Chapter 19: Randomized and Approximation algorithms
objectives

- Principles of randomized algorithms
- Generation of pseudo-random numbers
- Principle and design of randomized algorithms
- Randomized algorithms for hiring problem, primality testing, comparison of strings and randomized quicksort
- Principles of approximation algorithms
- Design of approximation algorithms using Heuristics
- Role of greedy, linear programming, and dynamic programming in approximation algorithms
What is a randomized algorithm?

Randomized algorithms, also called probabilistic algorithms, use the concept of randomness as a computing tool for algorithm design. Let us take the example of a maze problem. A deterministic algorithm (e.g. DFS or BFS) would construct a graph of all paths of the maze and exhaustively try and select a path. A randomized maze algorithm would, instead, take a random decision, which may be based on the output of tossing a coin at every instance, for selecting the next path. The resulting path is called a random walk. Thus, randomized algorithms take a radical approach compared to deterministic algorithms.
Advantages of randomized algorithms

Some of the advantages of randomized algorithms are as follows:

1. Randomized algorithms are known for their simplicity. Any deterministic algorithm can easily be converted to a randomized algorithm. These algorithms are very simple to understand and implement.

2. Randomized algorithms are very efficient. They utilize little execution time and space compared to any deterministic algorithms.

3. Randomized algorithms exhibit superior asymptotic bounds compared to deterministic algorithms. In other words, the algorithm complexity of randomized algorithms is better than that of most of the deterministic algorithms.
Disadvantages of randomized algorithms

1. Reliability is an important issue in many critical applications, as not all randomized algorithms give correct answers always. In addition, many randomized algorithms may not terminate. Hence, reliability is an important concern that needs to be dealt with.

2. The quality of randomized algorithms is dependent on the quality of the random number generator used as part of the algorithm.
## Differences

**Table 19.1** Difference between randomized and deterministic algorithms

<table>
<thead>
<tr>
<th>Randomized algorithms</th>
<th>Deterministic algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known for its simplicity</td>
<td>Complex compared to randomized algorithms</td>
</tr>
<tr>
<td>Very efficient</td>
<td>Less efficient compared to randomized algorithms</td>
</tr>
<tr>
<td>Different results in every run even for a fixed input</td>
<td>No variation in outputs for the same input</td>
</tr>
<tr>
<td>Associated with a probability of error</td>
<td>Deterministic outputs</td>
</tr>
</tbody>
</table>
Principles of randomized algorithms

Concept of witness  This principle involves the question of checking whether a given input possesses a property $X$ or not. It is established by finding a certain object called a witness or a certificate. The witness is identified to prove the fact that the input indeed has the desired property $X$. By conducting a few trials, it can be found out whether the property was really present. The presence of a witness is a strong proof of the property $X$. Otherwise one should conclude that the input does not have such a property $X$ based on the absence of witnesses. This principle is illustrated using the example of primality testing discussed in Section 19.3.2.

Fingerprinting  By definition, a fingerprint is a shorter message that is representative of a larger object. Fingerprinting is a technique wherein we get a comparison of two large objects $A$ and $B$ only by comparing their respective short fingerprints. If two fingerprints do not match, then the objects $A$ and $B$ are different. However, if the fingerprints match, then there is a strong circumstantial evidence that both objects are the same. This principle is illustrated in Section 19.3.3 with the help of an example comparing lengthy strings.
Principles

**Checking identities**  Let us assume that an algebraic expression is given, and the problem is to check whether the expression evaluates to zero or not. The principle of checking identities is to plug the random variables of a given algebraic equation and check whether the expression evaluates to zero. If it is not zero, then the given expression is not an identity. Otherwise, there is a strong circumstantial evidence that the expression is identically zero.

**Random sampling and ordering**  The performance of an algorithm sometimes improves by randomizing the input distribution or order. It can be observed that for certain ordering of the inputs the performance of the algorithm can be higher or just acceptable. Here, randomization leads to randomized ordering, partitioning, and sampling. In addition, randomized algorithms gather information about input distributions using random samples. This is illustrated through the hiring problem discussed in Section 19.3.1.
Principles

**Foiling the adversary** A randomized algorithm can be viewed as a game between a person and an adversary, that is, a person proposing an algorithm and an adversary who tries to foil the algorithm by designing suitable inputs so that the algorithm takes a larger time. In other words, a randomized algorithm can be viewed as a selection of an algorithm from a large set of deterministic algorithms, and this selection can be considered a scenario where things are made difficult by giving a random input, thus making the task more difficult.
Simulating random components in a deterministic algorithm requires a random number generator. A random number generator is a software component that is used to generate a random number often in the range \([0 \ldots 1]\). The generated random numbers often have uniform distribution and hence are independent of each other.

There are different types of random numbers such as true random number and pseudo-random numbers.
Properties of good random number generator

Some of the desirable properties of a ‘good random number generator’ are as follows:

**Efficiency**  A random number generator should be able to generate many numbers in a limited time.

**Deterministic property**  The generated number should be reproducible at a later time, provided the initial seed is given.

**Uniformity**  The generated random number should pass the statistical test of uniformity by adhering to uniform distribution.
Properties

*Independence*  The generated random number should pass the test of statistical independence. In other words, the generated numbers should have good numeric statistical distribution and do not have any relationship among themselves.

*Long cycle*  Generated random numbers should not be repeated. This implies that the repetition should happen after a very long time.
Random number generator

The simplest way of generating random numbers is through linear congruential generators (LCGs). LCGs are functions or methods that use a recursive relation to generate random numbers. LCG are pseudo-random number generators. Let \( X_0 \) be the initial seed or input. Then \( X_{i+1} \) is given recursively as follows:

\[
X_{i+1} = (a \times X_i + b) \mod m \quad \text{for} \quad i = 0, 1, 2, \ldots
\]

Here, \( a, b, \) and \( m \) are constants. The variable \( a \) is a constant multiplier, \( b \) is increment, and \( m \) is modulus. This generates number in the range \( \{0 \ldots m-1\} \). If \( b > 0 \), then the range would be \( \{1 \ldots m\} \) and if \( b = 0 \), the generated numbers would be \( \{1 \ldots m\} \). Here, \( m \) is used to scale down the numbers using modulo arithmetic. The following are some of the examples of LCGs:

1. Matlab software uses a \texttt{rand()} function that is defined as follows:

\[
X_{i+1} = 16807 \times X_i \mod (2^{31} - 1)
\]

2. Another function \texttt{RAND} generator uses an LCG, as follows:

\[
X_{i+1} = (69069 \times X_i + 1) \mod (2^{32})
\]
Polynomial algorithms  Let there be a polynomial $p(n)$ of degree $n$. An algorithm is called a polynomial algorithm if the upper bound of the algorithm is $O(p(n))$. Examples of polynomial algorithms are searching, sorting algorithms, etc. Even though polynomial algorithms having complexity $n^{100}$ are complex as exponential algorithms, in reality, one would never encounter algorithms having such a large polynomial degree. Jack Edmonds (refer to Box 18.1) claimed in 1965 that polynomial algorithms are easily solvable because of the following reasons:

1. All machines are inherently polynomial—only the constants of the polynomials vary from machine to machine.
2. Operations of polynomials are closed under addition and composition. That is, an algorithm that includes two polynomial algorithms $A$ and $B$ calling each other as a subroutine or is sequentially followed by another algorithm is itself a polynomial algorithm.
3. Polynomial algorithms take finite time and use finite resources.

In general, polynomial algorithms are desirable algorithms for any given task.
Example 19.1  Let us assume that the initial seed $X_0$ is 7, $a = 3$, $b = 10$, and $m = 100$. Assume an LCG and generate a random number sequence for these conditions.

Solution  Let us assume that the LCG is as follows:

$$X_{i+1} = (a \times X_i + b) \mod m$$

for $i = 0, 1, 2, \ldots$

Hence, by substituting the given values, one gets an LCG as $X_{i+1} = (7X_i + 10) \mod m$.

The initial seed is given as 7, that is, $X_0 = 7$. The generated number $R_0$ is $7/100 = 0.07$. This ensures that the output is in the range $[0 \ldots 1]$. Now the next number can be computed by substituting all these values in the preceding function.

$$X_1 = (3 \times 7 + 10) \mod 100$$

$$= 31 \mod 100 = 31$$

Thus, the generated random number $R_1$ would be $31/100 = 0.31$. To get the next number, substitute this again in the following recursive relation:

$$X_2 = (7 \times 31 + 10) \mod 100$$

$$= 227 \mod 100 = 27$$

The generated $R_2$ would be $27/100 = 0.27$. 


Thus, the random numbers generated are $R_1 = 0.31$ and $R_2 = 0.27$. This process is continued till all the numbers are generated as per program requirements.
Informal algorithm

1. Let the array A be of size ‘n’. Initialize, Index = 1,
2. Generate a random number using LCG
4. Index = Index + 1
5. Repeat steps 2-4 till index = n
Algorithm random array(A)

%% Input: Array A with all elements initialized to zero
%% Output: A random array
Begin
    for index = 1 to N
        k = generate random number in the range [0 .. 1]
        Exchange (A[index], A[k])
    End for
End
Types of randomized algorithm

A Las Vegas algorithm either will terminate when the correct answer of probability $\geq \frac{1}{2}$ or will not give any output. A Las Vegas algorithm may terminate and give correct answer or may not terminate at all. But if the algorithm produces an answer, it will be always correct.
Hiring Problem

Hiring problem is a problem of hiring a secretary among a group of potential secretaries using deterministic and randomized approaches. Let us illustrate hiring problem now.
Informal algorithm

Step 1: Store ‘n’ candidates as an array having Id and rank. Rank indicates skills.
Step 2: Assign current_best candidate as 0. Here zero is a dummy candidate.
Step 3: Conduct interview for all persons indexed by index 1 to n
   3.1: Interview candidate A[index]
   3.2: If candidate is better than current_best candidate in skill, hire him/her
   3.3: set index = index +1
Step 4: Return best candidate and exit
Algorithm deterministic_hiring (A, n)

% Input: Array A that has ID and rank of all secretaries who attend interview
% Output: Selected best secretary

Begin
    current_best = A[0]  % ID same as index of array
    while index ≤ n do
        interview A[index]
        if current_best has rank better than A[index] then
            do nothing
        else
            hire candidate A[index]
        end if
        index = index + 1
    end while
Return (current_best)
End
Informal algorithm

1. Store ‘n’ candidates as an array having Id and rank. Rank indicates skills.
2. Assign current best candidate as 0. Here zero is a dummy candidate.
3. Randomly permute the given array A.
4. Conduct interview for all persons indexed by index 1 to n
   4.1 Interview candidate A[index]
   4.2 If candidate is better than current best candidate, hire
   4.3 set index = index +1
5. Return best candidate and exit
Algorithm random_hire (A, n)

% Input: Array A of 'n' secretaries
% Output: selected secretary
Begin
    randomly permute array A[n]
    current_best = A[0]    % ID same as index
    while index ≤ n do
        if current_best has better rank than A[index] then
            do nothing
        else
            hire candidate A[index]
            current_best = A[index]
        end if
        index = index + 1
    end while
Return (current_best)
End
Complexity analysis

*Complexity Analysis*

To analyse this random algorithm, a random variable is declared. Let $X_i$ be the candidate $i$ being hired and is defined as follows:

$$x_i = \begin{cases} 
1 & \text{if candidate } i \text{ is hired} \\
0 & \text{if candidate } i \text{ is not hired}
\end{cases}$$

Therefore, $x_i$ is an indicator random variable. This concept is discussed in Appendix A.6. The expectation $E(X)$ is given as follows:

$$E(X) = \sum_{i=1}^{n} x \cdot \Pr[X = x]$$

Since all candidates have equal chance to get selected, the probability is equally likely:

$$\therefore E(X_i) = \frac{1}{i}$$
Complexity analysis

Hence,

\[ E(X) = E \left( \sum_{i=1}^{n} X_i \right) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) \]

Thus, for all the candidates, this would be as follows:

\[ E[\text{cost}] = \sum_{i=1}^{n} E[\text{fee}] = \sum_{i=1}^{n} R \times E[\text{fee}] = R \ln n \]

Thus, the cost is better than \( O(C_{\text{hire}}n) \).
Fermat algorithm

Step 1: Read number $p$ that needs to be checked for primality.
Step 2: Pick witness $a$ uniformly at random such that $1 < a \leq p - 1$
Step 3: If $a^{p-1} \equiv 1 \pmod{p}$ then do the following:
   Return $p$ is a prime number.
   Otherwise, return $p$ is probably a composite number.
Step 4: End.
**Algorithm FermatPrime(p)**

**Input:** A number $p$ that needs to be tested for primeness  
**Output:** Status of $p$ as prime or not

**Begin**

Pick a 'witness' integer $a$ uniformly at random such that $a \in \{2, 3, \ldots, p - 1\}$

if $a^{p-1} \equiv 1 \pmod{p}$ then  
    Return (p is probably prime)  
else  
    Return (p is not a prime)

**End if**

**End**
Complexity analysis

What is the complexity analysis of Fermat algorithm? It can be observed that for determining whether a number $p$ is prime or not, one has to find $a^{p-1}$, where ‘$a$’ is picked randomly in the range 2 to $p-1$ (inclusive). This requires repeated squaring, multiplication and modulo $p$ requires $O(\log n)$ steps, where $n$ is the number of bits of $p$. The witness ‘$a$’ can be chosen ‘$k$’ times. Put together, the complexity of the algorithm is $O(k \times \log^3 n)$. Hence, Fermat primality testing is $O(\log^3 n)$ as $k$ is a constant.
Randomized algorithms are used to reduce the communicational cost. How? Let us define communication cost first. Communication cost is the number of bits that are transmitted across the channel. A deterministic algorithm uses $n$ bits, and hence it is not effective. A randomized algorithm, on the other hand, uses the idea of ‘fingerprinting’. A fingerprint is a ‘shorter version’ of a very large string. A typical fingerprint consists of $O(\log n)$ bits. The smaller size of the fingerprint considerably reduces the complexity of checking larger strings (as discussed
Informal algorithm

The algorithm can informally be given as follows:

At the source side, \( A \) sends the message \( a' \), as follows:

**Step 1:** Convert \( a \) to the set of binary string \( a \in \{0, 1, 2, \ldots, 2^n - 1\} \)

**Step 2:** Randomly pick a prime number \( p \leq k \) uniformly; \( k \) is the upper bound of prime number \( p \).

**Step 3:** Find the fingerprint \( fp(a) \) of the string \( a \) using the following formula:

\[
fp(a) = a \mod p,
\]

**Step 4:** Send \( (fp(a), i) \) across the channel to destination.

At the receiver side, \( B \) receives the fingerprint and checks, as follows:

**Step 1:** Use \( p \), and compute \( fp(b) = b \mod p \).

**Step 2:** If \( fp(a) \neq fp(b) \) then
- Output ‘strings are not equal’
else
- Output ‘possibly equal’

**Step 3:** End.
Sender formal algorithm

At the source side, the formal algorithm for sender A is given as follows:

Algorithm senderA(message, fp)

%% Input: Message
%% Output: fingerprint
Begin
a = Binary string of Message
Randomly pick p
return(a mod p)
End
Algorithm receiver(fp(a), p)

%%% Input: Fingerprint of sender A and prime number p
%%% Output: Status of the string comparison

Begin
b = Binary string of local Message
fp(b) = b mod p
if fp(a) ≠ fp(b) then
    Output 'strings are not equal'
else
    Output 'possibly equal'
End if
End
Complexity analysis

*Complexity Analysis*
If there are \( n \) bits in the binary representation of the message \( a \) and \( b \), then \( \log n \) bits are present. Therefore, the complexity of the algorithm is \( O(\log n) \). This algorithm needs to be executed at least \( k \) times. Here, \( k \) is the number of trials. It can be observed that as the number of ‘\( k \)’ trials increases, reliability of the algorithm also increases.
Randomized quicksort

The informal algorithm for randomized quicksort is given as follows:

**Step 1:** Pick an element of the array randomly as a pivot element.
**Step 2:** Use the pivot element to position the list into sub-lists.
**Step 3:** Recursively sort the sub-lists.
**Step 4:** Combine all the sorted sub-lists.
Formal algorithm

Algorithm of randomized quicksort $A[\text{first, last}]$

%%% Input: Array A
%%% Output: pivot element

Begin
  if first < last then
    k = random (first, last)  % Generate the random number in this range
    swap (A[first], k)
    pivot = Rsplit(A)  % Select the pivot randomly
    randomizedquicksort(A[first .. pivot])
    randomizedquicksort(A[pivot + 1, last])
  end if
end
Complexity analysis

Let the input array $A$ be $\{x_1, x_2, x_3, \ldots, x_n\}$ and $X_{ij}$ be the indicator random variable indicating whether two elements $x_i$ and $x_j$ are compared or not. The question is to determine the number of counts, a given $x_i$, and $x_j$ be compared.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

where $X$ indicates the total number of comparisons made by the algorithms. Comparison of $x_i$ and $x_j$ would happen only under the following two conditions:

1. A sub-problem of quick sort contains $x_i$ and $x_j$.
2. Either $x_i$ or $x_j$ is chosen as a pivot element.
Rewriting the equation,

\[ E(X) = E \left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[x_{ij}] \]

\[ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(x_j \text{ is compared } x_j) \]

As said earlier, the question of comparing \( x_i \) and \( x_j \) would arise only if either is chosen as a pivot element. Otherwise, they would not be compared at all.

\[ \therefore \Pr[x_i \text{ is compared to } x_j] \]

\[ = \Pr[x_i \text{ is chosen as a pivot}] + \Pr[x_j \text{ is chosen as a pivot}] \]

\[ = \frac{1}{j-i+1} + \frac{1}{j-i+1} \]
Final complexity analysis

It can be observed that the pivot is chosen from a set which has $j - i + 1$ elements. In addition, both the events are equally likely. This implies that

$$Pr[x_i \text{ is compared to } x_j] = \frac{1}{j - i + 1} + \frac{1}{j - i + 1} = \frac{2}{j - i + 1}$$

In other words, the probability $x_p = 1$ is $\frac{2}{j - i + 1}$

As discussed earlier,

$$E(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[x_p] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Substitution of $j - i = k$ in this equation, it can be rewritten as follows:

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{i+1} \leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{i+1} + \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} = \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{i+1} = O(n \log n)$$

Recall that $1/k$ is a harmonic series whose time complexity is $O(\log n)$. Therefore, the expected run-time of randomized quick sort is $O(n \log n)$. 
Approximate algorithms

algorithms. An algorithm for problems (mostly NP-Hard or NP-Complete) is an approximate algorithm if it can give optimal solutions within a certain bound and also if it is possible to establish the solution guarantee analytically in worst case or on average in general.

The quality of an approximation algorithm is decided by (a) time complexity analysis and (b) comparing the generated feasible solution with the optimal solutions (‘goodness factor’).
Goodness Factor

The goodness of the approximation algorithm can be estimated through metrics. Let us assume that $A$ is the approximation algorithm for a given problem $Q$. Let $I$ be the instance of the problem $Q$. If $Q$ is a minimization problem, then the ratio $k = \frac{A(I)}{OPT(I)}$. Here, $OPT(I)$ is the optimal solution for the problem $Q$ and $A(I)$ is its feasible near-optimal solution. The range of the accuracy ratio is 0–1. On the other hand, if $Q$ is a maximization problem, then the accuracy ratio is given as the reciprocal of $k$ so that the accuracy ratio is in the range $0$–$1$. 
Goodness Factor

An approximation algorithm that gives a near-optimal solution in polynomial time, which is at most \( r \) times the optimal solution for any instance of the given problem is called \( r \)-approximation algorithm. Here, \( r \) is called the worst-case ratio bound or approximation ratio and is given as follows:

\[
\frac{A(I)}{OPT(I)} \leq r
\]

The best value of \( r \) for which the inequality holds for all instances of the problem, that is, \( r = \max_I \frac{A(I)}{OPT(I)} \), is called a performance ratio.

The value of performance ratio \( r \) is \( \geq 1 \) if the problem is of minimization problem and for maximization problem \( r \leq 1 \).
Advantages

1. Approximation algorithms optimize computer resources such as space and time. Hence, these algorithms are applicable not only to NP-hard problems but can also replace algorithms that utilize more computer resources.

2. Approximation algorithms help categorize problems based on their difficulty levels.
3. Approximation algorithms are valuable tools for developing and evaluating different types of heuristics for a given problem.
Disadvantages

The following are the disadvantages of approximation algorithms:

1. Approximation algorithms focus only on the worst-case measures and ignore heuristics that often work well in practical applications.
2. Approximation algorithms are limited to only a certain set of problems and not applicable to decision problems.
Classification of approximate algorithms

**Absolute approximation**  An algorithm $A$ is called an absolute or constant ratio approximation algorithm if approximate ratio $r$ is a fixed constant. This implies that the difference between an absolute optimal solution and the solution generated by the approximation algorithm is always a constant. It also means that approximation algorithm is independent of the input instance of the problem.

**Logarithmic approximation**  If $r$ is $O(\log (I))$, where $I$ is the instance of the problem, then the algorithm is called a logarithmic ratio or logarithmic approximation.
**$f(n)$-approximation**  An algorithm $A$ is called $f(n)$-approximation if and only if, for all instances of size, the approximation is $O(n^{1+\varepsilon})$. This assumes that the approximate solution $A(I)$ is greater than zero.

**$\varepsilon$-approximation**  An algorithm $A$ is called $\varepsilon$-approximation if and only if, for all instances of size, the following condition holds: $|A(I) - \text{OPT}(I)| \leq \varepsilon$ for all instances $I$ and for $\varepsilon > 0$. 
More Types

$f(n)$-approximation An algorithm $A$ is called $f(n)$-approximation if and only if, for all instances of size, the approximation is $O(n^{1-\varepsilon})$. This assumes that the approximate solution $A(I)$ is greater than zero.

$\varepsilon$-approximation An algorithm $A$ is called $\varepsilon$-approximation if and only if, for all instances of size, the following condition holds: $|A(I) - \text{OPT}(I)| \leq \varepsilon$ for all instances $I$ and for $\varepsilon > 0$. 
Approximation Schemes

**Approximation schemes** An approximation algorithm \( A(\varepsilon) \) is one that accepts two inputs – input instances and approximation ratio \( \varepsilon \) (where \( \varepsilon > 0 \)). It approximates the optimal solution within a bound of \((1 + \varepsilon)\) called a scheme. There are two approximation schemes available which are given as follows:

**Polynomial time approximation scheme (PTAS)** For every \( \varepsilon \), if the algorithm runs in polynomial time, then the algorithm is called PTAS. In other words, PTAS is a set of algorithms, where for every \( \varepsilon \), there is an \((1 + \varepsilon)\) approximation algorithm for a minimization problem and \((1 - \varepsilon)\) for a maximization problem that runs in polynomial time.

**Full time approximation scheme (FPTAS)** It includes a set of algorithms that belongs to PTAS and in addition ensures that for every \( \varepsilon \), there is an \((1 + \varepsilon)\) approximation algorithm for a minimization problem and \((1 - \varepsilon)\) for a maximization problem that runs in polynomial time in ‘\(n\)’ (\(n\) is the input instance) as well as \(\frac{1}{\varepsilon}\).
Classification of Approximation problems

**APX**  This stands for “Approximiable”. This is a class that includes problems that have relative performance as a constant.

**Log-APX**  This stands for “Logarithmic approximable”. Log-APX is a class problems whose performance ratio is $|\log(I)|$.

**Poly-APX**  This stands for “Polynomial approximable”. This is a class of problems that achieve the approximation ratio in polynomial time for all instances of the problem.

**PTAS**  Class PTAS is a set of problems that can be solved by polynomial time approximation scheme.

**FPTAS**  Class FPTAS is a set of problems that can be solved by fully polynomial time approximation time scheme.
Design Principles

Approximation algorithms can be designed using the following strategies:

**Heuristics and greedy approach**  One easy way to handle hard problems is to use some greedy rules, called heuristics. Normally heuristic rules work, but occasionally they may fail also.

**Linear programming and integer programming**  Linear programming has already been discussed in Chapter 17. One can formulate linear programming for an optimization problem. Integer programming forces the constraints to only integer values. The duality of linear programming is also helpful in finding approximation algorithms.

**Dynamic programming**  Dynamic programming is discussed in Chapter 13. Dynamic programming scales down the input of a given optimization problem so that it can be solved within reasonable time. This approach gives approximate solution that is reasonably effective.
### Difference between Heuristics and approximation algorithms

Table 19.2  Difference between heuristics and approximation algorithms

<table>
<thead>
<tr>
<th>S. no.</th>
<th>Heuristics</th>
<th>Approximation algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Do not guarantee a feasible solution</td>
<td>Guarantee a feasible solution within a bound and can be validated analytically.</td>
</tr>
<tr>
<td>2.</td>
<td>May fail</td>
<td>Approximation algorithm, if available, always work and will not fail as heuristic algorithms</td>
</tr>
<tr>
<td>3.</td>
<td>May take possible exponential time</td>
<td>Take only polynomial time</td>
</tr>
<tr>
<td>4.</td>
<td>Are tested experimentally</td>
<td>Are tested analytically</td>
</tr>
</tbody>
</table>
Nearest neighbor heuristic

**Step 1:** Choose an arbitrary node as the starting vertex.
**Step 2:** Visit all nodes based on the ‘nearest-neighbour rule’ and return to the starting vertex.
**Step 3:** Return the TSP tour and exit.
Example

Fig 19.3  General TSP
Let A be the starting vertex. Vertex A is connected to nodes B, C, and D. The nearest neighbour is C, as the distance from vertex A to C is only 3, which is less than the distance from the other vertices B and D. Therefore, the selected path is A–C. From C, the nearest neighbour is B. From B, the nearest neighbour is D, and finally one can return to the starting vertex A. So the path is A–C–B–D–A, which is having a cost of 24. On the other hand, if the starting vertex is B, then the selected path would be B–C–A–D–B with a cost of 24. Similarly, paths can be obtained by considering different starting nodes, and the best optimal tour can be obtained.
Formal algorithm

Algorithm NNA(G,d)

%% Input: G = <V, E> is the graph and d is the cost or distance
%% Output: TSP tour
Begin
    Initialize edge list L to all unvisited vertices.
    Start at an arbitrary vertex u and delete it from list L
    while (L is not null)
        pick the nearest neighbour of u, say vertex v, based on distance d
        visit vertex v from u
        remove the visited vertex v
    End while
End
Multi-fragment Heuristic method

A multi-fragment heuristic is a trial and error technique that aims to construct a TSP tour by ensuring that the vertex may not have a degree of 3 and tour length less than of ‘n’. Here n is the number of vertices of the given graph. The steps of a multi-fragment heuristic could informally be stated as follows:

**Step 1:** Sort the edges in ascending order based on the weights of the edges and store them in $E'$. 

**Step 2:** Repeat until a tour of given $V$ cities are involved. Add the next edge $u$ of $E'$ if feasible. The feasible condition is that the added edge should not create a vertex of degree 3 or a cycle of length $< n$. Here, n is the length of the tour. Or skip the edge.

**Step 3:** Return $S$. 
Formal algorithm

Algorithm MFH(G, d)

%% Input: Graph G and d-cost or distance
%% Output: Tour

Begin
    Sort E based on weights and stores it in list E'
    while (V is not null)
        pick the next edge, say u, of E'
        if feasible(u) then
            Add the next edge u to tour
        else
            Reject the edge
        End if
    End while
    return tour
End
Complexity Analysis

Experimentally, this algorithm is known to have an approximation ratio less than \( \frac{1}{2} \cdot (\lceil \log_2 n \rceil + 1) \)
Twice-the-path-Heuristic

Fig. 19.5  Stages of MST-based algorithm (a) Sample MST (b) Addition of double edges (c) Inorder traversal for Euler tour (d) Final tour
Informal algorithm

The informal algorithm for twice-the-path algorithm is given as follows:

**Step 1:** Find minimum spanning tree for the given graph.
**Step 2:** Create a multipath by duplicating the edges
**Step 3:** Find Eulerian Tour
**Step 4:** Output equivalent Hamiltonian tour.
**Step 5:** End.
Formal algorithm

Algorithm MST-Approx(G)

%%% Input: G with vertices V and edges E
%%% Output: TSP tour
Begin
    Compute MST T of G
    Get Eulerian graph by doubling edges
    Find Eulerian tour by inorder traversal
    Output Hamiltonian tour by shortcutting the Eulerian tour
End
Complexity analysis

An MST-based algorithm is a 2-approximation algorithm. How? Let OPT be the cost of the optimal tour. Let MST be the total length of the MST, which is the sum of all the edges of the MST. Let $A$ be the length returned by the algorithm. It can be observed that to get the inorder traversal, one has to visit every vertex twice. Therefore, the tour length $A = 2 \times \text{MST}$. The final tour is obtained by deleting the repeating vertices. Using triangular inequality, one can argue that this would not make the tour longer. Therefore, $A \leq 2 \times \text{MST}$. If one can remove one edge from the optimal tour, then a spanning tree can be obtained. Therefore, $\text{MST} \geq \text{OPT}$. So, one could combine these conclusions to get $A \leq 2 \times \text{OPT}$. Hence, this algorithm is a 2-approximation algorithm.
Example

Example 19.4  Consider the following MST given in Fig. 19.6. Construct the TSP tour using twice-the-path (or twice-around-the-tree) heuristics.

Solution  As per the heuristics, one has to create double edges and the duplicated MST is given in Fig. 19.7.
Solution

It can be observed that the Eulerian cycle can be created. This is given by 1-2-3-4-3-2-1. The feasible solutions can be created by deleting the repeated edges. Let us construct a tour: Starting from vertex 1, let us branch out to 2, 3, and 4. Again vertices 3 and 2 are encountered and hence they are deleted. This results in a tour 1-2-3-4-1 giving a path length of $2 + 6 + 3 + 3 + 6 + 2 = 22$. 
Christofides algorithm

**Step 1:** Initially, an MST is created using Prim or Kruskal algorithm, as discussed in Chapter 10.

**Step 2:** Add edges of minimum matching to all odd-degree vertices in the MST.

**Step 3:** Form a multi-graph by adding extra edges and find the Eulerian circuit.

**Step 4:** Find the equivalent Hamiltonian circuit and output it as a TSP tour.
The stages of the Christofides algorithm are shown in Fig. 19.8.

**Fig. 19.8** Stages of Christofides algorithm (a) Original graph (b) Matching (c) Final tour
Formal algorithm

Algorithm Christofides (G,d)

%% Input: G = <V, E> is the graph and d is the cost or distance matrix of G
%% Output: TSP tour

Begin

Compute MST T of G
S = {List of odd degree vertices of T}
Find minimum cost matching M of S in G
Eulerian graph is created using T + M
Find Eulerian tour and its equivalent Hamiltonian tour
Output Hamiltonian tour

End
Complexity analysis

**Complexity Analysis**

The Christofides algorithm is an approximation algorithm, and it performs better than the aforementioned approximation algorithm. How? Let OPT be the optimal tour. Let $A$ be the length of the tour completed by the algorithm and MST be the length of a minimum spanning tree. It can be observed that in this algorithm, the Euler tour is obtained by adding matching of an odd vertex to $T$. Let ML be the length of minimum odd-vertex matching. This implies that $A \leq$ MST + ML. ML cannot be greater than OPT, and also can be broken into two cycles. Therefore, one cycle can be at most OPT/2. By combining these, we get

$$ML \leq \frac{OPT}{2}$$

As, MST $\leq$ OPT, this implies that

$$A \leq OPT + \frac{OPT}{2}$$
$$\leq \frac{3}{2}OPT$$

Hence, Christofides is a 1.5-approximation algorithm.
Greedy knapsack approximation algorithm

1. Read all items with its profit and weight
2. Compute the profit to weight ratio
3. Sort the items based on profit to weight ratio
4. Fill the knapsack with the items sorted using step 3 if the entire item fits
5. Return the profit accumulated so far when added item crosses the knapsack capacity or profit of this added item as maximum profit.
6. End.
Formal algorithm

Algorithm greedy_approx_knapsack(S,w,p,W)

%% Input: Set of objects S with its weight and profit, W is the knapsack capacity
%% Output: Set of objects that satisfies knapsack constraints.

Begin
    Sort the objects of S based on $\frac{p_i}{w_i}$
    While adding the object j does not cross the capacity W do
        Add object onto knapsack
        Profit = Profit + pj
    End While
    %% come out of the loop when object j is crossing W
    Return max{$(a_1, a_2, \ldots, a_{j-1}) \cup a_j$}
End
Complexity analysis

Let us assume that addition of the object $a_j$ makes the weight cross the knapsack capacity $W$. In that case, the weight would be $\sum_{i=1}^{j-1} p_i$. The leftover space would be $W - \text{initial weight}$. Let us relax the condition of 0/1-approximation. Assume that the fraction of the object can be added. If so, the total profit would be $\sum_{i=1}^{j-1} p_i + \frac{W - S}{w_j} \times p_j$. Here, $S$ is the size of the objects \{1, 2, 3, \ldots, j - 1\}. This should be less than the optimal value $\text{OPT}$. Therefore, $\sum_{i=1}^{j-1} p_i + \frac{W - S}{w_j} \times p_j$ should be no less than $W$. It implies that \{$a_1$, $a_2$, \ldots, $a_{k-1}$\} or $a_k$ should be larger than $\frac{1}{2} \times \text{OPT}$. Therefore, this algorithm for knapsack problem is a $\frac{1}{2} \times \text{OPT}$ algorithm.
Vertex cover problem

The input for a vertex cover problem is a graph $G = (V, E)$. The aim of the vertex cover problem is to find a vertex cover such that every edge of $G$ is incident on at least one vertex in $vc$. 
Informal algorithm

**Step 1:** Initialize the vertex cover $vc$ as null, that is, $vc = \{\}$.  
**Step 2:** While the edge list $E$ is not empty do the following:  
- **2a:** Pick an edge $e = \langle u, v \rangle$ arbitrarily  
- **2b:** $vc = vc \cup \{v\}$.  
- **2c:** Remove $e$ along with all edges $u$.  
- **2d:** End while.  
**Step 3:** Return $vc$ and exit.
Algorithm greedy_vertex-cover(G)

Input: \( G = (V, E) \), \( E \) is the edge list and \( V \) the vertex list
Output: Cover for the given graph

Begin
\( \text{vc} = \emptyset \quad \text{%% Initialize vertex cover as null} \)
\( E' = E \)
While (\( E' \neq \emptyset \)) do
    pick an edge \( e = (u, v) \in E' \) arbitrarily
    \( \text{%% for weighted set cover this should be minimum} \)
    \( \text{vc} = \text{vc} \cup \{v\} \)
    \( \text{%% Update all edge list E'} \)
    \( E' = E' - \{\text{all edges that are adjacent to u or v}\} \)
End while
Return \( \text{vc} \)
End
Better approximation algorithm

Step 1: Initialize the vertex cover \( vc \) as null, that is, \( vc = \{\} \).

Step 2: While the edge list \( E \) is not empty do the following:
   2a: Pick an edge \( e = \{u, v\} \) arbitrarily.
   2b: \( vc = vc \cup \{u, v\} \).
   2c: Delete \( e \) and all edges that has \( u \) and \( v \) end points.

Step 3: Return vertex cover \( vc \).

Step 4: End.
Formal algorithm

Algorithm `mod_greedy_vertex-cover(G)`

%% Input: G = <V, E>, E is the edge list and V is the vertex list
%% Output: vertex cover
Begin
  vc = Ø  %% Initialize vertex cover as null
  E' = E
  while (E' ≠ Ø) do
    pick an edge e = (u, v) ∈ E' arbitrarily
    %% for weighted set cover this should be minimum
    vc = vc ∪ {u, v}
    %% Update all edges
    E' = E' - {all edges that are adjacent to either u or v}
  End while
  Return vc
End
Complexity analysis

A vertex cover covers every edge in matching $M$ and includes at least one end point. In matching, two edges share the same point. Therefore, it can be proved that $\text{OPT} \geq |M|$. It has been proved that the accuracy ratio is 2, as the arbitrary picking of an edge amounts to maximal matching $M$. The minimal covering is $|M|$. However, the vertex cover algorithm returns a vertex cover with at least vertices of size $2 \times |M|$. Therefore, one can conclude that the returned vertex cover is $\geq 2 \times \text{OPT}$. Therefore, it can be observed that the accuracy ratio of vertex cover is 2. Thus, the complexity analysis of vertex cover problem is $2 \times \text{OPT}$.
Set Cover Problem

A set cover is a generalized version of a vertex cover problem.
Example – How to Select Technicians

The technicians and the tests each of them performs are as follows:

Technician 1 = \{T_1, T_2\}
Technician 2 = \{T_2, T_5, T_6\}
Technician 3 = \{T_3, T_5, T_6\}
Technician 4 = \{T_3, T_4, T_5, T_6\}
Technician 5 = \{T_1, T_3, T_4, T_6\}

The set cover problem would be to find the minimum number of technicians who can cover all the tests. For example, in this case, Technicians 2 and 5 are sufficient to cover all the tests. The optimization version can involve cost also. If technicians charge for performing each test, then the problem would be to select technicians such that the cost is minimum.
Informal algorithm

**Step 1:** Initialize a set cover $c$ as null, where $c$ is the set of elements covered so far.

**Step 2:** If cover $c$ is not equal to the universal set $U$, then do the following:

2a: Calculate cost effectiveness of every set and find the minimum cost set $A$.

2b: Add the minimum cost-effective set $A$ ($A$ is a subset of $S$) as $c = c \cup \{A\}$.

2c: Remove the covered elements.

2d: Update the cost effectiveness of the sets with new elements and price of elements $e$ of picked set $A$ as the cost effectiveness of the picked set; update the remaining sets as $U = U \setminus \{A\}$

**Step 3:** Return $c$. 
Algorithm greedy_setcover(U, S)

%% Input: Universal set U and Set of elements S
%% Output: Set cover c

Begin
  c = φ  %% Initialize set cover as null
  while (c ≠ U) do
    pick minimum cost set A of S
    for all elements e, set price of e = minimum cost set
    c = c ∪ {A}
    U = U - {A}
  End while
  Return c
End
Complexity analysis

Let us assume that there are \( n \) points or elements that need to be covered. Assume that at certain stage, there are \( r \) items that remain to be covered. Let us assume that the optimal set cover is \( k \). Therefore, some set covers \( r/k \) items. Assuming the selected set covers new points, the uncovered elements would be \( r - r/k = r(1 - (r/k)) \). As there are \( n \) elements, this needs to repeated \( n \) times so that \( r \leq n \left(1 - \left(\frac{1}{k}\right)\right)^j \). It must be observed that \( r \) is an integer and greater than 0. So the process ends when \( n \left(1 - \left(\frac{1}{k}\right)\right)^j < 1 \). This implies that \( j = O(k \log n) \).

Therefore, the approximation ratio is \( j/k = O(k \log n)/k = O(\log n) \). Therefore, this algorithm is \( O(\log n) \)-approximate. In other words, the approximation ratio of a set cover problem is \( \leq \log n \times \text{OPT} \).
Approximation using Linear Programming

**Step 1:** Reduce an NP-hard problem to an integer programming problem.
**Step 2:** Relax the integer programming problem to linear programming.
**Step 3:** Find optimal solutions of the given problem using linear programming.
**Step 4:** Finally, round off the optimal fractional solution, either deterministically or randomly, to get integral approximate solutions.
Vertex cover problem

**Step 1:** Reduce the vertex cover problem to an integer programming problem using the principle of restriction. The variable $x_i$ can be assigned to a vertex cover $i$ as follows:

$$x_i = \begin{cases} 
1 & \text{if } i \in \text{vertex cover} \\
0 & \text{otherwise}
\end{cases}$$

This is subjected to the constraints that each edge has at least one end point in vertex cover. This constraint is represented as $x_i + x_j \geq 1$. The second constraint is that the vertex is either in vertex cover or not in it. This constraint is represented as follows: $x_i \in \{0, 1\}$. The objective function of this problem would be $x_1 + x_2 + x_3 + \cdots + x_n$.

**Step 2:** Integer programming is NP-hard. This can be reformulated, and it can be observed that all the constraints are now linear constraints. Reformulate the vertex cover as follows:

Minimize $x_1 + x_2 + x_3 + \cdots + x_n$

Subject to the constraints:

$$x_i + x_j \geq 1$$

$x_i \geq 0$ for each vertex $i$

$-x_i \geq -1$ for each vertex $i$
Informal algorithm

Step 3: This problem can be solved using an LP solver such as the simplex method. Let us assume that $x_i^*$ is the solution returned by the LP solver. If the solution is a fraction, then round it off to decide whether the node can be added to vertex cover or not.

Step 4: A rounding-off algorithm is simple. If $x_i^*$ is zero, then the vertex is not selected at all. If $x_i^*$ is 1, then it is selected. What if it is a fraction? As $x_i + x_j \geq 1$, either $x_i$ or $x_j$ must be greater than $\frac{1}{2}$. Hence, pick the vertex if it is at least $\geq \frac{1}{2}$.

Step 5: Return the vertices that are $x_i^* \geq \frac{1}{2}$ as solutions for all values of $i$. 
Algorithm approx-LP (G)

%%% Input: Graph G
%%% Output: vertex cover
Begin
   Formulate vertex cover as an LP problem

   Use simplex method to solve LP problem to return solutions \( x_i \) for all values of \( i \)
   for all values of \( i = 1 \) to \( n \)
      add \( x_i \) to \( C \) if \( x_i \geq 1/2 \)
   End for
Return \( C \)
End
Complexity analysis

*Complexity Analysis*

The linear programming-based set cover algorithm is a $2 \log (n)$ approximation algorithm.
Dynamic Programming

**Step 1:** Find a pseudo-polynomial or exact algorithm for solving the given problem.

**Step 2:** Trim or scale down input by rounding it off. This results in a reduction in the input size of the problem.

**Step 3:** Use dynamic programming to compute approximate solutions for the modified instances. This reduces the complexity of the algorithm, but may lead to inaccurate results.
Informal algorithm

Step 1: Choose a rounding factor $K$ as the maximum of $\left\{ 1, \frac{\min \{ p_1, p_2, \ldots, p_n \}}{n} \right\}$. Here, $n$ is the number of elements available for the knapsack problem. It can be observed that the maximum profit is used for scaling down a factor. Let the modified value be $\hat{p_i}$. In other words, $\hat{p_i}$ is between $p_i$ and $p_i + K$.

Step 2: Scale down the profit for all objects as $\hat{p_i} = \left\lfloor \frac{p_i}{K} \right\rfloor$ for all values of objects $i$.

Step 3: Compute the solution SOL of the modified instance using a dynamic programming algorithm/return SOL as the approximation solution.
Algorithm knapsack_approxamtion(A, p, w, ε)

%%% Input: A set of n items as an array A with its profits and weight, ε is approximation ratio
%%% Output: Approximate solution S'

Begin
    K = εP/n      %%% P is the largest profit
    \hat{p}_i = \frac{p_i}{K}  %%% Scaled profit

    Use dynamic programming to solve for the scaled down profit and get solution S'

End

return (S')
Complexity Analysis

Recollect that the dynamic programming solution of a knapsack problem is $O(n^2 P)$, where $P$ is the maximum of all object profits. When we round off $P$ by $K$, it implies that $O\left(n^2 \left\lfloor \frac{P}{K} \right\rfloor \right)$. It can be observed from Step 1 that $K = \frac{P}{n}$. By substituting this in the preceding equation, one gets

$$O\left(n^2 \left\lfloor \frac{P \cdot n}{P} \right\rfloor \right) = O\left(n^2 \left\lfloor \frac{n}{\varepsilon} \right\rfloor \right) = O\left(\frac{n^3}{\varepsilon} \right)$$

Now, it can be observed that the effectiveness is dependent on the value of $\varepsilon$. 
Subset Sum Problem – (Old Sum of Subsets) – Exact Solution

Step 1: Initially the set $s$ and target $t$ are read.
Step 2: Let $n$ be the number of elements of the set $s$.
Step 3: Let list$_i$ be the set of all values of the subset $s$ that ranges from 0 to $i - 1$. It is obtained by merging the list with the element $x_i$, given as list$_i + x_i$. For example, if list$_i$ is $\{1, 3, 4\}$, then list$_i + 3$ is given as $\{4, 6, 7\}$.
Step 4: If the set has elements greater than $t$, then discard the elements.
Step 5: Steps 3 and 4 are repeated till all subsets are processed.
Step 6: Return largest element of the final list list$|s|$.
Step 7: End.
Formal algorithm - Exact

Algorithm exact-sum(s, t)

%% Input: Set s of integer numbers, target t
%% Output: subset of numbers that sums to t
Begin
    list₀ = φ
    while i < |s| do
        list₁ = list₀ + xᵢ     %% Merge the lists
        Sort the elements of list₁
        If any xᵢ ∈ {list₁} > t then
            remove xᵢ
        end if
    end while
Return largest element of the list₁ₚ
End
This exact algorithm can be converted to an approximation algorithm by making certain alterations; for example, a trim procedure is included such that if several values are closer to one another, then only one of them is retained. This leads to the removal of many items, thereby ensuring that the input size is reduced. The reduced input size ensures that the growth would not be exponential. Trimming is done as follows: If $\delta$ is $(0 \leq \delta \leq 1)$ a trimming variable, then element $E$ is trimmed to another value $k$ if $\frac{k}{1+\delta} \leq E \leq k$. For example, if a list $\{21, 22, 53, 54\}$ is given, then one can approximate 22 as 21 and 54 as 53, leading to a compact list $\{21, 53\}$. Thus, by the trimming process, many elements can be removed from the list, as they are represented by the remaining elements of the list.
Algorithm \((s, t, \epsilon)\)

\begin{verbatim}
%% Input: Set \(s\) of integers, target \(t\), and approximation factor normally around 0.1
%% Output: Subsets of \(s\) that has elements whose sum is \(t\)
Begin

\text{List}_0 = \emptyset \quad %% \text{List} \emptyset \text{ is null}
while \(i \leq |s|\) do

\text{list}_i = \text{list}_{i-1} \cup \text{list}_{i-1} + x_i
trim the list by a factor of \(\epsilon/2n\)
if \(x_i \in \text{list}_i > t\) then \quad %% If any element is greater than \(t\), then remove it
end if
\(i = i+1\)
end while

Return largest element of the \text{list}_{|s|}

End
\end{verbatim}
Complexity Analysis

At any ‘i’ stage, the running time of the algorithm is $O(|\text{list}_i|)$. Trimming the list ensures that the successive elements $z'$ and $z$ have the property $z' \leq z \left(1 - \frac{\varepsilon}{n}\right)$. Therefore, the total number of the elements in the list at stage ‘i’ is $\log_\frac{1}{1 - \frac{\varepsilon}{n}} t \leq \Theta \left(\frac{n \ln t}{\varepsilon}\right)$. Since, the algorithm runs polynomial both in terms of ‘n’ and $1/\varepsilon$, the algorithm belongs to a fully polynomial approximation scheme.
Glossary

**Absolute approximation ratio**  The difference between absolute an optimal value and a solution generated by an approximation algorithm.

**Accuracy ratio**  The ratio $A(l)/OPT(l)$, where $Q$ is a minimization problem and $A$ is an approximation algorithm for an instance $l$.

**Approximation algorithm**  An algorithm that aims to provide near-optimal solutions with a performance guarantee.

**Approximation scheme**  An algorithm $A$ such that for $\varepsilon > 0$, the solution of $A$ satisfies the condition $SOL \leq (1 + \varepsilon)OPT$; in the case of a minimization problem, this would be $SOL \leq (1 - \varepsilon)OPT$.
**Glossary**

- **\( \varepsilon \)-approximation**  An algorithm such that for all instances of size \( n \), \( |A(l) - \text{OPT}(l)| \leq \varepsilon \) for all instances \( l \) and \( \varepsilon > 0 \).

- **Euclidean TSP**  A TSP for which all edges obey the triangular inequality and symmetry.

- **f(n)-approximation algorithm**  An algorithm such that for all instances of size \( n \), the approximation is \( O(n^{1+f}) \).

- **Fingerprint**  A shorter message that is representative of a very large message.

- **Heuristics**  ‘Educated guesses’ that are derived based on experience and intuition.

- **Las Vegas algorithm**  A randomized algorithm that terminates either with a correct answer with a probability of \( \geq 0.5 \) or with no answers at all.

- **Log ratio algorithm**  An approximation algorithm for which \( r \) is \( O(\log |I|) \).

- **Monte Carlo algorithm**  A randomized algorithm that gives yes/no kind of answers.

- **Performance ratio**  If \( A(l)/\text{OPT}(l) \leq r \), then the best value of \( r \) for which the inequality holds good for all the instances of the problem.

- **Randomized algorithms or probabilistic algorithms**  Algorithms that use randomized decisions to guide decision making in algorithms. These algorithms are also known as probabilistic algorithms.

- **Set cover**  A problem of choosing a subset \( S \) that minimizes cost such that the union of \( S \) equals set \( U \) that covers all the elements.

- **Subset sum problem**  The problem of finding a subset sum that is as large as possible, but less than the target \( T \).

- **Trim**  The idea of keeping only representative elements while eliminating all other elements.

- **Vertex cover**  A set of vertices such that every edge of \( G \) is incident on at least one vertex of vertex cover \( C \).