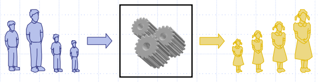
Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Pseudocode (§1.1)

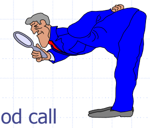
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax ← A[0]
  for i ← 1 to n − 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

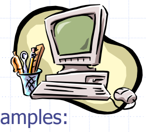
  Algorithm *method* (*arg* [, *arg*…])
  Input …
  Output …
- Method call

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions
  - ← Assignment (like = in Java)
  - = Equality testing (like == in Java)
  - n^2 Superscripts and other mathematical formatting allowed

---

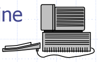
Primitive Operations



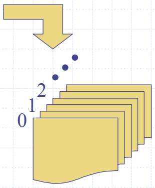
- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Estimating performance

- Count Primitive Operations
- = time needed by RAM model
- Random Access Machine (RAM) Model has:

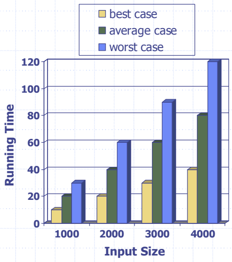


  - A **CPU**
  - An potentially unbounded bank of **memory** cells
  - Each cell can hold an arbitrary number or character
  - Memory cells are numbered
  - Accessing any cell takes unit time



Running Time (§1.1)

- The running time grows with the input size.
- Running time varies with different input
- Worst-case: look at input causing most operations
- Best-case: look at input causing least number of operations
- Average case: between best and worst-case.



Counting Primitive Operations (§1.1)

- Worst-case primitive operations count, as a function of the input size

```
Algorithm arrayMax(A, n)                      # operations
    currentMax ← A[0]                              2
    for i ← 1 to n − 1 do                          1 + n
        if A[i] > currentMax then                  2(n − 1)
            currentMax ← A[i]                      2(n − 1)
    { increment counter i }                        2(n − 1)
    return currentMax                              1
                                        Total      7n − 2
```

Counting Primitive Operations (§1.1)

- Best-case primitive operations count, as a function of the input size

```
Algorithm arrayMax(A, n)                      # operations
    currentMax ← A[0]                              2
    for i ← 1 to n − 1 do                          1 + n
        if A[i] > currentMax then                  2(n − 1)
            currentMax ← A[i]                      0
    { increment counter i }                        2(n − 1)
    return currentMax                              1
                                        Total      5n
```

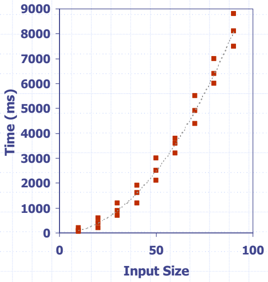
Defining Worst [W(n)], Best [B(N)], and Average [A(n)]

- Let I_n = set of all inputs of size n.
- Let t(i) = # of primitive ops by alg on input i.
- W(n) = maximum t(i) taken over all i in I_n
- B(n) = minimum t(i) taken over all i in I_n
- A(n) = $\sum_{i \in I_n} p(i)t(i)$, p(i) = prob. of i occurring.
- We focus on the worst case
  - Easier to analyze
  - Usually want to know how bad can algorithm be
  - average-case requires knowing probability; often difficult to determine

---

Experimental Studies (§ 1.6)

- Implement your algorithm
- Run your implementation with inputs of varying size and composition
- Measure running time of your implementation (e. g., with System.currentTimeMillis())
- Plot the results



Limitations of Experiments

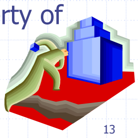
- Implement may be time-consuming and/or difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Infeasible to test for correctness on all possible inputs.

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
- Can prove correctness

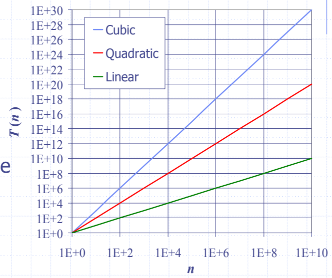
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects ***running time*** by a constant factor;
  - Does not alter its growth rate
- Example: linear growth rate of ***arrayMax*** is an intrinsic property of algorithm.



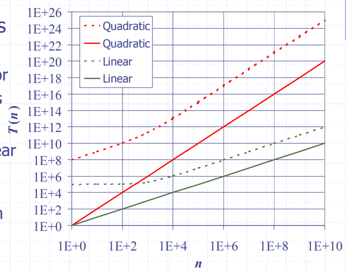
Growth Rates

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- In a log-log chart, the slope of the line corresponds to the growth rate of the function (for polynomials)



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

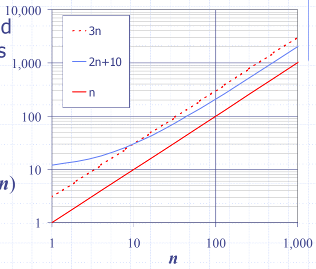
	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Notation (§1.2)

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

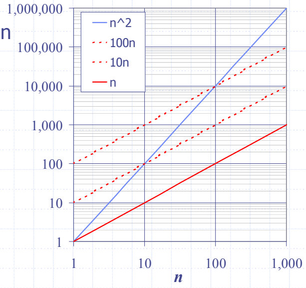
$f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- $7n-2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + \log \log n$

  $3 \log n + \log \log n$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 2$

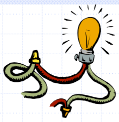
Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- asymptotic analysis = determining an algorithms running time in big-Oh notation
- asymptotic analysis steps:
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $7n - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time" or "runs in order n time"
- Since constant factors and lower-order terms are eventually dropped, we can disregard them when counting primitive operations!

Intuition for Asymptotic Notation



Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

little-oh
- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

little-omega
- f(n) is ω(g(n)) if is asymptotically **strictly greater** than g(n)

Relatives of Big-Oh

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is Θ(g(n)) if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$
- **little-oh**
  - f(n) is o(g(n)) if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$
- **little-omega**
  - f(n) is ω(g(n)) if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

Example Uses of the Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**

  f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\omega(n)$**

  f(n) is ω(g(n)) if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  need $5n_0^2 \geq c \bullet n_0 \rightarrow$ given c, the n_0 that satifies this is $n_0 \geq c/5 \geq 0$

Math you need to know

- Summations (Sec. 1.3.1)
- Logarithms and Exponents (Sec. 1.3.2)
  - **properties of logarithms:**
    $\log_b(xy) = \log_b x + \log_b y$
    $\log_b (x/y) = \log_b x - \log_b y$
    $\log_b xa = a\log_b x$
    $\log_b a = \log_x a/\log_x b$
  - **properties of exponentials**:
    $a^{(b+c)} = a^b a^c$
    $a^{bc} = (a^b)^c$
    $a^b / a^c = a^{(b-c)}$
    $b = a^{\log_a b}$
    $b^c = a^{c*\log_a b}$
- Proof techniques (Sec. 1.3.3)
- Basic probability (Sec. 1.3.4)

Math you need to know

- **Proofs are**
  - a sequence of statements
  - Each statement is true, based on
    - Definitions
    - Hypotheses
    - Well-known math principles
    - Previous statements
  - Statements lead towards conclusion

Induction proof

- Method of proving statements for (infinitely) large values of n, (n is the induction variable).
- Math way of using a loop in a proof.

Example induction proof

- Prove: for all int x, for all int y, for all int n,
  If n is positive, then $x^n – y^n$ is divisible by x-y.
- Let S_n denote "for all x and y, $x^n – y^n$ is divisible by x-y"

Example induction proof

- Prove: for all int x, for all int y, for all int n,
  If n is positive, then $x^n – y^n$ is divisible by x-y.
- Let S_n denote "for all x and y, $x^n – y^n$ is divisible by x-y"
- Proof with induction:
  - Base case: show S_1
  - Inductive Hypothesis (IH): for all k ≥1, if S_k is true, than S_{k+1} is true.
    OR
    Inductive Hypothesis (IH): for all k ≥2, if S_{k-1} is true, than S_k is true.

Example induction proof

- Prove: for all int x, for all int y, for all int n,
  If n is positive, then $x^n – y^n$ is divisible by x-y.
- Let S_n denote "for all x and y, $x^n – y^n$ is divisible by x-y"
- Proof with induction:

More math tools & proofs

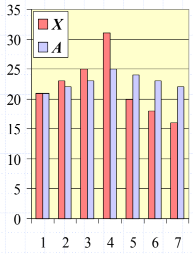
- Correctness of computing average
  - loop invariants and induction
- Recurrence equations
- Strong induction
- Cost of recursive algorithms with recurrence equations.

Computing Prefix Averages

- asymptotic analysis examples: two algorithms for prefix averages
- The ***i***-th prefix average of an array ***X*** is average of the first $(i + 1)$ elements of ***X***:

$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$

- Computing the array ***A*** of prefix averages of another array ***X*** has applications to financial analysis



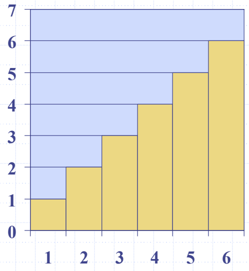
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X     #operations
     A ← new array of n integers                    n
    for i ← 0 to n − 1 do                           n
        s ← X[0]                                    n
        for j ← 1 to i do               1 + 2 + …+ (n − 1)
            s ← s + X[j]                1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                          n
    return A                                        1
```

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first ***n*** integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by computing prefix sums (and averages)

```
Algorithm recPrefixSumAndAverage(X, A, n)
    Input array X of  n ≥ 1 integer.
          Empty array A; A is same size as X.
    Output array A[0]…A[n-1] changed to hold prefix averages of X.
          returns sum of X[0], X[1],…,X[n-1]
    if n=1
        A[0] ← X[0]
        return A[0]
    tot ← recPrefixSumAndAverage(X,A,n-1)
    tot ← tot + X[n-1]
    A[n-1] ← tot / n
    return tot;
```

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by computing prefix sums (and averages)

```
Algorithm recPrefixSumAndAverage(X, A, n)            T(n) operations
    Input array X of  n ≥ 1 integer.
          Empty array A; A is same size as X.
    Output array A[0]…A[n-1] changed to hold prefix averages of X.
          returns sum of X[0], X[1],…,X[n-1]          #operations
    if n=1                                                 1
        A[0] ← X[0]                                        3
        return A[0]                                        2
    tot ← recPrefixSumAndAverage(X,A,n-1)               3+T(n-1)
    tot ← tot + X[n-1]                                     4
    A[n-1] ← tot / n                                       4
    return tot;                                            1
```

Prefix Averages, Linear

- Recurrence equation
  - $T(1) = 6$
  - $T(n) = 13 + T(n-1)$ for $n>1$.
- Solution of recurrence is
  - $T(n) = 13(n-1) + 6$
- $T(n)$ is $O(n)$.