CSE 21
Practice Final Exam
Winter 2016

1. **Sorting and Searching.** Give the number of comparisons that will be performed by each sorting algorithm if the input list of length n happens to be of the form $n, 1, 2, ..., n-3, n-2, n-1$ (i.e., sorted except the largest element is first). On the real exam, you would be given pseudocode for the algorithms, though it is a very good idea to be comfortable with how the algorithms work to save time on the exam. For now, you can refer to the textbook for pseudocode.

   (a) MinSort (SelectionSort)
   Solution: $\frac{n(n-1)}{2}$
   MinSort does the same amount of work on all inputs. It does $n-1$ comparisons to find the minimum of the list before swapping it to the front. Then it does $n-2$ comparisons on the next pass to find the minimum of the remaining elements, and so on, until it eventually does 1 comparison to find the minimum of the last remaining two elements. The total number of comparisons is $1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$.

   (b) BubbleSort
   Solution: $\frac{n(n-1)}{2}$
   BubbleSort does the same amount of work on all inputs. It does $n-1$ comparisons on the first pass, then $n-2$ comparisons on the next pass, and so on. The total number of comparisons is $1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$.

   (c) InsertionSort
   Solution: $\frac{n(n-1)}{2}$
   On this input, InsertionSort will do one comparison when $j=2$ (comparing a_2 with a_1). It will then do two comparisons when $j=3$, and so on, up to $n-1$ comparisons when $j=n$. This is because each new number a_j being inserted into the sorted list is greater than all but one of the numbers that have been inserted before it, and so it will need to be compared to each of the other elements in the list so far. The total number of comparisons is $1+2+3+\cdots+(n-1) = \frac{n(n-1)}{2}$.

2. **Order notation.**

   For each of the following pairs of functions f and g, is $f \in O(g)$? Is $f \in \Omega(g)$? Is $f \in \Theta(g)$?

   (a) $f(n) = n, g(n) = 2^{\lfloor \log n \rfloor}$
   Solution: Is $f \in O(g)$? YES
   Is $f \in \Omega(g)$? YES
   Is $f \in \Theta(g)$? YES
   First, observe that $n = 2^{\log n}$. So, $2^{\lfloor \log n \rfloor} \approx 2^{\log n} = n$. This suggests the Θ relationship for these functions, because they are always very close to each other. Let's pick constants to prove this. For all $n > 1$ we have

   $$\log n - 1 < \lfloor \log n \rfloor \leq \log n$$
   $$2^{\log n - 1} < 2^{\lfloor \log n \rfloor} \leq 2^{\log n}$$
   $$1/2n < 2^{\lfloor \log n \rfloor} \leq n$$

   By the definition of Θ, this proves $f \in \Theta(g)$. Since $f \in \Theta(g)$, this also implies $f \in O(g)$ and $f \in \Omega(g)$.

   (b) $f(n) = \log(n^4), g(n) = (\log n)^4$.
   Solution: Is $f \in O(g)$? YES
   Is $f \in \Omega(g)$? NO
   Is $f \in \Theta(g)$? NO

Again, we first simplify the functions and consider their relationship intuitively. Using properties of logs, $f(n) = \log(n^4) = 4\log n$. Since $g = (\log n)^4$, this suggests that eventually $g(n) > f(n)$. We'll use the limit test to confirm this:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4\log n}{(\log n)^4} = \lim_{n\to\infty} \frac{4}{(\log n)^3} = 0.$$

This proves that $f \in O(g)$ but $f \notin \Theta(g)$ and $f \notin \Omega(g)$.

(c) $f(n) = n^2, g(n) = 1000n^2 + 2^{100}$
Solution: Is $f \in O(g)$? YES
Is $f \in \Omega(g)$? YES
Is $f \in \Theta(g)$? YES
Considering the functions informally, we see that both functions are quadratics, dominated by the n^2 term. This suggests that they are equivalent, and we should attempt to prove the Θ relationship. The limit test says:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{1000n^2 + 2^{100}}{n^2} = \lim_{n\to\infty} 1000 + \frac{2^{100}}{n^2} = 1000.$$

Since the result is a nonzero constant, we know $f \in \Theta(g)$. Since $f \in \Theta(g)$, this also implies $f \in O(g)$ and $f \in \Omega(g)$.

(d) $f(n) = 2^{2n}, g(n) = 2^n$.
Solution: Is $f \in O(g)$? NO
Is $f \in \Omega(g)$? YES
Is $f \in \Theta(g)$? NO
Comparing the functions informally, we have $2^{2n} = 4^n > 2^n$ for any n. So this suggests the Ω relationship, which we can confirm with the following limit test.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^{2n}}{2^n} = 2^n = \infty$$

This proves $f \in \Omega(g)$ but $f \notin \Theta(g)$ and $f \notin O(g)$.

(e) $f(n) = n!, g(n) = n^n$.
Solution: Is $f \in O(g)$? YES
Is $f \in \Omega(g)$? NO
Is $f \in \Theta(g)$? NO
We can see that g grows strictly faster than f if we write $f(n) = n! = (n)(n-1)(n-2)\dots(1)$ and $g(n) = n^n = (n)(n)(n)\dots(n)$. Both expressions have the same number of terms (n terms) and each term in f, the corresponding term in g is at least as large. In particular, the last term in f is 1 whereas the last term in g is n. This means that for large values of n, g will outpace f by a factor that grows with n, so they are not in the same Θ class.

3. **Iterative algorithms.**

In the following problem, a polynomial $p(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ of degree $n-1$ is specified by a list of its n coefficients $A = a_0, \dots, a_{n-1}$. We want to evaluate $p(v)$ for some real number v. Here is an iterative algorithm to do so:

```
Evaluate($A$ : list of $n$ coefficients, $v$: a real number)

1. $y := 0$
2. for $i := 1$ to $n$
3.     $y := y * v + a_{n-i}$
4. return $y$
```

(a) Prove the following loop invariant for this algorithm: After i times through the **for** loop,

$$y = a_{n-i} + a_{n-i+1}v + a_{n-i+2}v^2 + \cdots + a_{n-1}v^{i-1}.$$

Solution: We will prove this by induction on i. For the base case, take $i = 1$. After the first iteration of the loop, $y = 0 * v + a_{n-1} = a_{n-1}$. This matches the form $y = a_{n-i} + a_{n-i+1}v + a_{n-i+2}v^2 + \cdots + a_{n-1}v^{i-1}$, which reduces to $a_{n-1}v^0$ when $i = 1$.

Now suppose that the invariant is true after $i - 1$ times through the loop. Then, before the ith iteration of the loop, we have

$$\begin{aligned} y &= a_{n-(i-1)} + a_{n-(i-1)+1}v + a_{n-(i-1)+2}v^2 + \cdots + a_{n-1}v^{(i-1)-1} \\ &= a_{n-i+1} + a_{n-i+2}v + a_{n-i+3}v^2 + \cdots + a_{n-1}v^{i-2}. \end{aligned}$$

In the ith iteration of the loop, we multiply y by v and add a_{n-1}. This means that after the ith iteration of the loop, we have

$$\begin{aligned} y &= (a_{n-i+1} + a_{n-i+2}v + a_{n-i+3}v^2 + \cdots + a_{n-1}v^{i-2}) * v + a_{n-i} \\ &= a_{n-i} + a_{n-i+1}v + a_{n-i+2}v^2 + \cdots + a_{n-1}v^{i-1} \end{aligned}$$

as desired. This proves that for all $i \geq 1$, the loop invariant holds.

(b) Conclude from the loop invariant that the algorithm is correct.

Solution: Since we have proven that for all $i \geq 1$, the loop invariant is true, we know in particular that it is true when $i = n$. That is, after n times through the **for** loop,

$$\begin{aligned} y &= a_{n-n} + a_{n-n+1}v + a_{n-n+2}v^2 + \cdots + a_{n-1}v^{n-1} \\ &= a_0 + a_1v + a_2v^2 + \cdots + a_{n-1}v^{n-1} \\ &= p(v). \end{aligned}$$

Since the algorithm returns y, which is equal to $p(v)$ and the algorithm was supposed to evaluate $p(v)$, it is correct.

(c) Describe the running time of this algorithm in Θ notation, assuming that arithmetic operations take constant time. Justify your answer.

Solution: This algorithm's runtime is $\Theta(n)$. Line 3 takes constant time and is in a for loop that runs n times, so lines 2 and 3 take time $\Theta(n)$. Lines 1 are 4 are constant time, or $\Theta(1)$, and when we do operations in sequence, the time is determined by the biggest of them, which is $\Theta(n)$.

4. **Recursive algorithms.**

In the following problem, a polynomial $p(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ of degree $n - 1$ is specified by a list of its n coefficients $A = a_0, \ldots, a_{n-1}$. We want to evaluate $p(v)$ for some real number v. Here is a recursive algorithm to do so:

```
RecEvaluate($A$ : list of $n$ coefficients, $v$: a real number)
1. if $n = 1$ then
2.     return $a_0$
3. $B := a_1, a_2, \ldots, a_{n-1}$
4. $y :=$ RecEvaluate($B, v$)
5. $y := y * v + a_0$
6. return $y$
```

(a) Prove by induction on n that RecEvaluate(A, v) returns $p(v)$.

Solution: We want to show that for any list $A = a_0, \ldots, a_{n-1}$ of length n, RecEvaluate(A, v) returns $a_0 + a_1v + a_2v^2 + \cdots + a_{n-1}v^{n-1} = p(v)$. We will prove this by induction on n, the length of A.

For the base case, suppose $n = 1$. Then the algorithm returns a_0 in line 2, which is $a_0 + a_1v + a_2v^2 + \cdots + a_{n-1}v^{n-1}$ when $n = 1$, so the algorithm is correct.

Now for the induction hypothesis, suppose that for any list $L = \ell_0, \ell_1 \ldots, \ell_{n-1}$ of length $n-1$, RecEvaluate(L, v) returns $\ell_0 + \ell_1 v + \ell_2 v^2 + \cdots + \ell_{n-2}v^{n-2}$. Then consider the behavior of RecEvaluate(A, v) for a list $A = a_0, \ldots, a_{n-1}$ of length n. In line 3, the algorithm creates a new list $B = a_1, a_2, \ldots, a_{n-1}$ of length $n - 1$ by removing the last element of A. Then in line 4, RecEvaluate(B, v) is called. Since this is a list of size $n - 1$, the inductive hypothesis says that this recursive call returns $a_1 + a_2v + a_3v^2 + \cdots + a_{n-1}v^{n-2}$. The algorithm stores this as y then in line 5, multiplies y by v and adds a_0. Thus, after line 5,

$$\begin{aligned} y &= (a_1 + a_2v + a_3v^2 + \cdots + a_{n-1}v^{n-2}) * v + a_0 \\ &= a_0 + a_1v + a_2v^2 + \cdots + a_{n-1}v^{n-1} \\ &= p(v). \end{aligned}$$

In line 6, this value of y is returned, so the algorithm returns $p(v)$ as desired.

(b) Give a recurrence for the time this algorithm takes on an input of size n, assuming that arithmetic operations take constant time. Explain your reasoning.

Solution: $T(1) = c,\ T(n) = T(n - 1) + d$, where c and d are constants

Note that this also assumes that it takes constant time to remove an element from a list. Under these assumptions, everything except line 4 takes constant time, which together we call d. Line 4 is a recursive call to the same function but with input of size one smaller, so it takes time $T(n - 1)$. The base case $T(1) = c$ comes from the first two lines. When $n = 1$, all the algorithm does is return a_0, which takes constant time.

(c) Solve this recurrence to determine the running time of this algorithm in Θ notation.

Solution: We will use the unraveling method.

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= T(n-2) + 2d \\ &= T(n-3) + 3d \\ &\vdots \\ &= T(n-k) + kd \\ &\vdots \\ &= T(1) + (n-1)d \\ &= c + (n-1)d \\ &= nd + (c-d) \end{aligned}$$

We choose a value of $k = n - 1$ to end at the base case of $T(1)$.

The result is a linear function in n, $nd + (c - d)$, so the algorithm is $\Theta(n)$. Notice that it's the same order as the iterative version in the previous problem, so neither is substantially more efficient than the other.

5. **Best and worst case.**

(a) Describe, in English and in pseudocode, an algorithm to determine whether a bitstring of length n contains the substring 101. For example, the string 1101 contains the substring 101 but the string 1001 does not.

Solution: The algorithm consists of a single pass through the bitstring of length n. Check each substring of length 3 to see if it matches the pattern 101. If so, return true, and otherwise, if you make it through the entire bitstring without finding a match, return false.

Find101($b_1b_2 \ldots b_n$: a bitstring of length n)

```
1. for $i := 1$ to $n - 2$
2.     if $b_i = 1$ AND $b_{i+1} = 0$ AND $b_{i+2} = 1$ then
3.         return $true$
4. return $false$
```

(b) Which bitstrings of length n would be best-case inputs to your algorithm? Describe the best-case time of your algorithm in Θ notation.

Solution: All strings that start with 101 as the first three bits would be best-case inputs. In this case, the running time is constant, or $\Theta(1)$ since only one iteration of the for loop would occur before returning *true*.

(c) Which bitstrings of length n would be worst-case inputs to your algorithm? Describe the worst-case time of your algorithm in Θ notation.

Solution: All strings that do not contain 101, or all strings where the first instance of 101 is in the last three bits, would be worst-case inputs. In both of these cases, the for loop executes the full $n - 2$ times, doing constant work inside the body, so the running time is $\Theta(n)$.

6. **Representing problems as graphs.** You have a system of n variables representing real numbers, $X_1, \ldots, X_n$. You are given a list of inequalities of the form $X_i < X_j$ for some pairs i and j. You want to know whether you can deduce with certainty from the given information that $X_1 < X_n$.

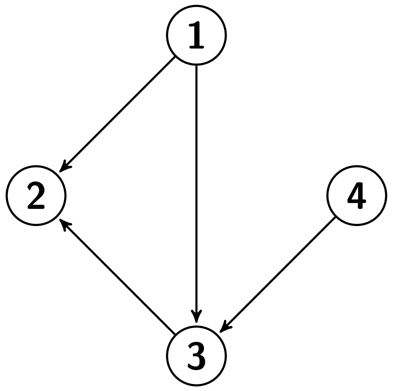
For example, say n is 4. A possible input is the list of inequalities $X_1 < X_2, X_1 < X_3, X_4 < X_3$ and $X_3 < X_2$. Does it follow that $X_1 < X_4$?

(a) Give a description of a directed graph that would help solve this problem. Be sure to define both the vertices and edges in terms of the variables and known inequalities.

Solution: There will be n vertices labeled $1, 2, \ldots, n$ which represent the variables. Connect vertex i to vertex j by a directed edge from i to j if and only if $X_i < X_j$ is one of the inequalities in the list. This helps solve the problem because we can deduce that for any a, b, $X_a < X_b$ if there exists a path from a to b in the graph.

(b) Draw the graph you described for the example above. Does $X_1 < X_4$ follow? Why or why not?

Solution:



It doesn't follow that $X_1 < X_4$ because there is no path from 1 to 4 in the graph.

(c) Say which algorithm from lecture we could use on such a graph to determine whether $X_1 < X_n$ follows from the known inequalities.

Solution: We could use the Graph Search algorithm to decide whether or not n is reachable from 1. If it is, then $X_1 < X_n$. Otherwise we can't be sure of the relationship between X_1 and X_n.

7. **DAGs and trees.** For each statement, either prove that it is true or give a counterexample to show that it is false.

(a) Every tree on n nodes has exactly $n - 1$ edges.

Solution: We can prove this using induction on n, the number of nodes in the graph. For the base case, when $n = 1$, a tree has no edges.

For the induction hypothesis, assume that any tree of k nodes has $k-1$ edges. We will prove that any tree T of $k+1$ nodes has k edges. Let u be a degree one node, or a leaf of the tree. We proved in class that all trees have at least one node which has degree one. Consider the graph $T' = T - u$, which is obtained by removing u and its only incident edge from T. T' is still a tree and it has k nodes. So, it must have $k-1$ edges from induction hypothesis. Now, if we add u back, we are adding one more edge. So, number of edges in T is k. This proves the statement by induction.

(b) Every graph with exactly $n-1$ edges is a tree.

Solution: This statement is false. A counter-example would be for $n = 4$, a triangle with an isolated vertex. That is, $V = v_1, v_2, v_3, v_4$ with the set of edges, $E = (v_1, v_2), (v_3, v_2), (v_1, v_3)$. Here, $n = 4$, there are $n - 1 = 3$ edges, but the graph is not a tree because it has a cycle.

However, the statement is correct if the graph is connected. That is, any *connected* graph of n vertices with $n-1$ edges is always a tree. See if you can try to prove it [Hint: Prove that there should be a leaf node and use induction].

8. **Counting.**

(a) How many 5-card hands can be formed from an ordinary deck of 52 cards if exactly two suits are present in the hand?

Solution: There are two disjoint possibilities to have exactly two suits:

- 3 cards of one suit + 2 cards of another:
  i. Select a suit (s_1) for 3 cards - $\binom{4}{1}$
  ii. Select a suit (s_2) for 2 cards - $\binom{3}{1}$
  iii. Select 3 cards from suit s_1 - $\binom{13}{3}$
  iv. Select 2 cards from suit s_2 - $\binom{13}{2}$

  The number of ways is $\binom{4}{1} \cdot \binom{3}{1} \cdot \binom{13}{3} \cdot \binom{13}{2}$.
- 4 cards of one suit + 1 card of another suit: Using similar counting as the previous case, the number of ways is $\binom{4}{1} \cdot \binom{3}{1} \cdot \binom{13}{4} \cdot \binom{13}{1}$.

Therefore, the total number of ways is: $\binom{4}{1} \cdot \binom{3}{1} \cdot \binom{13}{3} \cdot \binom{13}{2} + \binom{4}{1} \cdot \binom{3}{1} \cdot \binom{13}{4} \cdot \binom{13}{1}$.

(b) In any bit string, the longest consecutive run length is the maximum number of consecutive 1's or consecutive 0's in the string. For example, in the string 1101000111, the longest consecutive run length is 3. How many bit strings of length 10 have a longest consecutive run length of 6?

Solution: Let $c_1, c_2, c_3, \ldots, c_{10}$ be the 10 spots for 10 digits. We need to fill these spots with 0, 1 such that there are exactly 6 consecutive spots with same number. There are two cases:

- The six consecutive spots are at the beginning or the end. That is, starting location of six consecutive spots is either c_1 or c_5. All the consecutive spots need to take same value and the adjacent spot should take the opposite value. This can be done in two ways, either a run of 0s adjacent to a 1 or a run of 1s adjacent to a 0. The remaining three spots can take any of the two values each (in 2^3 ways). So, the total number of possibilities is $2 \cdot 2 \cdot 2^3 = 32$, where the first 2 counts whether we start at c_0 or c_5, the second 2 counts whether we have a run of 0s or a run of 1s, and the 2^3 counts the ways to fill the remaining three spots.
- The six consecutive spots are in the middle. That is, starting location of size consecutive spots is one of c_2, c_3, c_4. All the consecutive spots need to take same value and the two adjacent spots should take the opposite values. This can be done in two ways, either a run of 0s adjacent to 1s or a run of 1s adjacent to 0s. The remaining two spots can take any of the two values each (in 2^2 ways). So, the total number of possibilities is $3 \cdot 2 \cdot 2^2 = 24$, where the 3 counts whether we start at c_2, c_3, or c_4, the 2 counts whether we have a run of 0s or a run of 1s, and the 2^2 counts the ways to fill the remaining two spots.

Since these cases don't overlap, we add them to get that the total number of ways is 56.

(c) A software company assigns its summer interns to one of three divisions: design, implementation, and testing. In how many ways can a group of ten interns be assigned to these divisions if each division needs at least one intern?

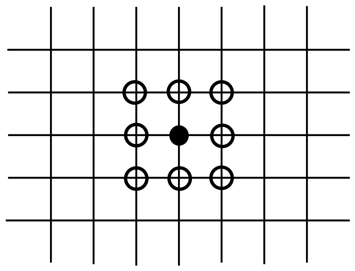
Solution: Each person can be assigned to one of the three divisions, so for 10 people the total number of ways of assigning interns to divisions is 3^{10}.

Now let's count the number of ways that all of them go to just two particular divisions, say design and implementation. We have to exclude this case from our count because testing will be empty. There are two possible assignments for each person, so the number of possibilities is 2^{10}. Similary, we have to account for other two cases where everyone goes to implementation and testing or everyone goes to design and testing. In total, there are $3 \cdot 2^{10}$ ways that the interns can be assigned to only two divisions.

Finally, we must compute the number of ways in which interns are assigned to only one division. There are 3 ways to select which single division will have all interns, and this is the only choice we can make.

Applying the principle of inclusion-exclusion, the number of ways the interns can be assigned to the divisions so that each division has at least 1 intern is given by the total number of possible assignments, minus the number of ways to assign interns to only 2 divisions, plus the number of ways to assign interns to 3 divisions, which is $3^{10} - 3 \cdot 2^{10} + 3$.

9. **Encoding and decoding.** Suppose you are standing on the coordinate plane at the origin $(0, 0)$. You will take a walk for n steps, each step moving one unit horizontally, vertically, or diagonally from your current position. The figure uses open circles to show where your next step can be, if your current position is on the shaded circle. You are allowed to revisit points on your walk.



(a) How many walks of n steps are possible?

Solution: 8^n because at each of the n steps, we can go in any of 8 directions.

(b) How many bits (0's and 1's) are required to represent a walk of n steps? Simplify your answer.

Solution: $\log 8^n = n \log 8 = 3n$

(c) Describe how to encode a walk of n steps using the number of bits you gave in part (b). Illustrate your description with an example.

Solution: Assign each of the eight directions a different 3-bit sequence. For example,

North	000
Northeast	001
East	010
Southeast	011
South	100
Southwest	101
West	110
Northwest	111

Then to encode a walk, replace each step with its corresponding bit sequence. For example, a walk of 4 steps that goes North, Southwest, West, West would be encoded as 000101110110.

10. **Probability.**

(a) I have 10 shirts, 6 pairs of pants, and 3 jackets. Every day I dress at random, picking one of each category. What is the probability that today I am wearing at least one garment I was wearing yesterday?

Solution: Let us instead compute the complement, since this is a question about whether there exists a match between today's clothing and yesterday's.

Let A be the event that I am wearing all different clothes from yesterday.

We can see that $P(A) = (9/10) * (5/6) * (2/3)$. The reason for this is that today I can wear any of the other garments that I didn't wear yesterday.

Thus the probability of wearing at least one of the same garments as yesterday is $1 - P(A) = 1 - (9/10) * (5/6) * (2/3)$.

(b) A permutation of size n is a rearrangement of the numbers $\{1, 2, \ldots, n\}$ in any order. A *rise* in a permutation occurs when a larger number immediately follows a smaller one. For example, if $n = 5$, the permutation 1 3 2 4 5 has three rises. What is the expected number of rises in a permutation of size n?

Solution: We want $E[X]$ where X is the number of rises in the permutation. However, the number of rises in the permutation is the total number of rises that start at each position. So for $i = 1$ to $n-1$, define X_i to be 1 if there is a rise starting at position i and 0 otherwise. Thus, using linearity of expectation,

$$E[X] = E[X_1 + X_2 + \cdots + X_{n-1}] = E[X_1] + E[X_2] + \cdots + E[X_{n-1}].$$

If we think of a permutation as a random rearrangement of the numbers from 1 to n, no position in a permutation is more likely to have a rise than any other. This means each X_i has the same distribution and the same expected value. With probability 1/2 there is a rise at position i and with probability 1/2 there is a fall, so $E[X_i] = 1 * 1/2 + 0 * 1/2 = 1/2$. Thus, $E[X] = (n-1) * 1/2$.

(c) There are n teams in a sports league. Over the course of a season, each team plays every other team exactly once. If the outcome of each game is a fair random coin flip, and there are no ties, what is the probability that some team wins all of its games?

Solution: As there n teams in the league, each team plays $n-1$ games, so for a team to win all of its games it has to win all $n-1$ games. Every game to be won has a probability of 1/2.

Define i events, E_i is the event that team i is undefeated. The event that some team is undefeated is $E_1 + E_2 + \cdots + E_n$, so we want $P(E_1 + E_2 + \cdots + E_n)$. But the events E_i are disjoint subsets of the sample space since we can't have two undefeated teams, which means we can add their probabilities together and instead compute $P(E_1) + P(E_2) + \cdots + P(E_n)$, by the sum rule.

The probability that team i wins all of its games is $P(E_i) = (1/2)^{n-1}$. As there are n teams the total probability that some team is undefeated is $n * (1/2)^{n-1}$.

(d) Suppose that one person in 10,000 people has a rare genetic disease. There is an excellent test for the disease; 99.9% of people with the disease test positive and only 0.02% who do not have the disease test positive. What is the probability that someone who tests positive has the genetic disease? What is the probability that someone who tests negative does not have the disease?

Solution: Define the event F to mean "Has the disease" and E to mean "Tests positive." Then F^C is "Doesn't have the disease" and E^C is "Tests negative."

We are given that

$$\begin{aligned}
P(E|F) &= 0.999 \\
P(E^C|F) &= 0.001 \\
P(E|F^C) &= 0.0002 \\
P(E^C|F^C) &= 0.9998 \\
P(F) &= 1/10000 \\
&= 0.0001 \\
P(F^C) &= 0.9999
\end{aligned}$$

We will use Bayes' Theorem, which says that

$$P(F|E) = \frac{P(E|F)P(F)}{P(E|F)P(F) + P(E|F^C)P(F^C)}.$$

Plugging in the values above gives that the probability that someone who tests positive actually has the disease is

$$\begin{aligned}
P(F|E) &= \frac{P(E|F)P(F)}{P(E|F)P(F) + P(E|F^C)P(F^C)} \\
&= \frac{0.999 * 0.0001}{0.999 * 0.0001 + 0.0002 * 0.9999} \\
&= 0.3331332533.
\end{aligned}$$

Thus, there is approximately a one third chance of having the disease even if you test positive, since the disease is so rare.
Now if we complement each event in Bayes' Theorem we get

$$P(F^C|E^C) = \frac{P(E^C|F^C)P(F^C)}{P(E^C|F^C)P(F^C) + P(E^C|F)P(F)},$$

which we can use to tell us the probability that someone who tests negative does not have the disease. Plugging in the values above gives

$$\begin{aligned}
P(F^C|E^C) &= \frac{P(E^C|F^C)P(F^C)}{P(E^C|F^C)P(F^C) + P(E^C|F)P(F)} \\
&= \frac{0.9998 * 0.9999}{0.9998 * 0.9999 + 0.001 * 0.0001} \\
&= 0.99999989997.
\end{aligned}$$

Thus, it is almost certain that you don't have the disease if you test negative, since the disease is so rare.