**al·go·rithm** A step-by-step problem-solving procedure, especially an established, recursive computational procedure for solving a problem in a *finite number of steps*.

**pro·gram** *n.* A set of instructions for solving a problem or processing data. Expression of an algorithm in a programming language.

- Algorithm properties
  - Input and output
  - Clear and unambiguous instructions
  - Termination in a finite number of steps
    - Collatz Conjecture, or $3x + 1$ problem
      - Start with any positive number $x$ and repeat:
      - If $x$ is even, divide by 2
      - If $x$ is odd, multiply by 3 and add 1
      - For example, if you start with 22, you get $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$
      - The conjecture is that no matter what number you start with, you will eventually get down to the same cycle, $4, 2, 1$, repeating over and over
  - Effectiveness of instructions
    - Should perform unambiguously, using a pencil and paper
    - Integers rather than floating point numbers; irrational numbers can not be expressed even with arbitrary long floating point numbers
- Study of algorithms
  - Devising algorithms
    - Art rather than a science
    - Different techniques such as greedy method, dynamic programming
  - Algorithm validation
    - Independent of coded program
    - Proof of correctness using assertions called predicates
    - Algorithm is *correct* if, for every input instance, it comes to a halt, and the result of the execution is correct
      - An incorrect algorithm may not terminate, or may give incorrect result
      - An algorithm never gives you a segmentation fault or a bus error
  - Analysis of algorithms
    - Analyze the performance of algorithms in terms of speed and storage requirements
  - Program testing
    - Debugging – Can only point to presence of bugs, not their absence
    - Profiling – Study of timing or performance measurement

**Algorithm specification**

- Possibilities include natural language, pseudocode, and flowcharts
- In this course, we'll use C/C++-like pseudocode but make it as simple to read as possible; no incomprehensible code like the one I use in OS
- Selection sort
  - Algorithm specs

```
Algorithm selection_sort ( A, n )
{
    // Input:   An array A of n elements
    // Output:  Elements in A are sorted

    for ( i = 0; i < n; i++ )
    {
        select smallest element A[j] in A[i..n-1];
        swap ( A[j], A[i] );
    }
}
```

- Subtask1: Select smallest element A[j] between A[i] and A[n-1]

```
j = i;
for ( k = i + 1; k < n; k++ )
    if ( A[k] < A[j] )
        j = k;
```

- Subtask2: Swap A[j] and A[i]

```
tmp = A[j];
A[j] = A[i];
A[i] = tmp;
```

- Notice that no type information or data structures are specified in the algorithm

| Program = Algorithm + Data Structures |
|---|

- Proof of correctness

  **Theorem 1** *Algorithm* `selection_sort ( A, n )` *correctly sorts a set of* $n \geq 1$ *elements; the result is left in* `A` *such that* $A[0] \leq A[1] \leq A[2] \leq \ldots \leq A[n-1]$.

  **Proof**: Look at iteration of the loop when $i = q$. At this point, we only compare the quantities $A[q] \ldots A[n-1]$, and no change is made to the elements $A[0] \ldots A[q-1]$. Also, $A[q] \leq A[r]$ at the end of the iteration such that $q < r \leq n$ due to the `swap` statement. It is obvious that if $q = n - 1$, the array is sorted.

- Recursive algorithms
  - Algorithms that invoke themselves on a [possibly] smaller instance of given data, and then combine the results
  - Direct *v.* indirect recursion
  - Towers of Hanoi
    - Tower with 64 disks, with smaller disks on top of larger
    - Three pegs A, B, C
    - Move all disks from A to B using C for intermediate storage such that at no time, a larger disk sits atop a smaller disk
    - Solution by recursion, assuming $n$ disks
      - Move $n - 1$ disks from A to C, using B as intermediate storage
      - Move the largest disk to B
      - Move $n - 1$ disks from C to B, using A as intermediate storage
      - Invoked by `towers_of_hanoi ( n, src, dest, tmp )`
    - The solution for a $n$-disk problem is formulated in terms of two $n - 1$ disk problems
  - Permutation generator
    - Given a set of $n \geq 1$ elements, print all possible permutations of the set

* Input: $\{a, b, c\}$
* Output: $\{\{a, b, c\}, \{a, c, b\}, \{b, a, c\}, \{b, c, a\}, \{c, a, b\}, \{c, b, a\}\}$
* $n$ elements lead to $n!$ permutations
* Simple algorithm involves picking up each element, followed by the permutation of remaining elements

```
algorithm perm ( A, k, n )
{
    // Input: An array A of n elements
    // Initiated by: perm ( A, 0, n );

    if ( k == n - 1 )        // Last element
        write ( A[0..n-1] );
    else
    {
        // A[k..n] has more than one permutation; generate recursively

        for ( i = k; i < n; i++ )
        {
            swap ( A[i], A[k] );
            perm ( A, k+1, n );
            swap ( A[i], A[k] );
        }
    }
}
```

**Performance Analysis**

- Judging an algorithm
  - Does the job. General protection faults do not do the job.
  - Correctness. Stays with original specifications.
  - Documentation.
  - Modular.
  - Readable and easily modifiable.
- Code optimization
  - Optimization can provide some improvement but not a lot
  - Optimizing the code for bubble sort does not compare with the performance of quicksort or mergesort
- Performance is also measured by the space and time complexity

  **Definition 1** ***Space Complexity*** *of an algorithm measures the effect on memory requirements as the amount of input data is increased.*

- Space complexity
  - Swapping two integers, without using a temporary variable in C
  - Iterative computation of $\sum a_i$

```
// Get the summation of n elements in array a

double sum_iter ( double * a, int n )
{
```

```
    double sum ( 0.0 );
    for ( int i ( 0 ); i < n; i++ )
    {
        sum += a[i];
    }
    return ( sum );
}
```

- Recursive computation of $\sum a_i$

```
// Recursively compute the sum of n elements in array a

double sum_rec ( double * a, int n )
{
    if ( n == 0 )
        return ( 0 )
    else
        return ( sum_rec ( a, n-1 ) + a[n-1] );
}
```

- Space needed is divided into
  - Fixed part, $c$, independent of input and output characteristics; $c$ denotes a constant
  - Variable part, $S_P$, dependent on I/O as well as recursion stack space; $P$ denotes the algorithm
- Space requirement $S(P)$ for algorithm $P$ is given by

$$S(P) = c + S_P$$

- We will concentrate on the component given by $S_P$
- For `sum_iter`, $S(P)$ works on an instance `a[n]` of the array. There are $n$ elements of `a`, with a requirement of one word for each. Also, there is one word for each of `i`, `sum`, and the variable `n`. Hence, $S_P \geq n$ for input array of size $n$.
- For `sum_rec`, the recursion stack space includes the space for formal parameters, the local variables, and the return address. Each recursive call requires at least three words. With the depth of recursion being $n$, recursion stack space needed is $\geq 3n$.

- Time complexity
  - Time complexity for algorithm $P$ is denoted by $t_P$
  - Counting the number of 1s in a word
  - We will only be concerned about the time of execution, ignoring the time for analysis, design, data collection, implementation, compiling, and testing
  - For theoretical time complexity analysis, we will not focus on individual micro operations such as addition and subtraction but be concerned with high level statements that may perform multiple micro operations (sequential statements only) in a single statement

    **Definition 2** *A **program step** is loosely defined as a semantically meaningful segment of a program that has an execution time independent of the instance characteristics. It is devoid of loops or other control statements.*
  - Both the following statements have a time complexity of $n$
    - Example 1:

```
for ( int i ( 0 ); i < n; i++ )
    sum += a[i];
```

    - Example 2:

```
for ( int i ( 0 ); i < n; sum += a[i++] );
```

- Recurrence relations
  - Consider the step count of the program `sum_rec` given by the recurrence

$$t_{\text{sum\_rec}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\text{sum\_rec}}(n-1) & \text{if } n > 0 \end{cases}$$

  - The recurrence is solved as:

$$\begin{aligned} t_n &= 2 + t_{n-1} \\ &= 2 + 2 + t_{n-2} \\ &= 2 + 2 + 2 + t_{n-3} \\ &\cdots \\ &= 2 * n + t_{n-n} \\ &= 2 * n + 2 \end{aligned}$$

**Definition 3** *The **input size** of any instance of a problem is described as the number of elements needed to describe that instance.*

- Asymptotic Notation (including Big-Oh)
  - Function with domain as the set of natural numbers
  - Allows the suppression of detail when analyzing algorithms
  - Convenient to describe the average and worst case running time function $T(n)$
  - $\Theta$-notation
    - Consider a given function $g(n)$
    - $\Theta(g(n))$ – Set of functions
    - $\Theta(g(n)) = \{f(n) : \exists$ positive constants $c_1, c_2$, and $n_0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$.
    - $f(n)$ can be *sandwiched* between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large $n$
      - $g(n)$ is both an upper and lower bound for $f(n)$
    - $\Theta(g(n))$ is a set
    - We write $f(n) = \Theta(g(n))$ to imply $f(n) \in \Theta(g(n))$
    - For all values of $n \geq n_0$, $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$
    - $\forall n \geq n_0$, $f(n)$ is equal to $g(n)$ within a constant factor
    - $g(n)$ is an asymptotically tight bound for $f(n)$
    - Every member of $\Theta(g(n))$ must be *asymptotically nonnegative*
    - $f(n)$ must be nonnegative whenever $n$ is sufficiently large
    - Consequently, $g(n)$ itself must be asymptotically nonnegative, or else, the set $\Theta(g(n))$ is empty
    - Therefore, it is reasonable to assume that every function used with $\Theta$-notation is asymptotically nonnegative
    - Prove $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
      - Determine positive constants $c_1, c_2$, and $n_0$ such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \forall n \geq n_0$$

      - Dividing by $n^2$ we have

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

      - $c_2 \geq \frac{1}{2}$ for $n \Rightarrow \infty$
      - $0 < c_1 \leq \frac{1}{14}$ for $n \geq 7$

* Prove $6n^3 \neq \Theta(n^2)$
  Assume that $c_2$ and $n_0$ exist such that $6n^3 \leq c_2 n^2 \; \forall n \geq n_0$
  $n \leq \frac{c_2}{6}$, not possible for arbitrarily large $n$ because $c_2$ is a constant
* Since any constant is a degree-0 polynomial, constant function can be expressed as $\Theta(n^0)$ or $\Theta(1)$

– $O$-notation

* Asymptotic upper bound
* Upper bound on a function within a constant factor
* Not as strong as $\Theta$-notation
* $O(g(n)) = \{f(n) : \exists$ positive constants $c$ and $n_0 \mid 0 \leq f(n) \leq cg(n) \; \forall n \geq n_0\}$
* Examples
  - $f(x) = 3n + 2 \Rightarrow f(x) = O(n)$ as $3n + 2 \leq 4n \; \forall n \geq 2$
  - $f(x) = 3n + 3 \Rightarrow f(x) = O(n)$ as $3n + 3 \leq 4n \; \forall n \geq 3$
  - $f(x) = 100n + 6 \Rightarrow f(x) = O(n)$ as $100n + 6 \leq 101n \; \forall n \geq 6$
  - $f(x) = 10n^2 + 4n + 2 \Rightarrow f(x) = O(n^2)$ as $10n^+4n + 2 \leq 11n^2 \; \forall n \geq 5$
* $f(n) = \Theta(g(n)) \; \Rightarrow \; f(n) = O(g(n))$
  - $f(n) = O(g(n)) \; \not\equiv \; g(n) = O(f(n))$
* $\Theta(g(n)) \; \supseteq \; O(g(n))$
* $O$-notation used to describe the running time of algorithm by inspection of algorithm structure
  - Doubly nested loop structure $\Rightarrow O(n^2)$
  - Biggest concern is the terms with the larger exponent, or the leading terms in a polynomial
* Three purposes of $O$-notation:
  1. Bound the error when small terms in mathematical formulas are ignored
  2. Bound the error when we ignore parts of a program that contribute a small amount to the total being analyzed
     - Such items will include initialization code and/or heuristics which may have a small but significant effect on the actual run-time
  3. Classify algorithms according to upper bounds on their total running times

| | |
|---|---|
| $O(1)$ | Constant computing time |
| $O(\log n)$ | Logarithmic computing time |
| $O(n)$ | Linear computing time |
| $O(n \log n)$ | |
| $O(n^2)$ | Quadratic computing time |
| $O(n^3)$ | Cubic computing time |
| $O(2^n)$ | Exponential computing time |

* Above reasoning allows us to focus on the leading term when comparing running times for algorithms (with the assumption that precise analysis can be performed, if necessary)
* $f(n) \in O(g(n)) \equiv f(n) = O(g(n))$
  - When $f(n)$ is asymptotically large compared to another function $g(n)$, i.e., $\lim_{N\to\infty} \frac{g(n)}{f(n)} = 0$, $f(n)$ is taken to mean $f(n) + O(g(n))$
  - We sacrifice mathematical precision in favor of clarity, with a guarantee that for large $N$, the effect of quantity given by $O(g(n))$ actually is negligible
  - As an example, we take the summation of the series $\sum_{i=1}^{N} i$ to be $\frac{N^2}{2}$ rather than $\frac{N(N+1)}{2}$
  - Such notation allows us to be both *precise* and *concise* when describing the performance of algorithms

**Theorem 2** *If* $f(n) = a_m n^m + \ldots + a_1 n^1 + a_0 n^0$, *then,* $f(n) = O(n^m)$.

**Proof**:

$$f(n) \quad \leq \quad \sum_{i=0}^{m} |a_i| n^i$$

$$\begin{aligned} &\leq \sum_{i=0}^{m} |a_i| n^m, \text{ since } n^m > n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| \\ &= O(n^m), \text{ Each } a_i \text{ is a constant} \end{aligned}$$

- $\Omega$-notation
  - Asymptotic lower bound
  - Best-case running time
  - $\Omega(g(n)) = \{f(n) : \exists$ positive constants $c$ and $n_0 \mid 0 \leq cg(n) \leq f(n) \; \forall n > n_0\}$
  - Best case running time of insertion sort $\Omega(n)$

  **Theorem 3** *For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$*

  - Useful to prove asymptotically tight bounds from upper and lower bounds
  - Running time of insertion sort falls between $\Theta(n^2)$ and $\Omega(n)$
- $o$-notation
  - Asymptotic upper bound provided by $O$-notation may or may not be asymptotically tight
  - $o$-notation denotes an upper bound that is not asymptotically tight
  - $o(g(n)) = \{f(n) :$ For any constant $c > 0$, $\exists$ a constant $n_0 > 0 \mid 0 \leq f(n) < cg(n) \; \forall n \geq n_0\}$
  - For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$
  - $f(n)$ becomes insignificant compared to $g(n)$ as $n$ approaches infinity, or

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- $\omega$-notation
  - $\omega$-notation denotes the asymptotic lower bound that is not tight
  - $\omega(g(n)) = \{f(n) :$ For any constant $c > 0$, $\exists$ a constant $n_0 > 0 \mid 0 \leq cg(n) < f(n) \; \forall n \geq n_0\}$
  - For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$
  - $f(n) = \omega(g(n))$ implies

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

  - $f(n)$ becomes arbitrarily large relative to $g(n)$ as $n$ approaches infinity.
- Comparison of functions
  - $f(n)$ and $g(n)$ are asymptotically positive
  - Transitivity

$$\begin{array}{lcllcl} f(n) = \Theta(g(n)) & \wedge & g(n) = \Theta(h(n)) & \Rightarrow & f(n) = \Theta(h(n)) \\ f(n) = O(g(n)) & \wedge & g(n) = O(h(n)) & \Rightarrow & f(n) = O(h(n)) \\ f(n) = \Omega(g(n)) & \wedge & g(n) = \Omega(h(n)) & \Rightarrow & f(n) = \Omega(h(n)) \\ f(n) = o(g(n)) & \wedge & g(n) = o(h(n)) & \Rightarrow & f(n) = o(h(n)) \\ f(n) = \omega(g(n)) & \wedge & g(n) = \omega(h(n)) & \Rightarrow & f(n) = \omega(h(n)) \end{array}$$

  - Reflexivity

$$\begin{array}{lcl} f(n) & = & \Theta(f(n)) \\ f(n) & = & O(f(n)) \\ f(n) & = & \Omega(f(n)) \end{array}$$

  - Symmetry

$$f(n) = \Theta(g(n)) \quad \text{if and only if} \quad g(n) = \Theta(f(n))$$

  - Transpose symmetry

$$\begin{array}{lll} f(n) = O(g(n)) & \text{if and only if} & g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) & \text{if and only if} & g(n) = \omega(f(n)) \end{array}$$

  - Analogy with two real numbers $a$ and $b$

$$\begin{array}{lcl} f(n) = O(g(n)) & \approx & a \leq b \\ f(n) = \Omega(g(n)) & \approx & a \geq b \\ f(n) = \Theta(g(n)) & \approx & a = b \\ f(n) = o(g(n)) & \approx & a < b \\ f(n) = \omega(g(n)) & \approx & a > b \end{array}$$

- Practical complexities
  - Time complexity is some function of instance characteristics
  - Useful to compare two algorithms that perform the same task
  - The size of instance $n_0$ above which the algorithms with complexity $O(n)$ and $O(n^2)$ diverge (sufficiently large)
  - Judging algorithms in practical terms on the basis of time
    - Run times of $10^6 n$ ms vs $n^2$ ms
  - For reasonably large $n$ ($n > 100$), only algorithms with polynomial complexity can provide a solution in reasonable time
- Performance measurement
  - Concerned with obtaining the space and time requirements of a given program, *by actually running it on a machine*
  - Must be careful that the test is performed on the same machine, under similar load conditions
  - Performance measurement *v.* asymptotic analysis
    1. Asymptotic analysis tells us the behavior only for sufficiently large values of $n$. For smaller values of $n$ (less than $n_0$ in the earlier definitions), the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of $n$.
    2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve, because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, a program with asymptotic complexity $\Theta(n)$ can have time complexity of $c_1 n + c_2 \log n + c_3$ or, for that matter, any other function of $n$ in which the highest-order term is $c_1 n$ for some constant $c_1$, $c_1 > 0$.
  - If you do not already know, write a program using `gethrtime(3C)` library function on Unix and check the performance of some code. You can also write a wrapper that forks and execs another program to measure its performance.
  - Generating test data
    - Worst-case data
    - Random data

**Hard Problems**

- Some hard problems can be solved with efficient algorithms; in polynomial time
- There are some problems for which there is no known efficient algorithm
- NP-complete problems
  - Nondeterministic Polynomial time
  - Subset of the class of problems for which there is no known efficient solution
  - It has never been proved whether there is or is not an efficient solution in existence for this class of problems

  - If you can find an efficient algorithm for any problem in the NP-complete class, then efficient algorithms exists for all problems in the class
  - Several NP-complete problems are similar, but not identical, to problems for which we know efficient solutions
- If an problem can be *mapped* to an NP-complete problem, you can stop looking for the best possible algorithm for it, and can settle with a *good* algorithm

**Parallelism**

- Physical limits to processor clock speed that cannot be increased beyond a limit
- High clock speed generates heat with the risk of melting the chips
- Design chips with multiple cores, designing algorithms that split work between those cores

**Randomized algorithms**

- Basics of probability theory

  **Sample point.** Each possible outcome of an experiment

  **Sample space.** Set of all possible outcomes; denoted by $S$

  **Discrete sample space.** If the number of all possible outcomes $S$ is finite, or countable

  **Event.** A subset of the sample space $S$; denoted by $E$
  - If $S$ consists of $n$ sample points, there are $2^n$ possible events, given by the power set of $S$
  - If two coins are tossed, there are four possible outcomes: $HH$, $HT$, $TH$, and $TT$, giving us $S$. With each outcome considered as a sample point, the possible events are enumerated as: $\{\}$, $\{HH\}$, $\{HT\}$, $\{TH\}$, $\{TT\}$, $\{HH, HT\}$, $\{HH, TH\}$, $\{HH, TT\}$, $\{HT, TH\}$, $\{HT, TT\}$, $\{TH, TT\}$, $\{HH, HT, TH\}$, $\{HH, HT, TT\}$, $\{HH, TH, TT\}$, $\{HT, TH, TT\}$, and $\{HH, HT, TH, TT\}$
  - It is easy to see that the number of outcomes is:

$$\left( \begin{array}{c} n \\ 0 \end{array} \right) + \left( \begin{array}{c} n \\ 1 \end{array} \right) + \cdots + \left( \begin{array}{c} n \\ n \end{array} \right)$$

  **Probability.** The probability of an event $E$ is defined to be $\frac{|E|}{|S|}$, where $S$ is the sample space
  - Probability of the event $\{\}$ is zero
  - Probability of the event $\{HH, TT\}$ is $\frac{2}{4}$

  **Mutual exclusion.** Two events $E_1$ and $E_2$ are said to be *mutually exclusive* if they do not have any common sample points, that is, if $E_1 \cap E_2 = \emptyset$
  - Toss three coins
  - Let $E_1$ be the event that there are at least two $H$s
  - Let $E_2$ be the event that there are at least two $T$s
  - $E_1$ and $E_2$ are mutually exclusive since there are no common sample points
  - Let $E_2'$ be the event that there is at least one $T$
  - $E_1$ and $E_2'$ are not mutually exclusive since they have $THH$, $HTH$, and $HHT$ as common sample points

  **Theorem 4**

  1. $P(\bar{E}) = 1 - P(E)$
  2. $P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$
     $\leq P(E_1) + P(E_2)$

**Conditional probability.** Let $E_1$ and $E_2$ be any two events of an experiment. The *conditional probability* of $E_1$ given $E_2$, denoted by $P(E_1|E_2)$, is defined as $\frac{P(E_1 \cap E_2)}{P(E_2)}$

- Toss four coins
- Let $E_1$ be the event that the number of $H$s is even and let $E_2$ be the event that there is at least one $H$
- $E_2$ is the complement of the event that there is no $H$
- Probability of no $H$ is $\frac{1}{16}$; $P(E_2) = 1 - \frac{1}{16} = \frac{15}{16}$
- $P(E_1 \cap E_2) = \frac{7}{16}$
- $P(E_1|E_2) = \frac{7/16}{15/16} = \frac{7}{15}$

**Independence.** Two events $E_1$ and $E_2$ are said to be *independent* if $P(E_1 \cap E_2) = P(E_1) \times P(E_2)$

- For independent events, $P(E_1|E_2) = P(E_1)$

**Random variable.** Let $S$ be the sample space of an experiment. A *random variable* on $S$ is a function that maps the elements of $S$ to the set of real numbers. For any sample point $s \in S$, $X(s)$ denotes the image of $s$ under this mapping. If the range of $X$, that is, the set of values $X$ can take, is finite, we say that $X$ is discrete.

- Let the range of discrete random variable $X$ be $\{r_1, r_2, \ldots, r_m\}$. Then, $P(X = r_i)$, for any $i$, is defined to be the number of sample points whose image is $r_i$ divided by the number of sample points in $S$.
- Flip a coin four times
- The sample space $S$ has $2^4$ sample points
- Define a random variable $X$ on $S$ as the number of heads in the coin flips
- For this random variable, $X(HTHH) = 3$, $X(HHHH) = 4$, and so on
- Possible values for $X$ are $0, 1, 2, 3, 4$, making $X$ discrete
- $P(X = 0) = \frac{1}{16}$
- $P(X = 1) = \frac{4}{16}$

**Expected value.** If the sample space of an experiment is $S = \{s_1, s_2, \ldots, s_n\}$, the *expected value* or the *mean* of any random variable $X$ is defined to be $\sum_{i=1}^{n} P(s_i) \times X(s_i) = \frac{1}{n} \sum_{i=1}^{n} X(s_i)$

- For three coin tosses, $S = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$
- Let $X$ be the number of heads in the coin flips
- Expected value of $X$ is: $\frac{1}{8}(3 + 2 + 2 + 1 + 2 + 1 + 1 + 0) = 1.5$

**Probability distribution.** Let $X$ be a discrete random variable defined over the sample space $S$. Let $\{r_1, r_2, \ldots, r_m\}$ be its range. Then, the *probability distribution* of $X$ is the sequence $P(X = r_1), P(X = r_2), \ldots, P(X = r_m)$, and $\sum_{i=1}^{m} P(X = r_i) = 1$.

**Binomial distribution.** A *Bernoulli* trial is an experiment that has two possible outcomes: `success` and `failure`. The probability of `success` is $p$. Consider the experiment of conducting the Bernoulli trial $n$ times. This experiment has a sample space $S$ with $2^n$ sample points. Let $X$ be a random variable on $S$ defined to be the number of `success`es in the$n$ trials. The variable $X$ is said to have a *binomial distribution* with parameters $(n, p)$. The expected value of $X$ is $np$. Also,

$$P(X = i) = \binom{n}{i} p^i (1-p)^{n-i}$$

**Lemma 1** ***Markov's inequality****. If $X$ is any nonnegative random variable whose mean is $\mu$, then*

$$P(X \geq x) \leq \frac{\mu}{x}$$

Markov's inequality is weak. Chernoff bounds provide tighter bounds for a number of important distributions, including binomial distribution.

**Lemma 2** ***Chernoff bounds****. If $X$ is a binomial with parameters $(n, p)$, and $m > np$ is an integer, then*

$$\begin{aligned} P(X \geq m) &\leq \left(\frac{np}{m}\right)^m e^{m-np} \\ P(X \leq \lfloor (1-\epsilon)pn \rfloor) &\leq e^{(-\epsilon^2 np/2)} \\ P(X \geq \lceil (1+\epsilon)np \rceil) &\leq e^{(-\epsilon^2 np/3)} \end{aligned}$$

*for all* $0 < \epsilon < 1$.

- Randomized algorithms
  - May make decisions inside the algorithms based on the output of a randomizer
  - May have different output and different run-time for different runs

    **Las Vegas algorithms.** Produce the same output for the same inputs

    **Monte Carlo algorithms.** May produce different outputs for different runs, even for the same input