Introduction to Algorithms

Dr. Navjot Singh

Design and Analysis of Algorithms

Algorithms

- ***Informally***,
  - A tool for solving a well-specified computational problem.

- **Example: sorting**

  input: A sequence of numbers.

  output: An ordered permutation of the input.

  issues: correctness, efficiency, storage, etc.

Strengthening the Informal Definiton

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness: Total number of steps used in algorithm should be finite.
  - Definiteness: Each step of algorithm must be clear and unambiguous.
  - Input: The algorithm must accept zero or more input.
  - Output: The algorithm must produce at least one output.
  - Effectiveness: Each step must be basic and essential.

Algorithm – In Formal Terms…

- In terms of mathematical models of computational platforms (general-purpose computers).
- One definition – **Turing Machine that *always* halts**.
- Other definitions are possible (e.g. Lambda Calculus.)
- Mathematical basis is necessary to answer questions such as:
  - Is a problem solvable? (Does an algorithm exist?)
  - Complexity classes of problems. (Is an efficient algorithm possible?)

Algorithm Analysis

- Determining performance characteristics.
  (Predicting the resource requirements.)
  - Time, memory, communication bandwidth etc.
  - **Computation time** (running time) is of primary concern.
- Why analyze algorithms?
  - **Choose** the **most efficient** of several possible algorithms for the same problem.
  - Is the best possible **running time** for a problem ***reasonably finite*** for practical purposes?
  - Is the algorithm **optimal** (best in some sense)? – Is something better possible?

Running Time

- Run time expression should be machine-independent.
  - Use a model of computation or “hypothetical” computer.
  - Our choice – **RAM model** (most commonly-used).
- Model should be
  - Simple.
  - Applicable.

RAM Model

- Generic single-processor model.
- **Supports simple constant-time instructions** found in real computers.
  - Arithmetic (+, –, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy).
  - Control (branch, subroutine call).
- Run time (**cost**) is uniform (**1 time unit**) for all simple instructions.
- Memory is unlimited.
- Flat memory model – no hierarchy.
- Access to a word of memory takes **1 time unit**.
- Sequential execution – **no concurrent operations**.

Model of Computation

- Should be simple, or even simplistic.
  - Assign uniform cost for all simple operations and memory accesses. (Not true in practice.)
  - **Question:** **Is this OK?**
- Should be widely applicable.
  - Can’t assume the model to support complex operations.

    Ex: **No SORT instruction.**
  - Size of a word of data is finite.
  - **Why?**

Running Time – Definition

- Call each simple instruction and access to a word of memory a "primitive operation" or "step."
- Running time of an algorithm for a given input is
  - The **number of steps** executed by the algorithm on that **input**.
- Often referred to as the ***complexity*** of the algorithm.

Complexity and Input

- Complexity of an algorithm generally depends on
  - **Size of input.**
    - Input size depends on the problem.
      - Examples: No. of items to be sorted.
      - No. of vertices and edges in a graph.
  - **Other characteristics of the input data.**
    - Are the items already sorted?
    - Are there cycles in the graph?

Runtime examples

	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
$n = 100$	< 1 sec	< 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

Some examples

- O(1) – constant. Fixed amount of work, regardless of the input size
  - add two 32 bit numbers
  - determine if a number is even or odd
  - sum the first 20 elements of an array
  - delete an element from a doubly linked list
- O(log *n*) – logarithmic. At each iteration, discards some portion of the input (i.e. half)
  - binary search

Some examples

- O(n) – linear. Do a constant amount of work on each element of the input
  - find an item in a linked list
  - determine the largest element in an array
- O(n log n) log-linear. Divide and conquer algorithms with a linear amount of work to recombine
  - Sort a list of number with MergeSort
  - FFT

Some examples

- O(n^2) – quadratic. Double nested loops that iterate over the data
  - Insertion sort
- O(2^n) – exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming
- O(n!)
  - Enumerate all permutations
  - determinant of a matrix with expansion by minors

Worst, Average, and Best-case Complexity

- Worst-case Complexity
  - **Maximum** steps the algorithm takes for any possible input.
  - Most tractable measure.
- Average-case Complexity
  - **Average** of the running times of all ***possible* inputs**.
  - Demands a definition of probability of each input, which is usually difficult to provide and to analyze.
- Best-case Complexity
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?

Pseudo-code Conventions

- Indentation (for block structure).
- Value of loop counter variable upon loop termination.
- Conventions for compound data. Differs from syntax in common programming languages.
- Call by value **not** reference.
- Local variables.
- Error handling is omitted.
- Concerns of software engineering ignored.
- …

Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false*

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	c_1	1
while $i \le n$ **and** A[i] != *key*	c_2	x
do i++	c_3	x-1
if $i \le n$	c_4	1
then return *true*	c_5	1
else return *false*	c_6	1

x ranges between 1 and n+1.

So, the running time ranges between

$c_1 + c_2 + c_4 + c_5$ – **best case**

and

$c_1 + c_2(n+1) + c_3 n + c_4 + c_6$ – **worst case**

Example – *Linear Search*

INPUT: **a sequence of *n* numbers, *key* to search for.**

OUTPUT: ***true*** **if *key* occurs in the sequence, *false***

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	1	1
while $i \leq n$ **and** A[i] != *key*	1	x
do i++	1	x-1
if $i \leq n$	1	1
then return *true*	1	1
else return *false*	1	1

Assign a cost of 1 to all statement executions.

Now, the running time ranges between

$1+ 1+ 1 + 1 = 4$ – **best case**

and

$1+ (n+1)+ n + 1 + 1 = 2n+4$ – **worst case**

Example – *Linear Search*

INPUT: **a sequence of *n* numbers, *key* to search for.**

OUTPUT: ***true*** **if *key* occurs in the sequence, *false***

LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \le n$ **and** A[i] != *key*	1	x
3 **do** i++	1	x-1
4 **if** $i \le n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

If we assume that we search for a random item in the list, on an average, Statements 2 and 3 will be executed $n/2$ times. Running times of other statements are independent of input.

Hence, **average-case complexity** is $1+ n/2+ n/2 + 1 + 1 = n+3$

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - **Ex:** $O(n)$, $\Theta(1)$, $\Omega(n^2)$, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - **Linear Search**: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**
- More on O, Θ, and Ω in next class. Use Θ for the present.

Comparison of Algorithms

- Complexity function can be used to compare the performance of algorithms.
- Algorithm *A* is more efficient than Algorithm *B* for solving a problem, if the complexity function of *A* is of lower order than that of *B*.
- Examples:
  - **Linear Search** – $\Theta(n)$ vs. **Binary Search** – $\Theta(\lg n)$
  - **Insertion Sort** – $\Theta(n^2)$ vs. **Quick Sort** – $\Theta(n \lg n)$

Comparisons of Algorithms

- **Multiplication**
  - classical technique: $O(nm)$
  - divide-and-conquer: $O(nm^{\ln 1.5}) \sim O(nm^{0.59})$

    For operands of size 1000, takes 40 & 15 seconds respectively on a Cyber 835.
- **Sorting**
  - insertion sort: $\Theta(n^2)$
  - merge sort: $\Theta(n \lg n)$

    For 10^6 numbers, it took 5.56 hrs on a supercomputer using machine language and 16.67 min on a PC using C/C++.

Why Order of Growth Matters?

- Computer speeds double every two years, so why worry about algorithm speed?
- When speed doubles, what happens to the amount of work you can do?
- What about the demands of applications?

Effect of Faster Machines

No. of items sorted

H/W Speed / *Comp. of Alg.*	1 M*	2 M	Gain
$O(n^2)$	1000	1414	1.414
$O(n \lg n)$	62700	118600	1.9

* Million operations per second.

- Higher gain with faster hardware for more efficient algorithm.
- Results are more dramatic for more higher speeds.

Correctness Proofs

- **Proving (beyond “any” doubt) that an algorithm is correct.**
  - Prove that the algorithm produces correct output when it terminates. **Partial Correctness.**
  - Prove that the algorithm will necessarily terminate. **Total Correctness.**
- **Techniques**
  - Proof by Construction.
  - Proof by Induction.
  - Proof by Contradiction.

Loop Invariant

- **Logical expression with the following properties.**
  - Holds true before the first iteration of the loop – **Initialization.**
  - If it is true before an iteration of the loop, it remains true before the next iteration – **Maintenance**.
  - When the loop terminates, the **invariant — along with the fact that the loop terminated —** gives a useful property that helps show that the loop is correct – **Termination**.
- **Similar to mathematical induction.**
  - Are there differences?

Correctness Proof of Linear Search

- Use **Loop Invariant** for the while loop:
  - At the start of each iteration of the while loop, the search *key* is not in the subarray A[1..*i*-1].

```
LinearSearch(A, key)
   i ← 1
   while i ≤ n and A[i] != key
         do i++
    if  i ≤ n
         then return true
         else  return false
```

- If the algm. terminates, then it produces correct result.
  - Initialization.
  - Maintenance.
  - Termination.
- Argue that it terminates.

Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill