Graphs and Trees ; Elements of Counting and Probability

(A quick graph Review) : Appendix B page 1168 and onwards

This subject is CS241 material

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.1 Directed Graphs. A **directed graph/digraph** G is a pair $G = (V, E)$ where V is a finite set and E is a binary relation on V (i.e a subset of $V \times V$). We shall customarily use n, m to denote the cardinalities of V and E, i.e. $n = |V|$ and $m = |E|$.

V is called the vertex set and E the edge set of G. The elements of both V and E are called respectively the vertices and edges of G. For directed graphs, we sometimes use the notation **nodes** and **arcs** to denote the vertices and edges respectively. Under such a notation we may use $G = (N, A)$ instead of $G = (V, E)$.

In a directed graph $(u, v) \in E$, where $u, v \in V$ represents an edge leaving u and entering v. In a digraph **self-loops** are possible. Multiple edges are not allowed. In G edge $e = (u, v)$ is **incident from** u or **leaves** vertex u and is also, **incident to** v or **enters** v. The endpoints of (u, v) are u and v. We sometimes also say that u, v are adjacent, but then "adjacency" does not specify the direction of it.

3.2 Undirected Graphs. An **undirected graph** G is a pair $G = (V, E)$ where V is a finite set and E is a set of unordered pairs of vertices.

In an undirected graph an edge such as $(u, v) \in E$ really means $\{u, v\}$. An UNDIRECTED GRAPH is **simple** if it has no self-loops and no multiple edges. By default we are going to assume that our undirected graphs are all simple. In G edge $e = (u, v)$ is **incident on** u and v. Two vertices connected by an edge are called **adjacent**, ie. u is adjacent to v or v is adjacent to u. For an undirected graph (u, v) and (v, u) are the same edge. The endpoints of (u, v) are u and v. We also say $e = (u, v)$ is incident on u and v.

3.3 'from', 'to', 'on' and 'between'. For an undirected graphs and edge $e = (u, v)$ we say e is incident on u and v. For a directed graph we say e is incident from u and incident to v thus the prepositions in the latter case indicate the direction of the edge. In the former case there is no direction so 'on' is neutral. By convention we use the same notation to represent an edge for the directed and undirected case, that is (u, v) is an edge from u to v in the directed case, whereas in the undirected case it represents an edge between u and v.

3.4 Set incidence. If S is a set of vertices an edge is **incident** to S if **exactly one** of its endpoints is in S.

3.5 Bipartite graphs. A graph $G = (V, E)$ is **bipartite** if its vertices V can be partitioned into two sets so that edges go from one set to the other only, i.e. the end points cannot both be in the same set.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Notation continued

3.6 Directed Graphs: In-degree, out-degree, and degree. For a directed graph G the **out-degree** of a vertex u is the number of edges incident from u. The **in-degree** of a vertex u is the number of edges incident to u. The **degree** of vertex u in a directed graph G is the sum of its out-degree and in-degree.

3.7 Undirected Graphs: degree. For an undirected graph G the **degree** of vertex u is the number of edges incident on u. Show that $\sum_{u \in V} deg(u) = 2m = 2|E|$.

3.8 Paths and Tours. We say that there is a path from u to v and indicate it as $u \rightsquigarrow v$ or we say that there is a path of length k from u to v and indicate it as $u \rightsquigarrow^k v$, if there is a sequence $\langle u_0, u_1, \ldots, u_k \rangle$ of vertices such that $u_0 = u$ to $u_k = v$ and $(u_i, u_{i+1}) \in E$, $0 \leq i < k$. $u_0 = u$ and $u_k = v$ are the endpoints of the path. The length of the path is the number of edges in it. We also say that u_i is **reachable** from u. A path is **simple** if the are no duplicate vertices (i.e. all vertices in the path are distinct). A subpath of a path is a contiguous subsequence of its (path's) vertices. In a path we have unique vertices and edges. In a **tour** we still have unique edges but vertices are not necessarily unique.

3.9 Cycles. A **cycle** in a directed graph is a path with $k \geq 0$ whose endpoints coincide ie $u = v$ and has at least one edge. A cycle is **simple** if all $u_1, \ldots u_k$ are distinct. A self-loop is a cycle of length 1. A cycle in an undirected graph is defined identically (self-loops are not allowed in undirected graphs as they are simple) with the addition that no edge is repeated.

3.10 Acyclic graphs. A graph with no cycles is called **acyclic**.

3.11 Subgraphs. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. G_1 is a subgraph of G_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

(c) Copyright A. Gerbessiotis.

3.12 Connected undirected Graph. An undirected graph is **connected** if every vertex v is reachable from every other vertex u i.e. $\forall u, \forall v \in V$ we have that there exists a path $\exists u \rightsquigarrow v$ (and by symmetry also $v \rightsquigarrow u$). The **connected components** of a graph are the equivalence classes under the **" is reachable from"** relation or in other words they are the maximal connected subgraphs of G. An undirected graph is connected if number of connected components is 1.

3.13 Strongly connected directed graph. A directed graph is **strongly connected** if every vertex v is reachable from every other vertex u i.e. $\forall u, \forall v \in V$ we have $\exists u \rightsquigarrow v$ and $\exists v \rightsquigarrow u$. We say then that u, v are mutually reachable. The **strongly connected components** of a graph are the equivalence classes under the **"are mutually reachable"** relation.

3.14 Isomorphism. Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection (surjective and injective function) f from V to V' such that $(u, v) \in E$ is mapped to $(f(u), f(v)) \in E'$.

3.15 Complete (undirected) graph. A **complete graph** on n vertices is a undirected graph such that every pair of vertices is adjacent. It is denoted by K_n.

3.16 Forest in an (undirected) graph. A **forest** is an acyclic undirected graph.

3.17 Tree in an (undirected) graph. A **(free) tree** or just a **tree** is an acyclic connected undirected graph.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.18 Free Trees. A **free tree** is an acyclic connected undirected graph. A connected forest is thus a tree. We often omit **(free)** from **(free) tree**.

3.19 Some tree facts.

Tree Theorem. Let G be an undirected graph. The following statements are equivalent

1. G is a free tree.
2. Any two vertices in G are connected by a unique single path.
3. G is connected but if any edge is removed from E the resulting graph is disconnected.
4. G is connected and $|E| = |V| - 1$.
5. G is acyclic and $|E| = |V| - 1$.
6. G is acyclic but if any edge is added to E the resulting graph contains a cycle.

3.20 Rooted Trees. A **rooted tree** is a free tree in which one of the vertices is distinguished from the others and is called the **root**. The root of a tree T will be represented sometimes by r. Vertices of a tree are also called **nodes**.

The number of children of the root define the 'degree' of the root. For a rooted tree we can define relationships such as descendant, ancestor, child and parent.

3.21 Ancestor, Descendant. Let P be a path from r to some vertex of T x. Then any node y of the unique path from r to x is called an **ancestor** of x. x is a **descendant** of y (r and itself). If $x \neq y$ then the x is a proper descendant of y and y is a proper ancesotr of x. x is an ancestor/descendant of x. The subtree rooted at x is the tree induced by descendants of x rooted at x.

3.22 Parent, Child, Siblings. If in the path the last edge from r to x is (z, x) z is called **parent** of x, x is called **child** of z. If x and u are both children of z they are called **siblings**.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.23 Leaf or external node. Internal nodes. A node with no children is called **a leaf** or **external node**. A non-leaf is called **an internal node**. (The root is also an internal node.)

3.24 Degree of a node in a tree. The number of children of a node defines its **degree** (note the difference between this definition and the degree of a vertex in a graph).

3.25 Depth and Height. The length of the path from r to x is the depth of x. A root r has depth $d(r) = 0$. If v is a tree node the $d(v) = d(p(v)) + 1$ where $p(v)$ is parent of v. The largest depth of any node is the height of the tree T. The height of a leaf v is $h(v) = 0$. For a non-leaf node u $h(u) = \{max\ \ h(w)$, where w is a child of $u\}$, $+1$.

3.26 Ordered tree. An **ordered** tree is a tree in which the children of a node are ordered (ie we can talk of the first, second etc).

3.27 Binary tree. A **Binary Tree** is a rooted and ordered tree that either

- contains no nodes, or
- it consists of three disjoint sets of nodes: a **root** node, a binary tree called **left-subtree** and a binary tree called **right-subtree**.

The root of the left subtree is called the left child of **root** and that of right subtree the right child.

3.28 Difference between binary trees and ordered trees. If a node has only one child it matters whether it is a left or a right child as opposed to ordered trees, where it doesn't matter.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.29 A full-binary tree, is a tree where each node has degree 2 or 0 (leaf).

ATTENTION !

3.30 A complete binary tree, is a tree where each leaf has the same depth and all internal nodes have degree 2.

END OF ATTENTION.

3.31 A full-binary tree vs complete binary tree. In several textbook the name full binary tree denotes a complete binary tree. BEWARE OF IT!

3.32 OUR TEXTBOOK's complete binary tree. The textbook is using an alternative definition for a complete binary tree on page 99 in the context of binary heaps. We shall call the tree that has the property of page 99 NOT a complete binary tree but a **heap tree** or **complete binary heap tree** instead. BEWARE OF IT!

3.33 Complete Binary tree of height h. Number of vertices in such a tree of height h is

$$1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1.$$

and the number of internal nodes is $2^h - 1$. Thus the number of external nodes (leaves) is 2^h.

Example. Suppose that internal node of depth i are d_i and internal nodes of depth $i+1$ are d_{i+1}. In a full binary tree how many leaves there are of depth $i+1$??? Answer is $2d_i - d_{i+1}$.

Example. If the height of the tree is h how many leaves of depth h there are?

Answer. There are no internal nodes of depth h (otherwise we would have had nodes in depth $h+1 >$ height of tree. Thus internal nodes of depth $h-1$ is by our notation d_{h-1}. Therefore number of leaves is $2d_{h-1}$ at depth h.

Question. A Complete binary tree has n nodes. How many of them are leaves, and how many of them are internal nodes (including the root).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.34 Dense graphs vs Sparse graphs. For a graph $G = (V, E)$ let $n = |V|$ and $m = |E|$, as already noted. A graph G is **dense** if $m \gg n$, and **sparse** otherwise. In some cases the former condition is restated as $m = \Theta(n^2)$ and the latter condition $m = o(n^2)$.

3.35 Adjacency matrix and Adjacency list. There are two major ways one can use to represent a graph

(1) **Adjacency matrix** representation. A graph $G = (V, E)$ on n vertices and m edges is represented by an $n \times n$ two-dimensional array A. Element $A(i, j)$ is 1 if there is an edge (i, j) in the graph, otherwise it is 0. For undirected graphs, A will be symmetric since for every edge (i, j) both $A(i, j), A(j, i)$ are set to 1. In general, m entries in the directed case and $2m$ entries in the undirected case are 1, whereas the remaining ones are 0. If the edges of the graph have weights, then instead of storing an 1 we store in A the weight of the edge. Total space requirements are $O(|V|^2)$.

(2) **Adjacency list** representation. The graph is represented by n linked lists in the form of an array of linked lists, one for each vertex of G. The linked list for vertex i contains all vertices that are neighbors of i, ie it contains all j such that (i, j) is an edge. For an undirected graph, j is in the linked list of i and i is in the list of j. Total space requirements are $O(|V| + |E|)$. The problem with this representation is that we cannot answer quickly questions of the form "Is j adjacent to i?" for a directed graph.

The **incidence matrix** of a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = [b_{ij}]$ such that $b_{ij} = -1$ if edge j leaves vertex i, $b_{ij} = +1$ if edge j enters vertex i, $b_{ij} = 0$ otherwise.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.36 Tree traversals. A **tree traversal** is a process of visiting each of the vertices of a rooted tree exactly once. For rooted trees preorder and postorder tree traversals are defined. For the specific case of a **binary** tree three such traversals are quite known: **inorder, postorder, preorder**. In preorder, parents are visited before their children; in postorder after their children; for binary trees, left children are visited before right children. Inorder traversal has a meaning only for binary trees: left subtree of a parent is visited before the parent followed by right subtree. As a rooted tree may or may not be ordered, inorder is not well-defined/meaningful.

```
pre-visit(u)   :: {pre[u]=pre++; print(<u,pre[u]>);} ;
in-visit(u)    :: {in[u]=in++; print(<u,in[u]>);} ;
post-visit(u)  :: {post[u]=post++; print(<u,post[u]>);} ;
```

```
BT-Inorder(u)
1. if u != NULL {
2.    BT-Inorder (left(u));
3.    in-visit(u) ;
4.    BT-Inorder (right(u));
   }
```

```
BT-Preorder(u)
if u != NULL {
  pre-visit(u);
  BT-Preorder(left(u));
  BT-Preorder(right(u));
}
```

```
BT-Postorder(u)
if u!= NULL {
 BT-Postorder(left(u));
 BT-Postorder(right(u));
 post-visit(u);
}
```

```
   BT-Perform-All-Three(u)  {
1. pre-visit(u);
2. if left(u) != NULL
3.   BT-Perform-All-Three (left(u));
4. in-visit(u);
5. if right(u) != NULL
6.   BT-Perform-All-Three (right(u));
5. post-visit (u);
   }
```

```
     Euler-tour(u)
  1. left-visit(u);   // similar to pre-visit
  2. if left(u)!=NULL
  3.    Euler-tour(left(u));
  4. down-visit(u);   // similar to in-visit
  5. if right(u) != NULL
  6.    Euler-tour(right(u));
  7. right-visit(u);  // similar to post-visit
```

(c) Copyright A. Gerbessiotis.

3.37 BFO (Bread-first Order) traversal. Another useful traversal is the **breadth-first order** (BFO) traversal obtained as follows.

```
 RT-BFO(root(RT))  // RT is a rooted (not necessarily binary) tree
1. visit(root(RT)); // visit[root(RT)]=i++; print(visit[root(RT)]);
2. repeat until all vertices are visited
3.  visit an unvisited child of the LEAST RECENTLY VISITED
    vertex with an unvisited child
 END_BFO
```

If **LEAST RECENTLY** is replaced by **MOST RECENTLY**, a **depth-first order** (DFO) traversal is obtained.

We can extend these two traversals from rooted trees to graphs. If G is a graph and s is an arbitrary starting vertex, we use this vertex as the "root" of the traversal by visiting s first and then repeating the following step until there is no unexamined edge (u, v) such that vertex u has been visited.

Search Step. Select an unexamined edge (u, v) such that u has been visited and examine this edge, visiting vertex v if v has not been visited yet.

3.38 DFO (Depth-First Order) traversal. If u is the MOST RECENTLY VISITED vertex we get a DEPTH-FIRST order. In BREADTH-FIRST ORDER traversal v is the least recently visited vertex.

(c) Copyright A. Gerbessiotis.

3.39 Union-Find data structures (aka disjoint-set data structures). A **disjoint-set data structure** maintains a collection of $S = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets consisting of n elements. Each set is identified by a **representative** (i.e. leader) which is some arbitrary member of that set. In many applications it doesn't matter which member becomes the representative of the set. We are satisfied that our queries return the same result as long as the sets are not modified. There are some cases where a representative may possess some additional property (ie it is the smallest element of a set with some order) but we are not interested in it right now.

For an object x we wish to support in such a data structure the following operations.

(1) **MakeSet(x)** creates a new set whose only member is pointed to by x. Since sets are disjoint we assume that x is not already in a set.

(2) **Union(x,y)** unites the dynamic sets that contain x and y say S_x and S_y into a new set that is the union of these two sets. The two sets are disjoint prior to the operation. The representative of $S_x \cup S_y$ is some member of that set (not necessarily x or y). The two old sets S_x and S_y are destroyed after the creation of $S_x \cup S_y$.

(3) **Find(x)** returns a pointer to the representative of the unique set containing x (parameter x is a pointer to x).

We are interested in operations where n is the number of elements and thus the maximum number of MakeSet operations and m is the total number of all three operations. Note that after $n - 1$ Union operations only one set may remain. Also, $m \geq n$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.40 Connectivity of an undirected graph. An application of disjoint sets is in finding the connected components of an undirected graph G.

```
Connected-Components(G)
1. for each vertex u in V
2.  do MakeSet(u)
3. for each edge (u,v) in E
4.  do if Find(u) != Find(v)
5.        then Union(u,v)

Same-Components(u,v)
1.  if Find(u) == Find(v)
2.        then return(TRUE)
3.        else return(FALSE)
```

What is the running time of the two algorithms?
case 1. Adjacency list representation of the graph
case 2. Adjacency matrix representation of the graph.

What is the running time of MakeSet, Union, Find? In order to answer the question about the running time of `Connected-Components` and `Same-Component` we need to determine the running time of these three operations. This depends on the implementation of the data structure that supports disjoint sets.

Note 1. The number of MakeSet operations performed in line 2 of `Connected-Components` is n, the number of elements i.e. the number of vertices of the graph. The number of Find(u) or Find(v) operations depends on the number of edges m. It could be be m or $2m$ for the edges and $2m$ or $4m$ for the Find ops. The number of iteration of line 3 depends on the representation of the graph. It might be $O(n^2)$ or just $O(m)$.

Note 2. The running time of Same-Components is the running time of Find (or twice of it)!

3.41 Linked-list implementation. We represent each set by a doubly linked list. Its elements are pointed by an array of n elements, the elements themselves. There is a head and tail pointer to the list and of course previous and next pointers from each element member of a linked list. In addition there is a representative pointer pointing to the representative of the set (list). In addition there is an array of n elements with several attributes. One attribute is a pointer pointing to the element itself that appears in some linked list (set). That's how we retrieve representative pointer information for the element. Two more attributes are head and tail linked list pointer information. This information is valid if the element of the array is the representative element of the linked list; otherwise it contains garbage, if not properly cleaned up during Union operations. Moreover an additional attribute contains the length of the list for which this element is a representative.

3.42 MakeSet. MakeSet(x) runs in time $O(1)$ as it creates a list with one element by creating the element x initializes its previous and next pointers and make its rep pointed pointing to it. It then goes to the array of n elements, previously created, and it sets its head and tail pointers to point to the element itself, and makes its size equal to 1.

3.43 Find(x). Find(x) runs in $\Theta(1)$ as we return the pointer to x's representative. We first find x in the array, follow the pointer to the element in the linked list and then the representative pointer of it. If the element is its own representative we know the head/tail/size attributes are useful and not garbage.

3.44. Union(x,y). Union(x,y) is a bit more difficult to implement. We first run `a=Find(x)` and `b=Find(y)` and if the representative elements a, b are indeed different i.e. $a \neq b$ then and only then do we need to perform a union of `Union(a,b)`. For this we first find size info for a, b. We are going to modify the smaller of the two linked lists' information. Thus if $a < b$, b's information will remain almost intact. To merge the two linked lists into one, head/tail and some previous/next pointers get updated the head and tail pointer of b gets updated and a's gets garbaged. The new size attributes of b gets updated as well. Overall the running time is the size of the smaller of the two lists/sets/

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

3.45 Running time of all Union Operations. If all MakeSet operations are performed we have n sets for n elements. The maximum number of Union operations that we can have is $n-1$ since every union reduces the number of sets by one. The costs of Union operations can be is $O(n)$ and can be $\Theta(n)$. (Consider the case were the two sets contains roughly $n/2$ elements.)

```
MakeSet(x1).... MakeSet(xn);
Union(x1,x2);Union(x2,x3);Union(x3,x4); etc
```

The **collective running time of all at most** $n-1$ **Union operations** however is not $Theta(n^2)$ but only $O(n \lg n)$. This is because we always update the smaller of the two lists. Every time we update the smaller list, the new list gets doubled in size. How many times can we double a list's size before it reaches n (i.e. it contains all the elements)? Since we start with a list of size 1 this can be no more than $\lceil \lg n \rceil$. Thus every rep pointer can be updated no more that $O(\lg n)$ times. Since there are n elements this means that the time for all the updates overall the elements is $O(n \lg n)$. Thus even if a single union operation is of cost $Theta(n)$ the collection of at most $n-1$ Union operations have cost $O(n \lg n)$ rather than $Theta(n^2)$.

3.46 Amortized running time vs Collective running time. One Union Operation can be $O(n)$ or $\Theta(n)$. Yet $n-1$ Union operations can and will be $O(n \lg n)$. Thus the **amortized cost of a Union** is total cost over the number of Union operations. This is $O(\lg n)$; note it is amortized cost per Union not worst-case (aka running-time) costs.

For m operations with n MakeSet operations, at most $n-1$ Unions operations the total cost is thus $O(m + n \lg n)$ since all other operations (MakeSet, Find) are $O(1)$ each.

(c) Copyright A. Gerbessiotis. All rights reserved.

Counting theory answers the question **How many?** without enumerating the many.

How many n-digit (distinct) binary numbers are there?

Rule of sums. The number of ways to choose an element from two disjoint sets is the sum of the sizes of the two sets.

Thus, $|A \cup B| = |A| + |B|$.

Rule of product. The number of ways to choose an ordered pair (a, b) is the number of ways to choose a times the number of ways to choose b.

Thus, $|A \times B| = |A| \cdot |B|$.

A string is sequence of 0 or more characters from a set S. Let $|S|$ be the cardinality of S. The length of a string is the number of characters in it. A string of length 0 is an empty string.

The number of strings of length k is $|S|^k$ (by the rule of product) as we have k ways to pick first element, k ways to pick second and so on.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A **permutation** of a finite set S is an ordered sequence of all the elements of the set S with each element appearing exactly once. Let $S = \{1, 2, 3\}$. Then the permutations on these three elements are 123,132,213,231,312,321.

There are $n! = n(n-1)(n-2)\dots 2\ 1$ permutations on n elements.

Proof. By induction. $n = 1$ trivial. Let $n-1$ elements. Assume ind. hypothesis. Then we have $(n-1)!$ permutations on these elements.

Let us be given n elements. In a permutation we have n choices for the first element. Let us fix this choice to be one of the n elements, say element called a. Then for the choice of a as the first element the remaining $n-1$ elements give rise to $(n-1)!$ permutations on the remaining $n-1$ elements which along with a as the first give a permutation on n elements. a can be any of n elements. Therefore number of permutations is $n(n-1)! = n!$.

A k-permutation is a ordered sequence of length k of elements of a set S with no element appearing more than once in the sequence. An ordinary permutation is an n-permutation.

The number of k permutations on a set of size n is $n(n-1)(n-2)\dots(n-k+1) = n!/(n-k)!$ as there are n choices for the first element, $(n-1)$ choices for the second, $(n-2)$ for the third and so on.

A k-combination of an n-set S is simply a k-subset of S The number of k combination of S is $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$.

Intuitive Proof. The number of k-permutations is $n!/(n-k)!$. Two k-permutations (eg 12 21 for 2-permutations) correspond to the same k-combination if they are permutations of one another. There are $k!$ such permutations for a given k subset. Therefore we need to divide the number of k permutations by $k!$ to get the number of k-combinations.

(c) Copyright A. Gerbessiotis. All rights reserved.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n!}{(n-k)!\,k!} = \binom{n}{n-k}$$

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

Set $x = y = 1$.

$$2^n = \sum_{k=0}^{n} \binom{n}{k}$$

Since $k! \geq (k/e)^k$, $1 \leq k \leq n$ we have

$$(n/k)^k \leq \binom{n}{k} \leq (en/k)^k$$

$$\binom{n}{an} = 2^{nH(a)}$$

where $H(a)$, $a < 1$, is the entropy function (see Problem 5 of Problem Set 1 for some properties), defined in the textbook, page 1187,
$H(a) = -a \cdot \lg a - (1-a) \cdot \lg(1-a)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A sample space S is a set whose elements are called **elementary events** (as opposed to an event). Each elementary event is a possible outcome of an experiment.

Experiment: Flip a coin twice.

Set S $= \{TT, HT, TH, HH\}$.

Events: Subsets of sample Space eg All heads event or All tails event

Event S is called certain event and $\emptyset$ is called null event. Two events A and B are mutually exclusive if their intersection is the null event.

A probability distribution Pr on a sample space S is a mapping from events of S to real numbers such that

1. $Pr(A) \geq 0$ for any event A.

2. $Pr(S) = 1$.

3. $Pr(A \cup B) = Pr(A) + Pr(B)$ if A, B exclusive.

Pr(A) is the probability of event A.

$$Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B) \leq Pr(A) + Pr(B)$$

$$Pr(A) = \sum_{s \in A} Pr(s)$$

Uniform probability: $Pr(s) = 1/|S|$.

Examples. Throw two dice.

1. What's the probability they are the same?
2. What's the probability we get 7?
3. What's the probability we get 1?

Flip a coin n times.
1. What's the probability of getting n heads?
2. What's the probability of getting n tails?
3. What's the probability of getting k heads?

Conditional probability

$$Pr(A/B) = \frac{Pr(A \cap B)}{Pr(B)}$$

For two **INDEPENDENT EVENTS**

$$Pr(A \cap B) = Pr(A) \cdot Pr(B)$$

and then

$$Pr(A/B) = Pr(A).$$

(c) Copyright A. Gerbessiotis. All rights reserved.

A discrete rv X is a function from a sample space S to the real numbers. It associates a real number to every outcome of an experiment.

Let X be a rv and a real number x, we define $X = s$ to be $\{s \in S : X(s) = x\}$. Thus $Pr(X = x) = \sum_{\{s \in S: X(s)=x\}} Pr(s)$. The function $f(x) = Pr(X = x)$ is the probability density function of rv X.

For the dice problem Pr(s)=1/36 for any s. Let X be the maximum of the two throws. Then Pr(X=3) = 5/36.

The expected value or expectation or mean of a random variable X is

$$E[X] = \sum_x xPr(X = x)$$

$$E[X + Y] = E[X] + E[Y]$$

$$E[g(X)] = \sum_x g(x)Pr(X = x)$$

The variance of a random variable X.

$$var(X) = E[X^2] - E^2[X].$$

$$var[X + Y] = var[X] + var[Y]$$

provided that X and Y are independent.

Example
HT games: earn 5USD if H, lose 3USD if T. What's the expected profit/earning when throwing two coins?