Data Structures and Algorithms

Chapter 4

Learning Objectives

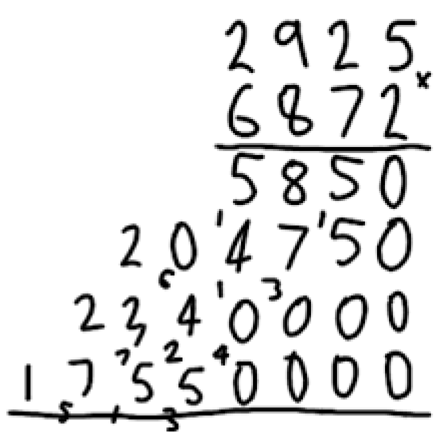
- Learn what algorithm analysis is
- Understand big-oh notation
- Be able to calculate worst-case runtime given an algorithm (in pseudocode or real implementation)

Algorithm Analysis
Basic Concepts

- A **problem** is an input-output relationship
- An **algorithm** is a finite sequence of steps which solves a problem.

Example:

- **Problem –** multiplication of n digit numbers
- **Algorithm** – grade school multiplication algorithm



Algorithm Analysis

Basic Concepts

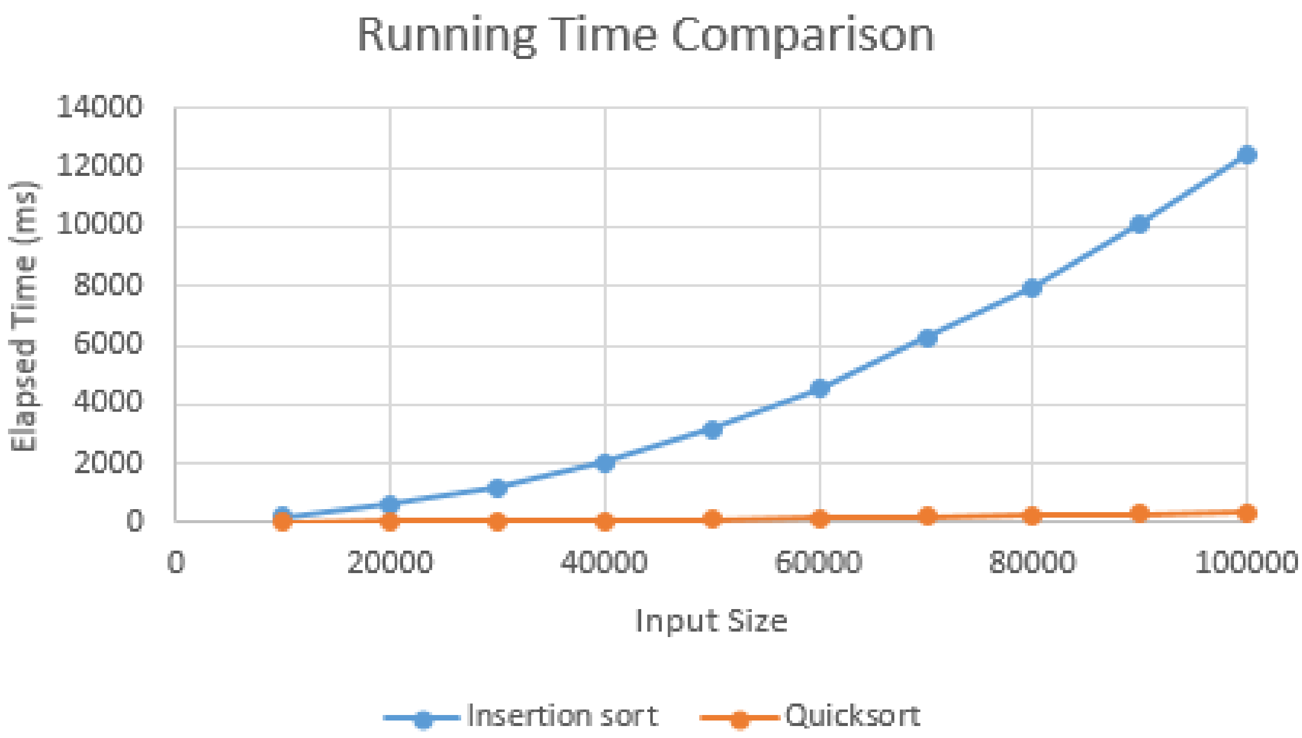
- A **problem** is an input-output relationship
- An **algorithm** is a finite sequence of steps which solves a problem.

- Efficiency of algorithms can be analyzed in terms of memory/space usage and in terms of running time.
- We will focus on running time analysis.

- Running time of an algorithm depends on the input size.
- We express running time as a function of the input size n.

Algorithm Analysis
Basic Concepts

- Running times of two sorting algorithms



Algorithm Analysis

Why not just compare different approaches by implementing them and measuring run time?

- Experiments must be performed on the same hardware/software configurations
- Experiments can only be done on a limited set of test inputs
- Algorithm must be fully implemented in order to study it experimentally

Algorithm Analysis
Basic Concepts

- Running times of an algorithm may be different for different inputs of the same size.
- For example, elapsed times of insertion sort algorithm on an array of 100,000 integers:
  - Best case: 1 ms, when elements are sorted in nondecreasing order
  - Average case: 12,145 ms, when elements are randomly distributed
  - Worst case: 24,810 ms, when elements are sorted in the reverse order
- Often, we perform only the worst-case analysis

Algorithm Analysis
Mathematical Functions

- $f(n) = c$ (constant)
- $f(n) = c \log n$ (log n)
- $f(n) = cn$ (linear)
- $f(n) = cn \log n$ (n log n)
- $f(n) = cn^2$ (quadratic)

Algorithm Analysis
Rate of Growth

- Rate of growth of different functions:

n	log n	n	n log n	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38
256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154

Algorithm Analysis

Rate of Growth

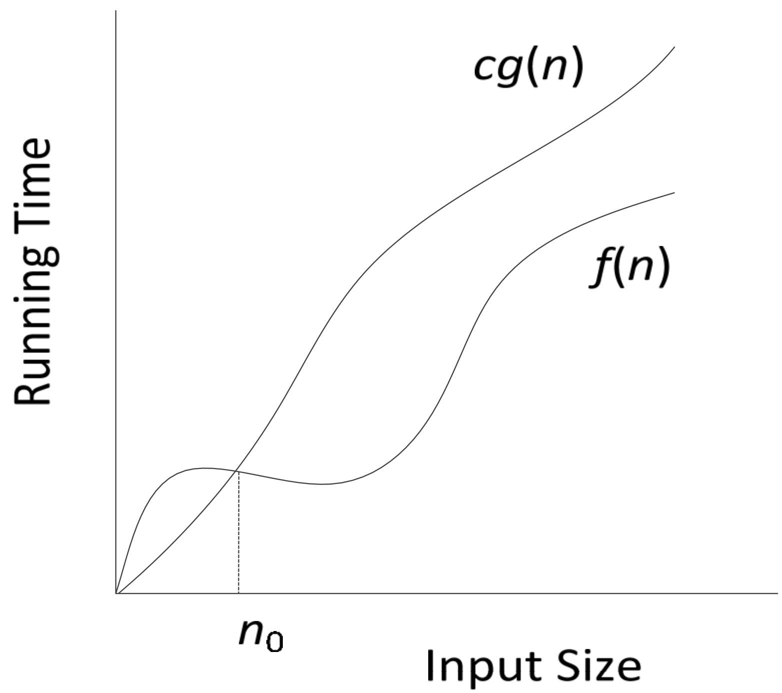
- When we analyze the running time of algorithms, we do not look at the actual running times.
- Instead, we focus on the rate of growth, i.e., how fast or slowly the running time grows as the input size increases.
- This is called *asymptotic analysis*.

- We use O (big-oh), Ω (big-omega), and Θ(big-theta) notations.

Algorithm Analysis

Rate of Growth

- $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0\}$



Algorithm Analysis

Rate of Growth

- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = O(n^3)$

  Proof:

  $f(n) = 5n^3 + 2n^2 + 8n + 4$

  $\leq 5n^3 + 2n^3 + 8n^3 + 4n^3$

  $= 19n^3$

  If we let $g(n) = n^3$, $c = 19$ and $n_0 = 1$, then

  $f(n) \leq cg(n)$ for all $n \geq n_0$

  So, $f(n) = O(n^3)$

- Note: $f(n) = O(n^4)$, $f(n) = O(n^5)$, . . .
- We always look for “lowest” function.

Algorithm Analysis

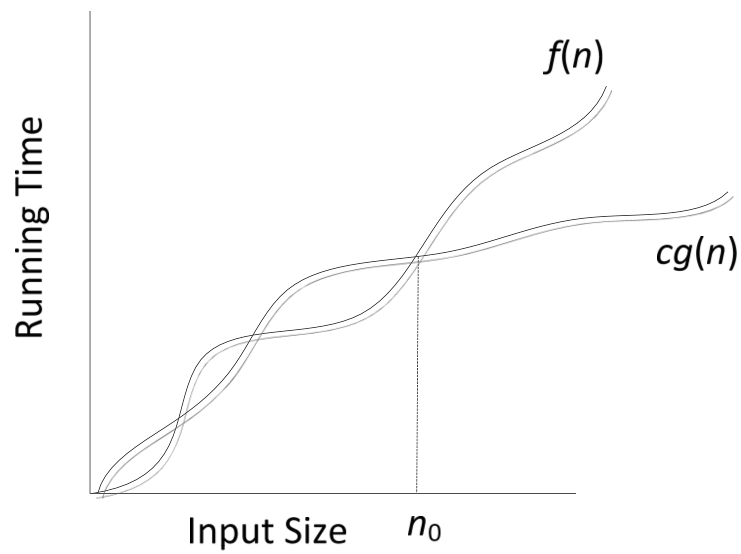
Rate of Growth

- $f(n) = 3n + 2 \Rightarrow O(n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow O(n^3)$
- $f(n) = 2n^2 + 2n \log n + 2n + 4 \Rightarrow O(n^2)$
- $f(n) = 2n \log n + 10n - 6 \Rightarrow O(n \log n)$
- $f(n) = 5n + 23\log n \Rightarrow O(n)$
- $f(n) = 3\log n + 10 \Rightarrow O(\log n)$

Algorithm Analysis

Rate of Growth

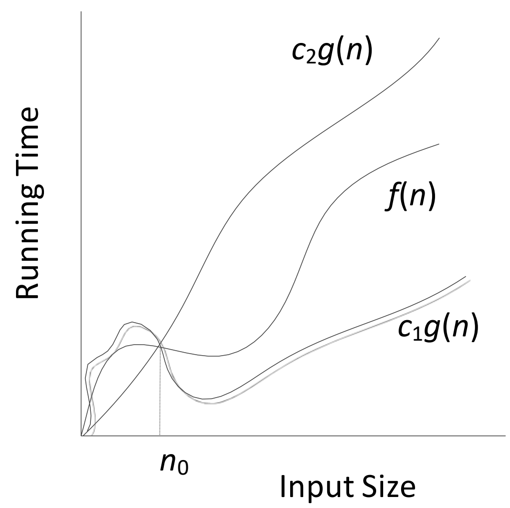
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0\}$



- $f(n) = 3n \log n - 2n \Rightarrow f(n) = \Omega(n \log n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = \Omega(n^3)$

Algorithm Analysis
Rate of Growth

- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0\}$



- $f(n) = 3n \log n + 4n + 5\log n \Rightarrow f(n) = \Theta(n \log n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = \Theta(n^3)$

Algorithm Analysis
Rate of Growth

- Example: Find the largest element

```
public static double arrayMax(double[] data)
{
  int n = data.length;                     // c1
  double currentMax = data[0];             // c2
  for (int j=1; j < n; j++)                // loop executed n – 1 times
      if (data[j] > currentMax)            // loop body takes c3
          currentMax = data[j];            // c4
  return currentMax;
}
```

- Total running time, $f(n) = c1 + c2 + c3(n - 1) + c4$
- $f(n) = O(n)$

Algorithm Analysis
Rate of Growth

- Example: Three-way set disjointness problem (solution 1)

```
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false
  return true;
}
```

- f(n) = $O(n^3)$

Algorithm Analysis
Rate of Growth

- Example: Three-way set disjointness problem (solution 2)

```
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
        if (a == b)
          for (int c : groupC)
            if (b == c)
              return false
  return true;
}
```

- $f(n) = O(n^2)$

Algorithm Analysis
Rate of Growth

- Example: Element uniqueness problem (solution 1)

```
public static boolean unique1(int[] data) {
  int n = data.length;
  for (int j=0; j < n-1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false;                    // found duplicate pair
  return true;                           // if we reach this, elements are unique
}
```

- In the worst case, $f(n) = (n - 1) + (n - 2) + \ldots + 1 = O(n^2)$

Algorithm Analysis

Rate of Growth

- Example: Element uniqueness problem (solution 2)

```
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n);    // make copy of data
  Arrays.sort(temp);                        // and sort the copy, O(n log n)
  for (int j=0; j < n-1; j++)               // for loop takes O(n)
    if (temp[j] == temp[j+1])               // check neighboring entries
      return false;                         // found duplicate pair
  return true;                              // if we reach this, elements are unique
}
```

- In the worst case, $f(n) = O(n \log n) + O(n) = O(n \log n)$

Caveat with Analysis

- O() runtime shows how the runtime of an algorithm changes as the input size gets large
- O() runtime ignores constant terms in the expression
- A large constant term could be more important than the O() runtime if the input size isn't expected to be very large
- Example: 10^7n could be worse than n^2

Proof Techniques

- To disprove a statement, it is sufficient to show a counterexample.
- To prove a statement, we must show it is true for all objects in the domain under consideration.

- Exhaustive proof, direct proof, proof by contraposition, proof by contradiction, mathematical induction

- Loop invariant method: to prove the correctness of an algorithm (or a program), which involves a loop.

Proof Techniques
Proof by Contradiction

- To prove P → Q: Assume Q is false and find a contradiction.
- Example: If an even integer is added to another even integer, the result is an even integer.

- Proof:
  – Let x and y be two even integers. Let $z = x + y$.
  – Let's assume that z is an odd integer (negating the conclusion of the given statement).
  – Since x is even, it can be rewritten as $x = 2n$, for some integer n.
  – Since y is even, it can be rewritten as $y = 2m$, for some integer m.
  – Since z is odd (this we assumed), it can be rewritten as $z = 2k + 1$, for some integer k.

Proof Techniques
Proof by Contradiction

- Proof (continued)
  - Then, we have:
  - $x + y = z$
  - $2n + 2m = 2k + 1$
  - $2n + 2m - 2k = 1$
  - $2(n + m - k) = 1$
  - This is a contradiction because the left hand side is an even number and the right hand side is 1, which is odd.

Proof Techniques
Induction

- Consists of base case (or base step) and inductive step.
- To prove a predicate P(n) is true for all positive integers n.
  - Base case: Show that P(1) is true
  - Inductive step: Assume that P(k) is true, and prove P(k + 1) is also true. This assumption is called *inductive hypothesis*.

- See the example in the next slide.

Proof Techniques
Induction

- Example: Prove that for any positive integer n, $2^n > n$

Base case: $n = 1$: We must show that $2^1 > 1$.

LHS = $2^1 = 2$, RHS = 1. So, LHS > RHS

Induction step: ($n \geq 1$).

Inductive hypothesis: Assume the statement is true for $n = k$,

i.e., $2^k > k$ (for all $k \geq 1$).

We show that it is also true for $n = k + 1$, i.e., $2^{k+1} > k + 1$

LHS = $2^{k+1} = 2 \times 2^k > 2k$ (by the inductive hypothesis)

$2k = k + k \geq k + 1$ = RHS.

So, LHS > RHS.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.