Algorithm Analysis

What is an Algorithm?

- An algorithm is a clearly specified set of instructions to be followed to solve a problem
  - Solves a problem but requires a year is hardly of any use
  - Requires several terabytes of main memory is not useful on most machines
- Problem
  - Specifies the desired input-output relationship
- Correct algorithm
  - Produces the correct output for every possible input in finite time (i.e., must terminate eventually)
  - Solves the problem (i.e., must be correct)
- Why bother analyzing algorithm or code; isn't getting it to work enough?
  - Estimate time and memory in the average case and worst case
  - Identify bottlenecks, i.e., where to reduce time and space
  - Speed up critical algorithms or make them more efficient

Algorithm Analysis

- Predict resource utilization of an algorithm
  - Running time
  - Memory usage
- Dependent on architecture: Serial, Parallel, Quantum, Molecular, ...
- Our main focus is on running time
  - Memory/time tradeoff
  - Memory is cheap
- Our assumption: simple serial computing model

What to Analyze (cont'd)

- Let T(N) be the running time, where N (sometimes n) is typically the size of the input
  - Linear or binary search?
  - Sorting?
  - Multiplying two integers?
  - Multiplying two matrices?
  - Traversing a graph?
- T(N) measures number of primitive operations performed
  - E.g., addition, multiplication, comparison, assignment
- Running Time Calculations
  - The declarations count for no time
  - Simple operations (e.g. +, *, <=, =) count for one unit each
  - Return statement counts for one unit

Example

```
                                              Cost
int sum (int n)                               0
{
     int partialSum;                          0

1.    partialSum = 0;                         1
2.   for (int i = 1; i <= n; i++)             1+(N+1)+N
3.          partialSum += i * i * i;          N*(1+1+2)
4.    return partialSum;                      1
}
```

$T(N) = 6N+4$

- $T(N) = 6N+4$

Running Time Calculations

General rules

- Rule 1 - Loops
  - The running time of a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations of the loop
- Rule 2 - Nested loops
  - Analyze these inside out
  - The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops

    $\Downarrow$

    Number of iterations

Running Time Calculations (cont'd)

Rule 1 - Loops

```
                                             # of operations
for (int i = 0; i < N; i++)                  ? (1 + N + N)
  sum += i * i;                              ?  3*N
                                             T(N) = ?
```

Rule 2 - Nested loops

```
                                             # of operations
for (int i = 0; i < n; i++)                  ? (1 + N + N)
  for (int j = 0; j < n; j++)                ? N*(1 + N + N)
    sum++;                                   ? N * N
                                             T(N) = ?
```

Running Time Calculations (cont'd)

General rules

- Rule 3 - Consecutive statements
  - These just add
  - Only the maximum is the one that counts
- Rule 4 - Conditional statements (e.g. if/else)
  - The running time of a conditional statement is never more than the running time of the test plus the largest of the running times of the various blocks of conditionally executed statements
- Rule 5 - Function calls
  - These must be analyzed first

Rule 3 - Consecutive statements

```
                                          # of operations
for (int i = 0; i < n; i++)               ?
  a[i] = 0;                               ?
for (int i = 0; i < n; i++)               ?
  for (int j = 0; j < n; j++)             ?
    a[i] += a[j] + i * j;                 ?
                                          T(n) = ?
```

Rule 4 - Conditional statements

```
                                          # of operations
if (a > b && c < d) {                     ?
  for (int j = 0; j < n; j++)             ?
    a[i] += j;                            ?
}
else {
  for (int j = 0; j < n; j++)             ?
    for (int k = 1; k <= n; k++)          ?
      a[i] += j * k;                      ?
}
                                          T(n) = ?
```

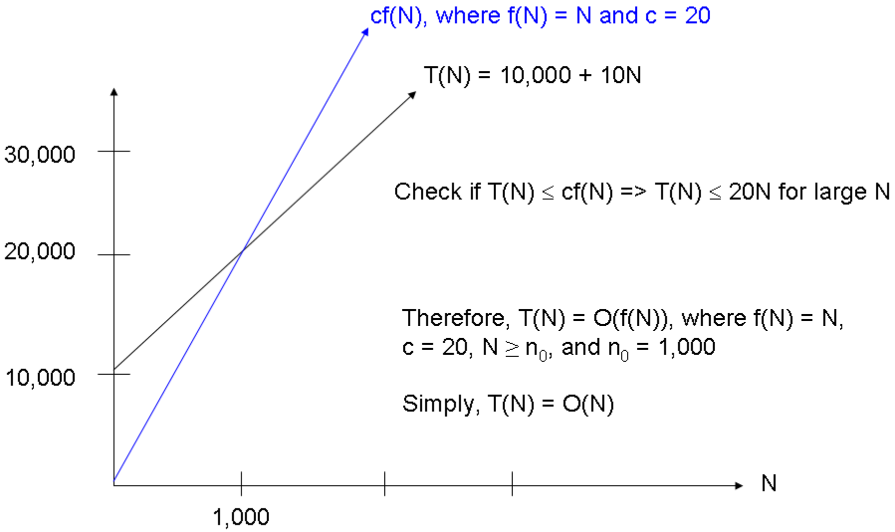
Average and Worst-Case Running Times

- Estimating the resource use of an algorithm is generally a theoretical framework and therefore a formal framework is required
- Define some mathematical definitions
- Average-case running time $T_{avg}(N)$
- Worst-case running time $T_{worst}(N)$
- $T_{avg}(N) \leq T_{worst}(N)$
- Average-case performance often reflects typical behavior of an algorithm
- Worst-case performance represents a guarantee for performance on any possible input
- Typically, we analyze worst-case performance
  - Worst-case provides a guaranteed upper bound for all input
  - Average-case is usually much more difficult to compute
  - Best-case ?

Asymptotic Analysis of Algorithms

- We are mostly interested in the performance or behavior of algorithms for very large input (i.e., as $N \to \infty$)
  - For example, let $T(N) = 10,000 + 10N$ be the running time of an algorithm that processes N transactions
  - As N grows large ($N \to \infty$), the term $10N$ will dominate
  - Therefore, the smaller looking term $10N$ is more important if N is large
- Asymptotic efficiency of the algorithms
  - How the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound

Asymptotic Analysis of Algorithms (cont'd)



Asymptotic Analysis of Algorithms (cont'd)

- Asymptotic behavior of $T(N)$ as N gets big
- Exact expression for $T(N)$ is meaningless and hard to compare
- Usually expressed as fastest growing term in $T(N)$, dropping constant coefficients
  - For example, $T(N) = 3N^2 + 5N + 1$
  - Therefore, the term N^2 describes the behavior of $T(N)$ as N gets big

Asymptotic Analysis of Algorithms (cont'd)

- Let $T(N)$ be the running time of an algorithm
- Let $f(N)$ be another function (preferably simple) that we will use as a bound for T(N)
- Asymptotic notations
  - Big-Oh notation $O()$
  - Big-Omega notation $\Omega()$
  - Big-Theta notation $\Theta()$
  - Little-oh notation $o()$
  - Little-omega notation $\omega()$

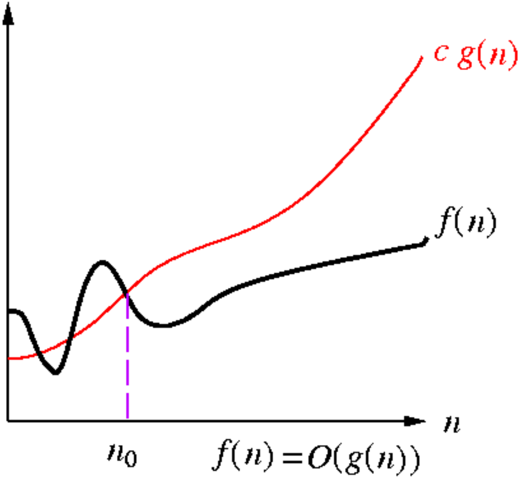
Big-Oh Notation

- $O(f(N))$ is the SET of ALL functions $T(N)$ that satisfy: There exist positive constants c and n_0 such that, for all $N \geq n_0$, $T(N) \leq cf(N)$
- Examples
  - $1,000,000N = O(N)$
    - Proof: Choose $c = 1,000,000$ and $n_0 = 1$
  - $N = O(N^3)$
    - Proof: Choose $c = 1$ and $n_0 = 1$
  - $N^3 + N^2 + N = O(N^3)$
    - Proof: Choose $c = 3$ and $n_0 = 1$

NOTE:

- Thus, big-oh notation doesn't care about (most) constant factors. It is unnecessary to write $O(2N)$. We can just simply write $O(N)$
- Big-Oh is an upper bound

Big-Oh Notation (cont'd)

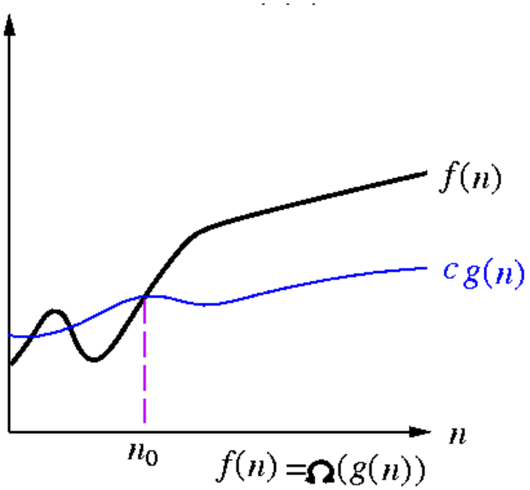


- $g(N)$ is asymptotically upper bounded by $f(N)$

Big-Omega Notation

- Definition: $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$
- Asymptotic lower bound
- The growth rate of $T(N)$ is $\geq$ that of $g(N)$
- Examples
  - $N^3 = \Omega(N^2)$ (Proof: $c = ?, n_0 = ?$)
  - $N^3 = \Omega(N)$ (Proof: $c = 1, n_0 = 1$)

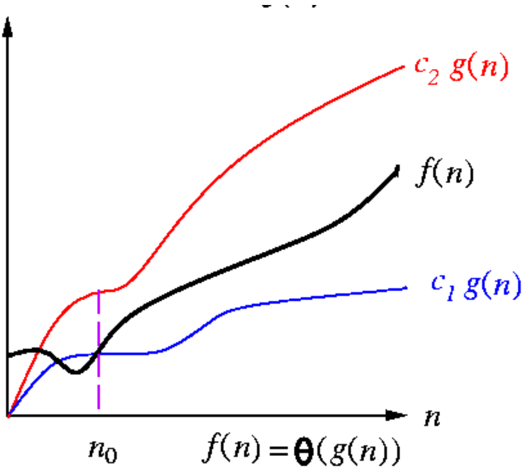
Big-Omega Notation (cont'd)



- $g(N)$ is asymptotically lower bounded by $f(N)$

Big-Theta Notation

- Definition: $T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$
- Asymptotic tight bound
- The growth rate of $T(N)$ equals the growth rate of $h(N)$
- Examples
  - $2N^2 = \Theta(N^2)$
  - Suppose $T(N) = 2N^2$ then $T(N) = O(N^4)$; $T(N) = O(N^3)$; $T(N) = O(N^2)$ all are technically correct, but last one is the best answer. Now writing $T(N) = \Theta(N^2)$ says not only that $T(N) = O(N^2)$, but also the result is as good (tight) as possible

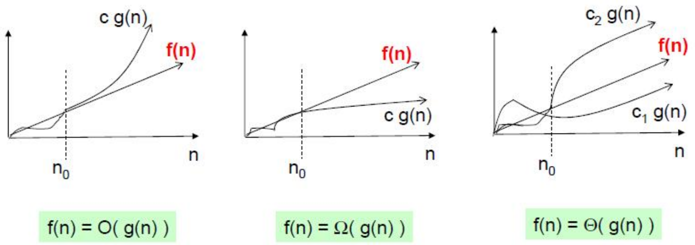


- $g(N)$ is asymptotically equal to $f(N)$

Little-oh Notation

- Definition: $T(N) = o(g(N))$ if for all constants c there exists an n_0 such that $T(N) < cg(N)$ when $N > n_0$
  - That is, $T(N) = o(g(N))$ if $T(N) = O(g(N))$ and $T(N) \neq \Theta(g(N))$
  - The growth rate of $T(N)$ less than ($<$) the growth rate of $g(N)$
  - Denote an upper bound that is not asymptotically tight
- The definition of O-notation and o-notation are similar
  - The main difference is that in $T(N) = O(g(N))$, the bound $0 \leq T(N) \leq cg(N)$ holds for some constant $c > 0$, but in $T(N) = o(g(N))$, the bound $0 \leq T(N) < cg(N)$ holds for all constants $c > 0$
  - For example , $N = o(N^2)$, but $2N^2 \neq o(N^2)$

The $O()$, $\Omega()$, $\Theta()$ Notations



- O-notation gives an upper bound for a function to within a constant factor
- Ω-notation gives an lower bound for a function to within a constant factor
- Θ-notation bounds a function to within a constant factor
  - The value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive

Some Rules

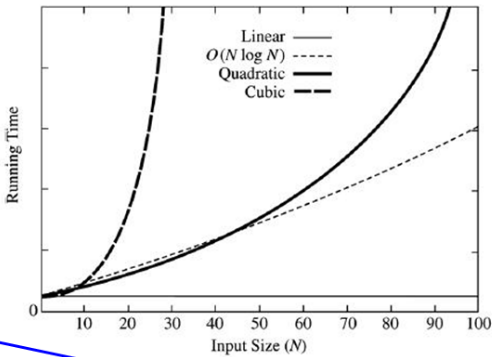
- Rule 1: If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = O(f(N) + g(N))$ less formally it is $max\{O(f(N)), O(g(N))\}$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
- Rule 2:If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$
- Rule 3: $log^k N = O(N)$ for any constant k
- Rule 4: $log_a N = \Theta(log_b N)$ for any constants a and b

Rate of Growth

- Rate of Growth

Considered "efficient"

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential



Considered "useless"

Maximum Subsequence Sum Problem

- Given (possibly negative) integers $A_1, A_2, \cdots, A_N$, find the maximum value (≥ 0) of:

$$\sum_{k=i}^{j} A_k$$

- We don't need the actual sequence (i, j), just the sum
- If the final sum is negative, the maximum sum is 0
- E.g. $< 1, -4, \underline{4, 2, -3, 5, 8}, -2 >$, the MaxSubSum is 16.

Solution 1 of MaxSubSum

- Idea: Compute the sum for all possible subsequence ranges (i, j) and pick the maximum sum

```
MaxSubSum1(A)
  maxSum = 0
  for i = 1 to N
    for j = i to N
      sum = 0
      for k = i to j
        sum = sum + A[k]
      if (sum > maxSum)
        maxSum = sum
  return maxSum
```

All possible starting point — O(N)

All possible ending point — O(N)

Calculate sum for range (i, j) — O(N)

$T(N) = O(N^3)$

Solution 2 of MaxSubSum

- Observation

$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

- So, we can re-use the sum from previous range

```
MaxSubSum2(A)
  maxSum = 0
  for i = 1 to N
    sum = 0
    for j = i to N
      sum = sum + A[j]
      if (sum > maxSum)
        maxSum = sum
  return maxSum
```

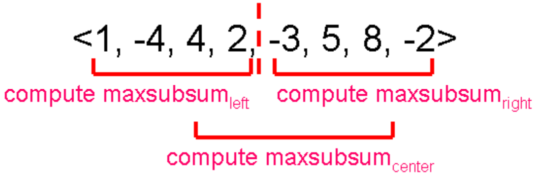
O(N)

O(N)

$T(N) = O(N^2)$

Solution 3 of MaxSubSum

- Idea: Recursive, divide and conquer
  - Divide sequence in half: $A_{1..center}$ and $A_{(center+1)..N}$
  - Recursively compute MaxSubSum of left half
  - Recursively compute MaxSubSum of right half
  - Compute MaxSubSum of sequence constrained to use A_{center} and $A_{(center+1)}$
  - Example



Solution 3 of MaxSubSum (cont'd)

- 4, -3, 5, -2 || -1, 2, 6,-4, where || marks the half-way point
  - The maximum subsequence sum of the left half is 6: 4 + -3 + 5.
  - The maximum subsequence sum of the right half is 8: 2 + 6.
  - The maximum subsequence sum of sequences having -2 as the right edge is 4: 4 + -3 + 5 + -2; and the maximum subsequence sum of sequences having -1 as the left edge is 7: -1 + 2 + 6.
  - Comparing 6, 8 and 11 (4 + 7), the maximum subsequence sum is 11 where the subsequence spans both halves: 4 + -3 + 5 + -2 + -1 + 2 + 6.

Solution 3 of MaxSubSum (cont'd)

```
MaxSubSum3(A, i, j)
  maxSum = 0
  if (i == j)
    if (A[i] > 0)
      maxSum = A[i]
  else
    k = floor((i + j) / 2)
    maxSumLeft = MaxSubSum3(A, i, k)
    maxSumRight = MaxSubSum4(A, k + 1, j)
    compute maxSumThruCenter
    maxSum = Maximum(maxSumLeft, maxSumRight, maxSumThruCenter)
  return maxSum
```

Analysis:

- $T(1) = O(1), \quad T(N) = 2T(N/2) + O(N)$
- $T(N) = O(N \log_2 N)$ – will be derived later in the class

Observations

- Any negative subsequence cannot be a prefix to the maximum subsequence
- Or, only a positive, contiguous subsequence is worth adding
- Example: $< 1, -4, 4, 2, -3, 5, 8, -2 >$

```
MaxSubSum4(A)
  maxSum = 0
  sum = 0
  for j = 1 to N
    sum = sum + A[j]
    if (sum > maxSum)
      maxSum = sum
    else if (sum < 0)
      sum = 0
  return maxSum
```

MaxSubSum Running Times

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
N = 10	0.000009	0.000004	0.000006	0.000003
N = 100	0.002580	0.000109	0.000045	0.000006
N = 1,000	2.281013	0.010203	0.000485	0.000031
N = 10,000	NA	1.2329	0.005712	0.000317
N = 100,000	NA	135	0.064618	0.003206

38 min

26 days

Time in seconds.
Does not include time to read array.

Logarithmic Behavior

- $T(N) = O(log_2 N)$
- An algorithm is $O(log_2 N)$ if it takes constant $O(1)$ time to cut the problem size by a fraction (which is usually 1/2)
- Usually occurs when
  - Problem can be halved in constant time
  - Solutions to sub-problems combined in constant time
- Examples
  - Binary search
  - Euclid's algorithm
  - Exponentiation

Euclid's Algorithm

Compute the greatest common divisor $gcd(M, N)$ between the integers M and N; e.g., $gcd(50, 15) = 5$

```
long gcd( long m, long n )
{
    while( n != 0 )
    {
        long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

Example: gcd(3360,225)
- m = 3360, n = 225
- m = 225, n = 210
- m = 210, n = 15
- m = 15, n = 0

Euclid's Algorithm

- Estimating the running time: how long the sequence of remainders is?
  - $logN$ is a good answer, but value of the remainder does not decrease by a constant factor
  - Indeed the remainder does not decrease by a constant factor in one iteration, however we can prove that after two iterations the remainder is at most half of its original value
    - Theorem 2.1: If $M > N$, then $M \ mod \ N < M/2$
  - Number of iterations is at most $2logN = O(logN)$
- $T(N) = 2log_2 N = O(log_2 N)$; T(225) = 16
- Better worst-case: $T(N) = 1.44 log_2 N$; T(225) = 11
- Average-case: $T(N) = (12 \ln 2 \ln N)/\pi^2 + 1.47$; T(225) = 6

Exponentiation

- Compute $X^N = \overbrace{X * X * \cdots * X}^{N}$, for integer $N \geq 0$
- Obvious algorithm: To compute X^N uses $(N-1)$ multiplications
- Observations
  - A recursive algorithm can do better
  - $N \leq 1$ is the base case
  - $X^N = X^{N/2} * X^{N/2}$ (for even N)
  - $X^N = X^{(N-1)/2} * X^{(N-1)/2} * X$ (for odd N)
- E.g., $X^3 = (X^2) \cdot X; \quad X^7 = (X^3)^2 \cdot X; \quad X^{15} = (X^7)^2 \cdot X;$ $X^{31} = (X^{15})^2 \cdot X; \quad X^{62} = (X^{31})^2$
- Minimize number of multiplications
- $T(N) = 2log_2N = O(log_2N)$

Complexity of an Algorithm

- Best case analysis: too optimistic, not really useful.
- Worst case analysis: usually only yield a rough upper bound.
- Average case analysis: a probability distribution of input is assumed, and the average of the cost of all possible input patterns are calculated. However, it is usually difficult than worst case analysis and does not reflect the behavior of some specific data patterns.
- Amortized analysis: this is similar to average case analysis except that no probability distribution is assumed and it is applicable to any input pattern (worst case result).
- Competitive analysis: Used to measure the performance of an on-line algorithm w.r.t. an adversary or an optimal off-line algorithm.

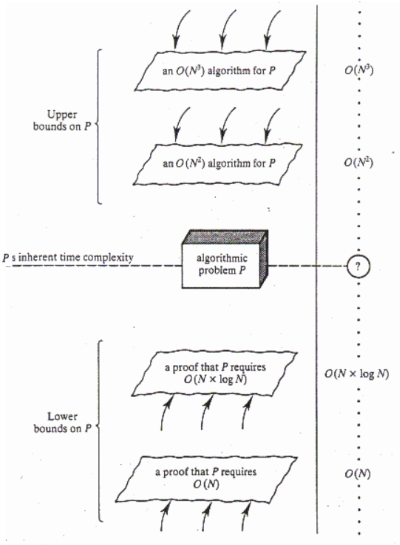
Amortized analysis

- Given a stack S with 2 operations: $push(S, x)$, and $multipop(S, k)$, the cost of the two operations are 1 and $min(k, |S|)$ respectively. What is the cost of a sequence of n operations on an initially empty stack S?
  - Best case: n, 1 for each operation.
  - Worst case: $O(n^2)$, $O(n)$ for each operation.
  - Average case: complicate and difficult to analyze.
  - Amortized analysis: $2n$, 2 for each operation. (There are at most n push operations and hence at most n items popped out of the stack.)

The Difficulty of a Problem

- Upper bound $O(f(n))$ means that for sufficiently large inputs, running time $T(n)$ is bounded by a multiple of $f(n)$
- Existing algorithms (upper bounds).
- Lower bound $\Omega(f(n))$ means that for sufficiently large n, there is at least one input of size n such that running time is at least a fraction of $f(n)$ for any algorithm that solves the problem.
- The inherent difficulty $\Rightarrow$ lower bound of algorithms
- The lower bound of a method to solve a problem is not necessary the lower bound of the problem.

The Difficulty of a Problem (cont'd)



Examples

- Sorting n elements into ascending order.
  - $O(n^2), O(n \log n)$, etc. – Upper bounds.
  - $O(n), O(n \log n)$, etc. – Lower bounds.
  - Lower bound matches upper bound.
- Lower bound matches upper bound.
- Multiplication of 2 matrices of size n by n.
  - Straightforward algorithm: $O(n^3)$.
  - Strassen's algorithm: $O(n^{2.81})$.
  - Best known sequential algorithm: $O(n^{2.376})$.
  - Best known lower bound: $\Omega(n^2)$.
  - The best algorithm for this problem is still open.

Complexity of Algorithms and Problems

- Notations

Symbol	Meaning
P	a problem
I	a problem instance
I_n	the set of all problem instances of size n
A	an algorithm for P
A_P	the set of algorithms for problem P
$Pr(I)$	probability of instance I
$C_A(I)$	cost of A with input I
R_A	the set of all possible versions of a randomized algorithm A

Formal Definitions

worst cost (A)	$\max_{I \in I_n} C_A(I)$
best cost (A)	$\min_{I \in I_n} C_A(I)$
average cost (A)	$\sum_{I \in I_n} \Pr(I) \cdot C_A(I)$
expected cost (R_A, I)	$\sum_{A \in R_A} \Pr(A) \cdot C_A(I)$
(worst case) complexity (P)	$\min_{A \in A_P} \left\{ \max_{I \in I_n} C_A(I) \right\}$
average case complexity (P)	$\min_{A \in A_P} \left\{ \sum_{I \in I_n} \Pr(I) \cdot C_A(I) \right\}$
worst expected complexity (R_A)	$\max_{I \in I_n} \left\{ \sum_{A \in R_A} \Pr(A) \cdot C_A(I) \right\}$
average expected complexity (R_A)	$\sum_{I \in I_n} \Pr(I) \cdot \left\{ \sum_{A \in R_A} \Pr(A) \cdot C_A(I) \right\}$