maintains after an iteration.

1	2	$\cdots$	i	$i+1$	$i+2$	$\cdots$	j
$A[1]$	$A[2]$	$\cdots$	$A[i]$	$A[i+1]$	$A[i+2] <$	$\cdots <$	$A[j]$
			i				

Termination: There are three cases to cover for termination.

- If the loop terminates before the 1st iteration, we have $A[j-1] \leq A[j]$. Since $A[1..j-1]$ is sorted, it follows that $A[1..j]$ is sorted.
- If the loop terminates due to $i == 0$, then we have $A[2..j]$ sorted and $A[2] > key$ based on the loop-invariant. After line 8, we get $A[1] = key$, it leads to the sorted $A[1..j]$.
- If the loop terminates due to $A[i] \leq key$, then $A[i+2..j]$ is sorted with the largest $j-i-1$ in the original $A[1..j-1]$. Evidently, $A[1..i]$ has the smallest i elements sorted in the original $A[1..j-1]$ since they are not touched by the loop. After line 8, we have

  $$A[1..i] \leq A[i+1] = key < A[i+2..j]$$

  where the first "$\leq$" was based on the while loop termination comparison, and the second "$<$" was based on the loop invariant. Since both $A[1..i]$ and $A[i+2..j]$ are sorted, we have $A[1..j]$ sorted.

2.3 Running time analysis

- General rules for time analysis: constants are used to denote the running time of statement independent of input size.

- Count the cost of each statement and the times of execution.
- Solve the total running time

Let t_j be the number of iterations in the while-loop when $A[j]$ is being processed.

INSERTION-SORT(A)	cost * times
1 **for** $j = 2$ **to** $length[A]$	$c_1 * n$
2 $key = A[j]$	$c_2 * (n-1)$
3 // Insert $A[j]$ to the sorted sequence $A[1..j-1]$	$0 * (n-1)$
4 $i = j - 1$	$c_4 * (n-1)$
5 **while** $i > 0$ and $A[i] > key$	$c_5 * \sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	$c_6 * \sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	$c_7 * \sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	$c_8 * (n-1)$

Best case: when A is already sorted $t_j = 1$ for any j

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

Worst case: when A is sorted in reverse order, $t_j = j$

$$\begin{aligned} T(n) = {} & 0.5(c_5 + c_6 + c_7)n^2 \\ & + 0.5(2c_1 + 2c_2 + 2c_4 + c_5 - c_6 - c_7 + 2c_8)n \\ & - (c_2 + c_4 + c_5 + c_8) \end{aligned} \quad (1)$$

which is a **quadratic function** of n.

3 Summary (Generalization)

What do we do in algorithm design and analysis?

- Design: Come up with an strategy to solve a problem.

  - incremental approach
  - divide-and-conquer
  - greedy algorithms
  - dynamic programming
  - graph algorithms
- Correctness: Argue logically whether an algorithm solves or does not solve a problem correctly
  - loop invariance
  - contrapositives
  - by counter example
- Time analysis: Explain how the running time of an algorithm grows as a function of the input size.
  - efficient: running time is at most a polynomial function of input size

    E.g., n^2, $\log n$, $\sqrt{n}$, n^{100}.
  - NP-complete: a class of problems whose worst-case running time remains unclear – they could be solved efficiently in polynomial time. The biggest actively pursued open problem in theoretical computer science.
  - inefficient: running time is at least an exponential function of input size

    E.g., e^n, 2^n, $n!$
  - Why are we interested in efficient algorithm design?
    * An intellectual pursuit.

* Practical application demands. E.g. Human and mouse genomes each has 3 billion (3×10^9) letters in it.

  A short chromosome has about 50 million letters.

  An optimal alignment algorithm to align a human chromosome and a mouse chromosome each at 50 M long will take

  $$50,000,000 \times 50,000,000 \approx 2.5 \times 10^{15}$$

  We assume a computer can process 10G instructions per second. It is going to take 2.5×10^5 seconds which is about 3 days to finish.

  If one can find a linear algorithm to do it, i.e., the running time is $50,000,000$, it is going to take only 5 milliseconds.

  However, no one knows how to do it exactly in linear time! (though linear-time approximate solutions do exist and are widely used! – BLAST)