

Decision Trees

Introduction

In many situations one needs to make a series of decisions. This leads naturally to a structure called a “decision tree,” which we will define shortly. Decision trees provide a geometrical framework for organizing the decisions. The important aspect is the decisions that are made. Everything we do in this chapter could be rewritten to avoid the use of trees; however, trees

- give us a powerful intuitive basis for viewing the problems of this chapter,
- provide a language for discussing the material,
- allow us to view the collection of all decisions in an organized manner.

We’ll begin by studying some basic concepts concerning decision trees. Next we’ll relate decision trees to the concepts of “ranking” and “unranking,” ideas which allow for efficient computer storage of data which is *a priori* not indexed in a simple manner. Finally we’ll study decision trees which contain some bad decisions; i.e., decisions that do not lead to solutions.

Decision trees are particularly useful in the “local” study of recursive procedures. In Sections 7.3 (p. 210) and 9.3 (p. 259) we’ll apply this idea to various algorithms.

Make sure you’re familiar with the concepts in Chapter 2, especially the first section, the definition of a permutation and the bijections in Theorem 2.4 (p. 52).

3.1 Basic Concepts of Decision Trees

Decision trees provide a method for systematically listing a variety of functions. We’ll begin by looking at a couple of these. The simplest general class of functions to list is the entire set \underline{n}^k . We can create a typical element in the list by choosing an element of \underline{n} and writing it down, choosing another element (possibly the same as before) of \underline{n} and writing it down next, and so on until we have made k decisions. This generates a function in one line form sequentially: First $f(1)$ is chosen, then $f(2)$ is chosen and so on. We can represent all possible decisions pictorially by writing down the decisions made so far and then some downward edges indicating the possible choices for the next decision.

The lefthand part of Figure 3.1 illustrates this way of generating a function in $\underline{2}^3$ sequentially. It’s called the *decision tree* for generating the functions in $\underline{2}^3$. Each line in the left hand figure is labeled with the choice of function value to which it corresponds. Note that the labeling does not completely describe the corresponding decision—we should have used something like “Choose 1 for

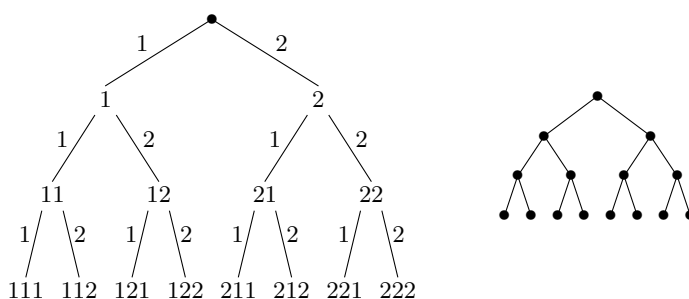


Figure 3.1 (a) The decision tree for all functions in 2^3 ; that is, 3-long lists from $\{1, 2\}$ with repeats allowed is on the left. We've omitted commas in the lists. Functions are written in one line notation. (b) The underlying tree with all labels removed is on the right.

the value of $f(2)$ " instead of simply "1" on the line from 1 to 11. At the end of each line is the function that has been built up so far. We've omitted commas, so 211 means 2,1,1.

To the right of the figure for generating the functions is the same structure without any labels. The dots (\bullet) are called *nodes* or *vertices* and the lines connecting them are called *edges*. Sometimes it is more convenient to specify an edge by giving its two end points; for example, the edge from 1 to 12 in the figure can be described uniquely as $(1,12)$. The nodes with no edges leading down are called the *leaves*. The entire branching structure is called a *tree* or, more properly, an *ordered rooted tree*. The topmost node is called the *root*.

In this terminology, the labels on the vertices show the partial function constructed so far when the vertex has been reached. The edges leading out of a vertex are labeled with all possible decisions that can be made next, given the partial function at the vertex. We labeled the edges so that the labels on edges out of each vertex are in order when read left to right. The leaves are labeled with the finished functions. Notice that the labels on the leaves are in lexicographic order. If we agree to label the edges from each vertex in order, then any set of functions generated sequentially by specifying $f(i)$ at the i th step will be in lex order.

To create a single function we start at the root and choose downward edges (i.e., make decisions) until we reach a leaf. This creates a *path* from the root to a leaf. We may describe a path in any of the following ways:

- the sequence of vertices v_0, v_1, \dots, v_m on the path from the root v_0 to the leaf v_m ;
- the sequence of edges e_1, e_2, \dots, e_m , where $e_i = (v_{i-1}, v_i)$, the edge connecting v_{i-1} to v_i ;
- the integer sequence of *decisions* D_1, D_2, \dots, D_m , where e_i has D_i edges to the left of it leading out from v_{i-1} .

Note that if a vertex has k edges leading out from it, the decisions are numbered $0, 1, \dots, k-1$. We will find the concept of a sequence of decisions useful in stating our algorithms in the next section. We illustrate the three approaches by looking at the leaf 2,1,2 in Figure 3.1:

- the vertex sequence is root, 2, 21, 212;
- the edge sequence is 2, 1, 2;
- the decision sequence is 1, 0, 1.

The following example uses a decision tree to list a set of patterns which are then used to solve a counting problem.

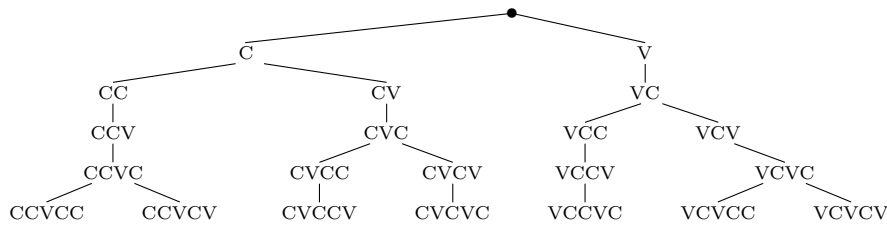


Figure 3.2 The decision tree for 5-long C-V patterns.

#V's	#CC's	ways to fill	patterns
1	2	$(20 \times 19)^2 \times 6$	CCVCC
2	0	$20^3 \times 6^2$	CVCVC
2	1	$(20 \times 19) \times 20 \times 6^2$	CCVCV CVCCV VCCVC VCVCC
3	0	$20^2 \times 6^3$	VCVCV

Figure 3.3 Grouping and counting the patterns.

Example 3.1 Counting words Using the 26 letters of the alphabet and considering the letters AEIOUY to be vowels how many five letter “words” (i.e. five long lists of letters) are there subject to the following constraints?

- (a) No vowels are ever adjacent.
- (b) There are never three consonants adjacent.
- (c) Adjacent consonants are always different.

To start with, it would be useful to have a list of all the possible patterns of consonants and vowels; e.g., CCVCV (with C for consonant and V for vowel) is possible but CVVCV and CCCVC violate conditions (a) and (b) respectively and so are not possible. We’ll use a decision tree to generate these patterns in lex order. Of course, a pattern CVCCV can be thought of as a function f where $f(1) = C, f(2) = V, \dots, f(5) = V$. In Example 3.2 we’ll use a decision tree for a counting problem in which there is not such a straightforward function interpretation.

We could simply try to list the patterns (functions) directly without using a decision tree. The decision tree approach is preferable because we are less likely to overlook something. The resulting tree is shown in Figure 3.2. At each vertex there are potentially two choices, but at some vertex only one is possible because of conditions (a) and (b) above. Thus there are one or two decisions at each vertex. You should verify that this tree lists all possibilities systematically.

* * * Stop and think about this! * * *

With enough experience, one can list the leaves of a fairly simple decision tree like this one without needing to draw the entire tree. It’s easier to make mistakes using that short cut, so we don’t recommend it—especially on exams!

We can now group the patterns according to how many vowels and how many adjacent consonants they have since these two pieces of information are enough to determine how many ways the pattern can be filled in. The results are given in Figure 3.3. To get the answer, we multiply the “ways to fill” in each row by the number of patterns in that row and sum over all four rows. The answer is $20^2 \times 6 \times 973$. \blacksquare

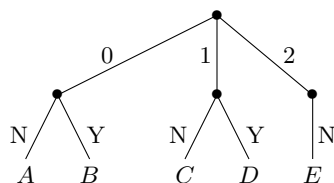


Figure 3.4 A decision tree for two 2 pair hands.

The next two examples also deal with patterns. The first example looks at patterns of overlap between two hands of cards and uses this to count hands. In contrast to the previous example, no locations are involved. The second example looks at another word problem. We take the patterns to be the number of repetitions of letters rather than the location of types of letters.

Example 3.2 Two hands of cards In Chapter 1 we studied various problems involving a hand of cards. Now we complicate matters by forming more than one hand on the same deal from the deck. How many ways can two 5 card hands be formed so that each hand contains 2 pairs (and a fifth card that has a different value)?

The problem can be solved by forming the first hand and then forming the second, since the number of choices for the second hand does not depend on what the first hand is as long as we know it is a hand with 2 pairs. We solved the two pair problem in Example 1.16 (p. 22), so we know that the first hand can be formed in 123,552 ways.

Forming the second hand is complicated by the fact that the first hand has used up some of the cards in the deck. As a result, we must consider different cases according to the amount of overlap between the first and second hands. We'll organize the possibilities by using a decision tree. Let P_i be the set of values of the pairs in the i th hand; e.g., we might have $P_1 = \{3, Q\}$ and $P_2 = \{2, 3\}$, in which case the hands have one pair value in common. Our first decision will be the value of $|P_1 \cap P_2|$, which must be 0, 1 or 2. Our next decision will be based on whether or not the value of the unpaired card in the first hand is in P_2 ; i.e., whether or not a pair in the second hand has the same value as the nonpair card in the first hand. We'll label the edges Y and N according as this is true or not. The decision tree is shown in Figure 3.4, where we've labeled the leaves A – E for convenience.

We'll prove that the number of hands that correspond to each of the leaves is

$$\begin{aligned}
 A: & \binom{10}{2} \binom{4}{2}^2 (52 - 8 - 5) = 63,180, \\
 B: & \binom{3}{2} \times \binom{10}{1} \binom{4}{2} (52 - 8 - 4) = 7,200, \\
 C: & 2 \times \binom{10}{1} \binom{4}{2} (52 - 8 - 3) = 4,920, \\
 D: & 2 \binom{3}{2} (52 - 8 - 2) = 252, \\
 E: & 52 - 8 - 1 = 43,
 \end{aligned}$$

giving a total of 75,595 choices for the second hand. Multiplying this by 123,552 (the number of ways of forming the first hand) we find that there are somewhat more than 9×10^9 possibilities for two hands. Of course, if the order of the hands is irrelevant, this must be divided by 2.

We'll derive the values for leaves A and D and let you do the rest. For A , we choose the two pairs not using any of the three values in the first hand. As in Example 1.16, this gives us $\binom{10}{2} \binom{4}{2}^2$ possibilities. We then choose the single card. To do the latter, we must avoid the values of the two pairs already chosen in the second hand and the cards in the first hand. Thus there are $52 - 8 - 5$ choices for this card. For D , we choose which element of P_1 is to be in P_2 (2 choices). The suits

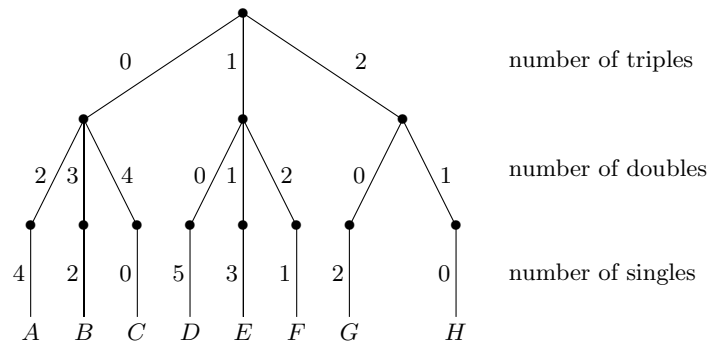


Figure 3.5 The decision tree for forming 8-letter words using ERRONEOUSNESS in Example 3.3. The first level of edges from the root indicates the number of different letters used three times; the next level, two times; and the bottom level, once.

of that pair in the second hand are determined since only two cards with that value are left in the deck after selecting the first hand. Next we choose the suits for the other pair ($\binom{3}{2}$ choices). Finally we select the fifth card. We must avoid (a) the values of the two pairs already chosen in the second hand and (b) the pair that was used in the first hand which doesn't have the same value as either of the pairs in the second hand. \square

Example 3.3 Words from a collection of letters Review the problem discussed in Examples 1.11 (p. 13) and 1.19 (p. 24), where we counted possible “words” that could be made using the letters in ERROR. In Example 1.11, we counted by listing words one at a time. In Example 1.19, we counted by grouping words according to the number of occurrences of each letter. Both approaches could be carried out using a decision tree, for example, we could first choose m_1 , then m_2 and finally m_3 in Example 1.19. Let's consider a bigger problem where it will be useful to form even larger groups of words than in Example 1.19. (Larger groups result in fewer total groups and so lead to easier counting.) We'll group words by patterns, as we did in Example 3.1, but the nature of the patterns will be different—it will be more like what we did for hands of cards in Chapter 1. Let's compute the number of 8-letter words that can be formed using the letters in ERRONEOUSNESS, using a letter no more often than it appears in ERRONEOUSNESS. We have three each of E and S; two each of O, N, and R; and one U. We will make three levels of decisions. The first level will be how many letters to use three times, the second level how many to use twice, and the third level how many to use once. Of course, since the total number of letters used must be 8 to create our 8-letter word, the choice on the third level is forced. Since there are only 6 distinct letters, we must either use at least one letter 3 times or use at least two letters twice. The decision tree is shown in Figure 3.5.

Why did we go from three times to twice to once in the levels instead of, for example in the reverse order?

* * * Stop and think about this! * * *

Had we first chosen how many letters were to appear once, the number of ways to choose letters to appear twice would depend on whether or not we had used U. The choices for triples would be even more complex. By starting with the most repeated letters and working down to the unrepeatd ones, this problem does not arise. Had we made decisions in the reverse order just described, there would have been 23 leaves. You might like to try constructing the tree.

We can easily carry out the calculations for each of eight leaves. For example, at leaf F , we choose 1 of the 2 possibilities at the first choice AND 2 of the 4 possibilities at the second choice AND 1 of the 3 possibilities at the third choice. This gives $\binom{2}{1} \binom{4}{2} \binom{3}{1}$. Listing the choices at each

time in alphabetical order, they might have been E; N; S; O. AND now we choose locations for each of these letters. This can be counted by a multinomial coefficient: $\binom{8}{3, 2, 2, 1}$. In a similar manner, we have the following results for the eight cases.

$$\begin{aligned}
 A : & \binom{2}{0} \binom{5}{2} \binom{4}{4} \binom{8}{2, 2, 1, 1, 1, 1} = 100,800 \\
 B : & \binom{2}{0} \binom{5}{3} \binom{3}{2} \binom{8}{2, 2, 2, 1, 1} = 151,200 \\
 C : & \binom{2}{0} \binom{5}{4} \binom{2}{0} \binom{8}{2, 2, 2, 2} = 12,600 \\
 D : & \binom{2}{1} \binom{4}{0} \binom{5}{5} \binom{8}{3, 1, 1, 1, 1, 1} = 13,440 \\
 E : & \binom{2}{1} \binom{4}{1} \binom{4}{3} \binom{8}{3, 2, 1, 1, 1} = 107,520 \\
 F : & \binom{2}{1} \binom{4}{2} \binom{3}{1} \binom{8}{3, 2, 2, 1} = 60,480 \\
 G : & \binom{2}{2} \binom{3}{0} \binom{4}{2} \binom{8}{3, 3, 1, 1} = 6,720 \\
 H : & \binom{2}{2} \binom{3}{1} \binom{3}{0} \binom{8}{3, 3, 2} = 1,680.
 \end{aligned}$$

Adding these up, we get 454,440. This is about as simple as we can make the problem with the tools presently at our disposal. In Example 11.6 (p. 319), we'll see how to immediately identify the answer as the coefficient of a term in the product of some polynomials. \square

In the previous examples we were interested in counting the number of objects of some sort—words, hands of cards. Decision trees can also be used to provide an orderly listing of objects. The listing is simply the leaves of the tree read from left to right. Figure 3.6 shows the decision tree for the permutations of $\underline{3}$ written in one line form. Recall that we can think of a permutation as a bijection $f : \underline{3} \rightarrow \underline{3}$ and its one line form is $f(1), f(2), f(3)$. Since we chose the values of $f(1)$, $f(2)$ and $f(3)$ in that order, the permutations are listed lexicographically; that is, in “alphabetical” order like a dictionary only with numbers instead of letters. On the right of Figure 3.6, we've abbreviated the decision tree a bit by shrinking the edges coming from vertices with only one decision and omitting labels on nonleaf vertices. As you can see, there is no “correct” way to label a decision tree. The intermediate labels are simply a tool to help you correctly list the desired objects (functions in this case) at the leaves. Sometimes one may even omit the function at the leaf and simply read it off the tree by looking at the labels on the edges or vertices associated with the decisions that lead from the root to the leaf. An example of such a tree appears in Figure 3.16 (p. 90).

Definition 3.1 Rank *The **rank** of a leaf is the number of leaves to its left. Thus, if there are n leaves the rank is an integer between 0 and $n - 1$, inclusive.*

In Figure 3.6, the leaf 1,2,3 has rank 0 and the leaf 3,1,2 has rank 4. Rank is an important tool for storing information about objects and for generating objects at random. See the next section for more discussion.

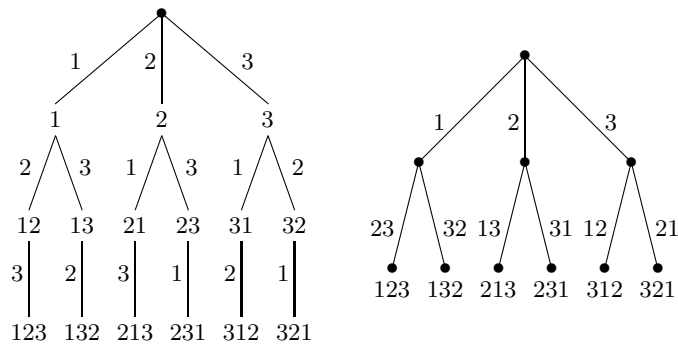


Figure 3.6 The decision tree for the permutations of $\underline{3}$ in lex order. (a) The full tree is on the left side. (b) An abbreviated tree is on the right. We’ve omitted commas and written permutations in one line form.

Example 3.4 Direct insertion order Another way to create a permutation is by *direct insertion*. (Often this is simply called “insertion.”) Suppose that we have an ordered list of k items into which we want to insert a new item. It can be placed in any of $k + 1$ places; namely, at the end of the list or immediately before the i th item where $1 \leq i \leq k$. By starting out with 1, choosing one of the two places to insert 2 in this list, choosing one of the three places to insert 3 in the new list and, finally, choosing one of the four places to insert 4 in the newest list, we will have produced a permutation of $\underline{4}$. To do this, we need to have some convention as to how the places for insertion are numbered when the list is written from left to right. The obvious choice is from left to right; however, right to left is often preferred because the leftmost leaf is then $12 \dots n$ as it is for lex order. We’ll use the obvious choice (left to right) because it is less confusing.

Here’s the derivation of the permutation associated with the insertions 1, 3 and 2.

number to insert	2	3	4	Answer
partial permutation	1	2 1	2 1 3	2 4 1 3
positions in list	1 2	1 2 3	1 2 3 4	
position to use	↑	↑	↑	

Figure 3.7 is the decision tree for generating permutations of $\underline{3}$ by direct insertion. The labels on the vertices are, of course, the partial permutations, with the full permutations appearing on the leaves. The decision labels on the edges are the positions in which to insert the next number. To the left of the tree we’ve indicated which number is to be inserted. This isn’t really necessary since the numbers are always inserted in increasing order starting with 2. Notice that the labels on the leaves are no longer in lex order because we constructed the permutations differently. Had we labeled the vertices with the positions used for insertion, the leaves would then be labeled in lex order. If you don’t see why this is so, label the vertices of the decision tree with the insertion positions.

Unlike lex order, it is not immediately clear that this method gives all permutations exactly once. We now prove this by induction on n . The case $n = 1$ is trivial since there are no insertions. Suppose $n > 1$ and that a_1, \dots, a_n is a permutation of $1, \dots, n$. If $a_k = n$, then the last insertion must have been to put n in position k and the partial permutation of $1, \dots, n - 1$ before this insertion was $a_1, \dots, a_{k-1}, a_{k+1}, \dots, a_n$. By the induction assumption, the insertions leading to this permutation are unique and so we are done.

Like the method of lex order generation, the method of direct insertion generation can be used for other things besides permutations. However, direct insertion cannot be applied as widely as lex order. Lex order generation works with anything that can be thought of as an (ordered) list, but direct insertion requires more structure. \square

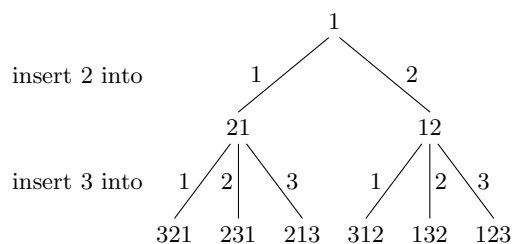


Figure 3.7 The decision tree for the permutations of $\underline{3}$ in direct insertion order. We've omitted commas.

Example 3.5 Transposition order Yet another way to create a permutation is by *transposition*. It is similar to direct insertion; however, instead of pushing a number into a space between elements of the partial permutation, we put it into a place that may already be occupied, bumping that element out of the way. If an element is bumped, it is moved to the end of the list.

Here's the derivation of the permutation associated with 1, 3 and 2. Note that, while direct insertion used positions between elements of the partial permutation, transposition uses the positions of the partial permutation plus one more space at the end.

number to insert	2	3	4	Answer
partial permutation	1	2 1	2 1 3	2 4 3 1
positions in list	1 2	1 2 3	1 2 3 4	
position to use	↑	↑	↑	

As with direct insertion, unique generation of all permutations is not obvious. A similar inductive proof works: Suppose the permutation is a_1, \dots, a_n and $a_k = n$. The preceding partial permutation is

- a_1, \dots, a_{n-1} , if $k = n$,
- $a_1, \dots, a_{k-1}, a_n, a_{k+1}, \dots, a_{n-1}$, if $k < n$.

By the induction assumption, we can generate any $(n-1)$ -long permutation, so generate whichever of the above permutations of $\{1, \dots, n-1\}$ is required and then put n into position k , bumping if necessary. Thus the map from the $n!$ possible transposition sequences to permutations is a surjection. Since the domain and image of the map are the same size, the map is a bijection. This completes the proof.

Why is this called the transposition method? Suppose the sequence of positions is p_2, \dots, p_n . Then the permutation is obtained by starting with the identity permutation and applying, in order, the transpositions $(2, p_2), (3, p_3), \dots, (n, p_n)$, where the pseudo-transposition (k, k) is interpreted as doing nothing.

Since this seems more complicated than lex order and insertion order, why would anyone use it? It is an easy way to generate random permutations: Generate a sequence p_2, \dots, p_n of random integers where $1 \leq p_k \leq k$. Then apply the method of the previous paragraph to create the random permutation. \square

Example 3.6 Programs to list the leaves A decision tree consists of a set of sequential decisions, and so the code fragment discussed in connection with the Rule of Product is relevant. Here it is in decision tree language:

```

For each first decision  $d_1$ 
...
    For each  $k$ th decision  $d_k$ 
        List the structure arising from the decisions  $d_1, \dots, d_k$ .
    End for
...
End for

```

We now give the code to generate the leaves of Figure 3.2. In `Create(a,b)`, we find the next possible decisions given the two preceding ones, `a` followed by `b`. Note that, if d_1 is the first decision, `Create(V, d_1)` generates the correct set of possible second decisions.

```

Function Create(a,b)                                /* Create next set of decisions. */
    If  $b = V$ 
        Return {C}                                  /* Avoid VV pattern. */
    Elseif  $a = C$ 
        Return {V}                                  /* Avoid CCC pattern. */
    Else
        Return {C, V}
    End if
End

```

```

Procedure ListCV
     $D_1 = \{C, V\}$ 
    For  $d_1 \in D_1$ :
         $D_2 = \text{Create}(V, d_1)$                     /* V is fake zeroth decision. */
        For  $d_2 \in D_2$ :
             $D_3 = \text{Create}(d_1, d_2)$ 
            For  $d_3 \in D_3$ :
                 $D_4 = \text{Create}(d_2, d_3)$ 
                For  $d_4 \in D_4$ :
                     $D_5 = \text{Create}(d_3, d_4)$ 
                    For  $d_5 \in D_5$ : List  $d_1 d_2 d_3 d_4 d_5$  End for
                End for
            End for
        End for
    End for
End for
End for
End

```

Exercises

The following exercises are intended to give you some hands-on experience with simple decision trees before we begin our more systematic study of them in the next section.

- 3.1.1. What permutations of $\underline{3}$ have the same rank in lexicographic order and insertion order?

- 3.1.2. Draw the decision tree for the permutations of $\underline{4}$ in lexicographic order.
- What is the rank of 2,3,1,4? of 4,1,3,2?
 - What permutation in this tree has rank 5? rank 15?
- 3.1.3. Draw the decision tree for the permutations of $\underline{4}$ in direct insertion order.
- What is the rank of 2314? of 4132?
 - What permutation in this tree has rank 5? rank 15?
- 3.1.4. Draw the decision tree for the permutations of $\underline{4}$ in transposition order.
- What is the rank of 2314? of 4132?
 - What permutation in this tree has rank 5? rank 15?
- 3.1.5. Draw a decision tree to list all 6-long sequences of A's and B's that satisfy the following conditions.
- There are no adjacent A's.
 - There are never three B's adjacent.
- 3.1.6. Draw the decision tree for the strictly decreasing functions in $\underline{6^4}$ in lex order.
- What is the rank of 5,4,3,1? of 6,5,3,1?
 - What function has rank 0? rank 7?
 - What is the largest rank and which function has it?
 - Your tree should contain the decision tree for the strictly decreasing functions in $\underline{5^4}$. Indicate it and use it to list those functions in order.
 - Indicate how all the parts of this exercise can be interpreted in terms of subsets of a set.
- 3.1.7. Draw the decision tree for the nonincreasing functions in $\underline{4^3}$.
- What is the rank of 321? of 443?
 - What function has rank 0? rank 15?
 - What is the largest rank and which function has it?
 - Your tree should contain the decision tree for the nonincreasing functions in $\underline{3^3}$. Circle that tree and use its leaves to list the nonincreasing functions in $\underline{3^3}$.
 - Can you find the decision tree for the strictly decreasing functions in $\underline{4^3}$ in your tree?
- 3.1.8. Describe the lex order decision tree for producing all the names in Example 2 in the Introduction to Part I.
- *3.1.9. In a list of the permutations on \underline{n} with $n > 1$, the permutation just past the middle of the list has rank $n!/2$.
- What is the permutation of \underline{n} that has rank 0 in insertion order? rank $n! - 1$? rank $n!/2$?
 - What is the permutation of \underline{n} that has rank 0 in transposition order? rank $n! - 1$? rank $n!/2$?
 - What is the permutation of \underline{n} that has rank 0 in lex order? rank $n! - 1$? rank $n!/2$ when n is even? rank $n!/2$ when n is odd?
- Hint.* For $n!/2$, look at some pictures of the trees.
- 3.1.10. How many ways can two full houses be formed?
- 3.1.11. How many ways can two 5-card hands be formed so that the first is a full house and the second contains two pair?
- Do this in the obvious manner by first choosing the full house and then choosing the second hand.
 - Do this in the less obvious manner by first choosing the second hand and then choosing the full house.
 - What lesson can be learned from the previous parts of this exercise?

3.1.12. How many ways can three full houses be formed?

3.1.13. How many 7-letter “words” can be formed from ERRONEOUSNESS where no letter is used more times than it appears in ERRONEOUSNESS?

*3.2 Ranking and Unranking

Suppose we have a decision tree whose leaves correspond to a certain class of objects; e.g., the permutations of $\underline{3}$. An important property of such a decision tree is that it leads to a bijection between the n leaves of the tree and the set $\{0, 1, \dots, n - 1\}$. In the previous section we called this bijection the rank of the leaf.

Definition 3.2 RANK and UNRANK *Suppose we are given a decision tree with n leaves. If v is a leaf, the function $\text{RANK}(v)$ is the number of leaves to the left of it in the decision tree. Since RANK is a bijection from leaves to $\{0, 1, \dots, n - 1\}$, it has an inverse which we call UNRANK.*

How can these functions be used?

- **Simpler storage:** They can simplify the structure of an array required for storing data about the objects at the leaves. The RANK function is the index into the storage array. The UNRANK function tells which object has data stored at a particular array location. Thus, regardless of the nature of the N objects at the leaves, we need only have an N -long singly indexed array to store data.
- **Less storage:** Suppose we are looking at the 720 permutations of $\underline{6}$. Using RANK and UNRANK, we need a 720-long array to store data. If we took a naive approach, we might use a six dimensional array (one dimension for each of $f(1), \dots, f(6)$), and each dimension would be able to assume six values. Not only is this more complicated than a singly dimensioned array, it requires $6^6 = 46,656$ storage locations instead of 720.
- **Random generation:** Suppose that you want to study properties of typical objects in a set of N objects, but that N is much too large to look at all of them. If you have a random number generator and the function UNRANK, then you can look at a random selection of objects: As often as desired, generate a random integer, say r , between 0 and $N - 1$ inclusive and study the object $\text{UNRANK}(r)$. This is sometimes done to collect statistics on the behavior of an algorithm.

You’ll now learn a method for computing RANK and UNRANK functions for some types of decision trees. The basic idea behind all these methods is that, when you make a decision, all the decisions to the left of the one just made contribute the leaves of their “subtrees” to the rank of the leaf we are moving toward.

To talk about this conveniently, we need some terminology. Let $e = (v, w)$ be an edge of a tree associated with a decision D at vertex v ; that is, e connects v to the vertex w that is reached by making decision D at v . The *residual tree of (v, w)* , written $R(v, w)$ or $R(e)$, is v together with all edges and vertices that can be reached from v by starting with one of the D decisions to the left of (v, w) . For example, the residual tree of the edge labeled 2 in Figure 3.6(b) consists of the edges labeled 1, 23 and 32 and the four vertices attached to these edges. The edge labeled 2 has $D = 1$. When $D = 0$, there are no edges to the left of e that have v as a parent. Thus, the *residual tree of e* consists of just one vertex when $D = 0$. Let $\Delta(e)$ be the number of leaves in $R(e)$, not counting the root. Thus $\Delta(e) = 0$ when e corresponds to $D = 0$ and $\Delta(e) > 0$ otherwise. The following result forms the basis of our calculations of RANK and UNRANK.

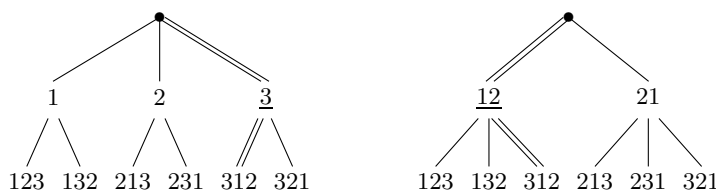


Figure 3.8 Decision trees for permutations of $\{1, 2, 3\}$. The decisions taken to reach 312 are shown in double lines. The lex order is on the left and direction insertion on the right. We've omitted commas.

Theorem 3.1 Rank formula *If the sequence of edges on the path from the root to a leaf X in a decision tree is e_1, e_2, \dots, e_m , then*

$$\text{RANK}(X) = \sum_{i=1}^m \Delta(e_i).$$

Before proving this let's look at some simple situations to see what's happening. In Figure 3.8 we have the lex and insertion order decision trees for permutations of $\underline{3}$ with the decisions needed to reach 312 drawn double. For lex order we have $\Delta(e) = 4$ for $e = (\bullet, 3)$ and $\Delta(e) = 0$ for $e = (3, 312)$. (We write this simply as $\Delta(\bullet, 3) = 4$ and $\Delta(3, 312) = 0$.) For the insertion order we have $\Delta(1, 12) = 0$ and $\Delta(12, 312) = 2$. This gives ranks of 4 and 2 in the two trees.

Proof: Now let's prove the theorem. We'll use induction on m . Let $e_1 = (v, w)$. The leaves to the left of X in the tree rooted at v are the leaves in $R(v, w)$ together with the leaves to the left of X in the tree starting at w . In other words, $\text{RANK}(X)$ is $\Delta(v, w)$ plus the rank of X in the tree starting at w .

Suppose that $m = 1$. Since $m = 1$, w must be a leaf and hence $w = X$. Thus the tree starting at w consists simply of X and so contains no leaves to the left of X . Hence X has rank 0 in that tree. This completes the proof for $m = 1$.

Suppose the theorem is true for decision sequences of length $m - 1$. Look at the subtree that is rooted at w . The sequence of edges from the root w to X is e_2, \dots, e_m . Since $R(e_i)$ (for $i > 1$) is the same for the tree starting at w as it is for the full tree, it follows by the induction assumption that X has rank $\sum_{i=2}^m \Delta(e_i)$ in this tree. This completes the proof. \square

Calculating RANK

Calculating $\text{RANK}(X)$ is, in principle, now straightforward since we need only obtain formulas for the $\Delta(e_i)$'s. Unfortunately, each decision tree is different and it is not always so easy to obtain such formulas. We'll work some examples to show you how it is done in some simple cases.

Example 3.7 Lex order rank of all functions If D_1, \dots, D_k is a sequence of decisions in the lex order tree for \underline{n}^k leading to a function $f \in \underline{n}^k$, then $D_i = f(i) - 1$ since there are $f(i) - 1$ elements of \underline{n} that are less than $f(i)$. What is the value of $\text{RANK}(f)$?

By Theorem 3.1, we need to look at $R(e_i)$, where e_i is an edge on the path. The structure of $R(e_i)$ is quite simple and symmetrical. There are D_i edges leading out from the root. Whichever edge we take, it leads to a vertex that has n edges leading out. This leads to another vertex with n edges leading out and so on until we reach a leaf. In other words, we make one D_i -way decision and $(k - i)$ n -way decisions. Each leaf of $R(e_i)$ is reached exactly once in this process and thus, by the Rule of Product $\Delta(e_i) = D_i n^{k-i}$. We have proved

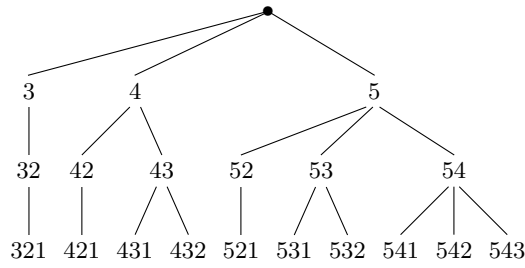


Figure 3.9 The lex order decision tree for the strictly decreasing functions in $\underline{5}^3$. We've omitted commas.

Theorem 3.2 Lex order rank of all functions For f in the lex order listing of all functions in \underline{n}^k ,

$$\text{RANK}(f) = \sum_{i=1}^k (f(i) - 1)n^{k-i}.$$

Notice that when $n = 10$ this is just a way of writing the number whose arabic notation is $D_1D_2 \dots D_k$. Indeed, the functions in $\underline{10}^k$ written in one line form look just like k digit numbers except that each digit has been increased by one. Lex order is then the same as numerical order.

This realization, generalized from base 10 to base n , gives another approach to the formula for $\text{RANK}(f)$. It also makes it easy to see how to find the function that immediately follows f : Simply write out the set of decisions that lead to f as a number in base n and add 1 to this number to obtain the decisions for the function immediately following f . To get the function that is k further on, add k . To get the function that is k earlier, subtract k . (If we have to carry or borrow past the leftmost digit D_1 , then we are attempting to go beyond the end of the tree.) For instance, the successor of $1,3,2,3,3$ in $\underline{3}^5$ is obtained as follows. The decisions that lead to $1,3,2,3,3$ are $0,2,1,2,2$. Since there are three possible decisions at each point, we can think of the decisions as the base 3 number 02122 and add 1 to it to obtain 02200 , which we think of as decisions. Translating from decisions to function values, we obtain $1,3,3,1,1$ as the successor of $1,3,2,3,3$. \square

Example 3.8 Strictly decreasing functions We will now study the strictly decreasing functions in \underline{n}^k , which we have observed correspond to k element subsets of \underline{n} .

The decision tree for lex order with $n = 5$ and $k = 3$ is in Figure 3.9. You can use it to help you visualize some of the statements we'll be making.

Calculating the rank of a leaf differs from the preceding examples because the tree lacks symmetry; however, it has another feature which enables us to calculate the rank quite easily. Let's look for the rank of the strictly decreasing function $f \in \underline{n}^k$.

Let e_1 be the first edge on the path to the leaf for f . $R(e_1)$ is the tree for all strictly decreasing functions $g \in \underline{n}^k$ with $g(1) < f(1)$. In other words, $R(u, v)$ is the tree for the strictly decreasing functions in $\underline{f(1) - 1}^k$.

We can generalize this observation. Suppose the path is e_1, e_2, \dots, e_k . Suppose that g is a leaf of $R(e_i)$. This can happen if and only if $g \in \underline{n}^k$ is a strictly decreasing function with

$$g(1) = f(1), g(2) = f(2), \dots, g(i - 1) = f(i - 1), \text{ and } g(i) < f(i).$$

Since $g(j)$ is determined for $j < i$, look just at $g(i), g(i + 1), \dots, g(k)$. It is an arbitrary strictly decreasing function with initial value less than $f(i)$. Thus the leaves of $R(e_i)$ can be associated with the strictly decreasing functions in $\underline{f(i) - 1}^{k+1-i}$. Since there are $\binom{b}{a}$ strictly decreasing functions in \underline{b}^a , $\Delta(e_i) = \binom{f(i)-1}{k+1-i}$. Thus

Theorem 3.3 Lex order rank of strictly decreasing functions For f in the lex order listing of all strictly decreasing functions in \underline{n}^k ,

$$\text{RANK}(f) = \sum_{i=1}^k \binom{f(i)-1}{k+1-i}. \quad 3.1$$

Let's use the theorem to calculate the rank of the strictly decreasing function $f = 96521$. What are k and n ? Since there are five numbers in the list, we can see that $k = 5$; however, there's no way to determine n . This means that f has not been fully specified since we don't know its range. On the other hand, our formula doesn't require n and, as we remarked when defining functions, a specification of a function which omits the range may often be considered complete anyway. By our formula

$$\text{RANK}(96521) = \binom{8}{5} + \binom{5}{4} + \binom{4}{3} + \binom{1}{2} + \binom{0}{1} = 56 + 5 + 4 + 0 + 0 = 65.$$

What is the decision sequence for a strictly decreasing function? For the strictly decreasing functions in \underline{n}^k , the decision associated with choosing $f(i) = j$ is $i + j - (k + 1)$. Why is this so? We must choose distinct values for $f(i + 1), \dots, f(k)$. Thus there must be at least $k - i$ values less than j . Hence smallest possible value for $f(i)$ is $k + 1 - i$. Thus there are $j - (k + 1 - i)$ possible values less than j .

Note that (3.1) does not depend on n and so, for each k , there is one decision tree that works for all n . The root has an infinite number of children—one for each value of $f(1)$. Thereafter the number of decisions is finite since $f(i) < f(1)$ for $i > 1$. This “universal” decision tree arises because, once $f(1)$ is given, the value of n imposes no constraints on $f(i)$ for $i > 1$. Had we used strictly *increasing* functions instead, this would not have been the case because $n \geq f(i) > (1)$ for all $i > 1$. \square

***Example 3.9 Direct insertion order rank of all permutations** Now let's compute ranks for the permutations of \underline{n} in direct insertion order, a concept defined in Example 3.4. We can use practically the same approach that was used in Example 3.7. Number the positions where we can insert j as $0, 1, 2, \dots, j$. (We started with 0, not 1 for the first position.) If D_2, D_3, \dots, D_n is the sequence of decisions, then D_j is the position into which j is to be inserted. Note that we've started the sequence subscripting with 2 so that the subscript j equals the number whose insertion is determined by D_j .

Since we usually tend to write permutations in one line form, we should develop a method for determining D_j from the one line form. Here is a simple rule:

Write the permutation in one line form. Count the elements of the permutation *from right to left* up to but not including j and ignoring all elements that exceed j . That count is the position in which j was inserted and so equals D_j .

As an illustration, consider the permutation with the one line form 4,6,1,2,5,7,3. Starting from the right, counting until we reach 2 and ignoring numbers exceeding 2, we get that $D_2 = 0$. Similarly, $D_3 = 0$. Since we encounter 3, 2 and 1 before reaching 4, $D_4 = 3$. Only 3 intervenes when we search for 5, so $D_5 = 1$. Here's the full list:

$$D_2 = 0, D_3 = 0, D_4 = 3, D_5 = 1, D_6 = 4, D_7 = 1.$$

Again $R(e_i)$ has a nice symmetrical form as we choose a route to a leaf:

- Choose one of D_i edges AND
- Choose one of $i + 1$ edges (position for $i + 1$) AND
-
- Choose one of n edges (position for n).

By the Rule of Product, $\Delta(e_i) = D_i(i+1)(i+2)\cdots n = D_i n! / i!$ and so we have

Theorem 3.4 Direct insertion order rank *For f in the direct insertion order listing of all permutations on \underline{n} ,*

$$\text{RANK}(f) = \sum_{i=2}^n \frac{D_i n!}{i!}.$$

This formula together with our previous calculations gives us

$$\begin{aligned} \text{RANK}(4, 6, 1, 2, 5, 7, 3) &= 0 \times 7!/2! + 0 \times 7!/3! + 3 \times 7!/4! \\ &\quad + 1 \times 7!/5! + 4 \times 7!/6! + 1 \times 7!/7! = 701. \end{aligned}$$

What permutation immediately follows $f = 9, 8, 7, 3, 1, 6, 5, 2, 4$ in direct insertion order? The obvious way to do this is to calculate $\text{UNRANK}(1 + \text{RANK}(9, 8, 7, 3, 1, 6, 5, 2, 4))$, which is a lot of work. By extending an idea in the previous example, we can save a lot of effort. Here is a general rule:

The next leaf in a decision tree is the one with the lexically next (legal) decision sequence.

Let's apply it. The decisions needed to produce f are 0,2,0,2,3,6,7,8. If we simply add 1 to the last decision, we obtain an illegal sequence because there are only nine possible decisions there. Just like in arithmetic, we reset it to 0 and carry. Repeating the process, we finally obtain 0,2,0,2,4,0,0,0. This corresponds to the sequence 1,3,1,3,5,1,1,1 of insertions. You should be able to carry out these insertions: start with the sequence 1, insert 2 in position 1, then insert 3 in position 3, and so on, finally inserting 9 in position 1. The resulting permutation is 3,6,1,5,2,4,7,8,9.

For special situations such as the ones we've been considering, one can find other short cuts that make some of the steps unnecessary. In fact, one need not even calculate the decision sequence. We will not discuss these short cuts since they aren't applicable in most situations. Nevertheless, you may enjoy the challenge of trying to find such shortcuts for permutations in lex and direct insertion order and for strictly decreasing functions in lex order. \square

Calculating UNRANK

The basic principle for unranking is greed.

Definition 3.3 Greedy algorithm *A greedy algorithm is a multistep algorithm that obtains as much as possible at the present step with no concern for the future.*

The method of long division is an example of a greedy algorithm: If d is the divisor, then at each step you subtract off the largest possible number of the form $(k \times 10^n)d$ from the dividend that leaves a nonnegative number and has $1 \leq k \leq 9$.

The greedy algorithm for computing UNRANK is to choose D_1 , then D_2 and so on, each D_i as large as possible at the time it is chosen. What do we mean by “as large as possible?” Suppose we are calculating UNRANK(m). If D_1, \dots, D_{i-1} have been chosen, then make D_i as big as possible subject to the condition that $\sum_{j=1}^i \Delta(e_j) \leq m$.

Why does this work? Suppose that D_1, \dots, D_{i-1} have been chosen by the greedy algorithm and are part of the correct path. (This is certainly true when $i = 1$ because the sequence is empty!) We will prove that D_i chosen by the greedy algorithm is also part of the correct path.

Suppose that a path starts $D_1, \dots, D_{i-1}, D'_i$. If $D'_i > D_i$, this cannot be part of the correct path because the definition of D_i gives $\Delta(e_1) + \dots + \Delta(e_{i-1}) + \Delta(e'_i) > m$.

Now suppose that $D'_i < D_i$. Let x be the leftmost leaf reachable from the decision sequences that start D_1, \dots, D_i . Clearly $\text{RANK}(x) = \Delta(e_1) + \dots + \Delta(e_i) \leq m$. Thus any leaf to the left of x will have rank less than m . Since all leaves reachable from D'_i are to the left of x , D'_i is not part of the correct decision sequence.

We have proven that if $D'_i \neq D_i$, then $D_1, \dots, D_{i-1}, D'_i$ is not part of the correct path. It follows that D_1, \dots, D_{i-1}, D_i must be part of the correct path.

As we shall see, it's a straightforward matter to apply the greedy algorithm to unranking if we have the values of Δ available for various edges in the decision tree.

Example 3.10 Strictly decreasing functions What strictly decreasing function f in $\underline{9}^4$ has rank 100?

In view of Theorem 3.3 (p. 78), it is more natural to work with the function values than the decision sequence. Since $\Delta(e_i) = \binom{f(i)-1}{k+1-i}$, it will be handy to have a table of binomial coefficients to look at while calculating. Thus we've provided one in Figure 3.10. Since $k = 4$, the value of $f(1) - 1$ is the largest x such that $\binom{x}{4}$ does not exceed 100. We find $x = 8$ and $\binom{8}{4} = 70$. We now need to find x so that $\binom{x}{3} \leq 100 - 70 = 30$. This gives 6 with $30 - 20 = 10$ leaves unaccounted for. With $\binom{5}{2} = 10$ all leaves are accounted for. Thus we get $\binom{0}{1}$ for the last Δ . Our sequence of values for $f(i) - 1$ is thus 8, 6, 5, 0 and so $f = 9, 7, 6, 1$.

Although we specified that the domain of f was $\underline{9}$, the value 9 was never used in our calculations. This is like the situation we encountered when computing the rank. Thus, for ranking and unranking decreasing functions in \underline{n}^k , there is no need to specify n .

Now let's find the strictly decreasing function of rank 65 when $k = 5$. The rank formula is

$$65 = \binom{f(1)-1}{5} + \binom{f(2)-1}{4} + \binom{f(3)-1}{3} + \binom{f(4)-1}{2} + \binom{f(5)-1}{1}. \quad 3.2$$

Since $\binom{8}{5} = 56 < 65 < \binom{9}{5} = 126$, it follows that $f(1) = 9$. This leaves $65 - 56 = 9$ to account for. Since $\binom{5}{4} = 5 < 9 < \binom{6}{4} = 15$, we have $f(2) = 6$ and $9 - 5 = 4$ to account for. Since $\binom{4}{3} = 4$, $f(3) = 5$ and there is no more rank to account for. If we choose $f(4) \leq 2$ and $f(5) \leq 1$, the last two binomial coefficients in (3.2) will vanish. How do we know what values to choose? The key is to remember that f is *strictly* decreasing and takes on positive integer values. These conditions force $f(5) = 1$ and $f(4) = 2$. We got rid of the apparent multiple choice for f in this case, but will that always be possible?

* * * Stop and think about this! * * *

Yes, in any unranking situation there is at most one answer because each thing has a unique rank. Furthermore, if the desired rank is less than the total number of things, there will be some thing with that rank. Hence there is exactly one answer. \square

	0	1	2	3		0	1	2	3	4	5	6	7		
0	1				8	1	8	28	56	70					
1	1				9	1	9	36	84	126					
2	1	2			10	1	10	45	120	210	252				
3	1	3			11	1	11	55	165	330	462				
4	1	4	6			12	1	12	66	220	495	792	924		
5	1	5	10			13	1	13	78	286	715	1287	1716		
6	1	6	15	20			14	1	14	91	304	1001	2002	3003	3432
7	1	7	21	35			15	1	15	105	455	1365	3003	5005	6435

Figure 3.10 Some binomial coefficients. The entry in row n and column k is $\binom{n}{k}$. For $n > k/2$ use $\binom{n}{k} = \binom{n}{n-k}$.

***Example 3.11 Direct insertion order** What permutation f of $\underline{7}$ has rank 3,000 in insertion order?

Let the decision sequence be D_2, \dots, D_7 . The number of leaves in the residual tree for D_i is $D_i \times 7!/i!$ by our derivation of Theorem 3.4 (p. 79). Since $7!/2! = 2,520$, the greedy value for D_2 is 1. The number of leaves unaccounted for is $3,000 - 2,520 = 480$. Since $7!/3! = 840$, the greedy value for D_3 is 0 and 480 leaves are still unaccounted for. Since $7!/4! = 210$, we get $D_4 = 2$ and 60 remaining leaves. Using $7!/5! = 42$, $7!/6! = 7$ and $7!/7! = 1$, we get $D_5 = 1$, $D_6 = 2$ and $D_7 = 4$. Thus the sequence of decisions is 1,0,2,1,2,4, which is the same as insertion positions. You should be able to see that $f = 2, 4, 7, 1, 6, 5, 3$. \square

***Gray Codes**

Suppose we want to write a program that will have a loop that runs through all permutations of n . One way to do this is to run through numbers $0, \dots, n! - 1$ and apply UNRANK to each of them. This may not be the best way to construct such a loop. One reason is that computing UNRANK may be time consuming. Another, sometimes more important reason is that it may be much easier to deal with a permutation that does not differ very much from the previous one. For example, if we had n large blocks of data of various lengths that had to be in the order given by the permutation, it would be nice if we could produce the next permutation simply by swapping two adjacent blocks of data.

Methods that list the elements of a set so that adjacent elements in the list are, in some natural sense, close together are called *Gray codes*.

Suppose we are given a set of objects and a notion of closeness. How does finding a Gray code compare with finding a ranking and unranking algorithm? The manner in which the objects are defined often suggests a natural way of listing the objects, which leads to an efficient ranking algorithm (and hence a greedy unranking algorithm). In contrast, the notion of closeness seldom suggests a Gray code. Thus finding a Gray code is usually harder than finding a ranking algorithm. If we are able to find a Gray code, an even harder problem appears: Find, if possible, an efficient ranking algorithm for listing the objects in the order given by the Gray code.

All we'll do is discuss one of the simplest Gray codes.

Example 3.12 A Gray code for subsets of a set We want to look at all subsets of \underline{n} . It will be more convenient to work with the representation of subsets by n -strings of zeroes and ones: The string $s_1 \dots s_n$ corresponds to the subset S where $i \in S$ if and only if $s_i = 1$. Thus the all zeroes string corresponds to the empty set and the all ones string to \underline{n} .

Providing ranking and unranking algorithms is simple; just think of the strings as n -digit binary numbers. While adjacent strings in this ranking are close together numerically, their patterns of zeroes and ones may differ greatly. As we shall see, this tells us that the ranking doesn't give a Gray code.

Before we can begin to look for a Gray code, we must say what it means for two subsets (or, equivalently, two strings) to be close. Two strings will be considered close if they differ in exactly one position. In set terms, this means one of the sets can be obtained from the other by removing or adding a single element. With this notion of closeness, a Gray code for all subsets when $n = 1$ is 0, 1. A Gray code for all subsets when $n = 2$ is 00, 01, 11, 10.

How can we produce a Gray code for all subsets for arbitrary n ? There is a simple recursive procedure. The following construction of the Gray code for $n = 3$ illustrates it.

0 00	1 10
0 01	1 11
0 11	1 01
0 10	1 00

You should read down the first column and then down the second. Notice that the sequences in the first column begin with 0 and those in the second with 1. The rest of the first column is simply the Gray code for $n = 2$ while the second column is the Gray code for $n = 2$, read from the last sequence to the first.

We now prove that this “two column” procedure for building a Gray code for subsets of an n -set from the Gray code for subsets of an $(n - 1)$ -set always works. Our proof will be by induction. For $n = 1$, we have already exhibited a Gray code. Suppose that $n > 1$ and that we have a Gray code for $n - 1$. (This is the induction assumption.) We form the first column by listing the Gray code for $n - 1$ and attaching a 0 at the front of each $(n - 1)$ -string. We form the second column by listing the Gray code, starting with the last and ending with the first, and attaching a 1 at the front of each $(n - 1)$ -string. Within a column, there is never any change in the first position and there is only a single change from line to line in the remaining positions because they are a Gray code by the induction assumption. Between the bottom of the first column and the top of the second, the only change is in the first position since the remaining $n - 1$ positions are the last element of our Gray code for $n - 1$. This completes the proof.

As an extra benefit, we note that the last element of our Gray code differs in only one position from the first element (Why?), so we can cycle around from the last element to the first by a single change.

It is a simple matter to draw a decision tree for this Gray code. In fact, Figure 3.1 is practically the $n = 3$ case—all we need to do is change some labels and keep track of whether we have reversed the code for the second column. (Reversing the code corresponds to interchanging the 0 and 1 edges.) The decision tree is shown in Figure 3.11. There is an easy way to decide whether the two edges leading down from a vertex v should be labeled 0-1 or 1-0: If the edge e leading into v is a 0 decision (i.e., 0 edge), use the same pattern that was used for e and the other decision e' it was paired with; otherwise, reverse the order. Another way you can think of this is that as you trace a path from the root to a leaf, going left and right, a 0 causes you to continue in the same direction and a 1 causes you to switch directions. \square

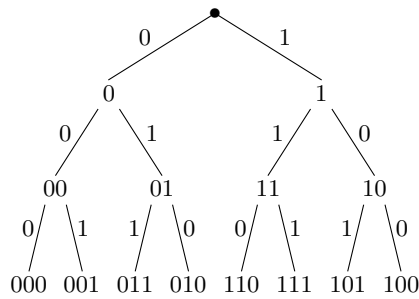


Figure 3.11 The decision tree for the Gray code for $\underline{3}$.

Exercises

Unless otherwise noted, the domain is \underline{k} for some k and the range is \underline{n} for some n . Usually, there is no need to specify n and specifying the function determines k .

- 3.2.1. This problem concerns strictly decreasing functions listed in lex order.
- Compute the rank of the functions 11,6,4 and 9,6,3,1. (Note that the domain of the first is $\underline{3}$ and the domain of the second is $\underline{4}$.)
 - What strictly decreasing function with domain $\underline{4}$ has rank 35? rank 400?
 - What strictly decreasing function immediately follows 9,6,3,2,1? 9,6,5,4?
 - What strictly decreasing function immediately precedes 9,6,3,2,1? 9,6,5,4?
- 3.2.2. This problem concerns permutations listed in direct insertion order.
- Compute the ranks of 6,3,5,1,2,4 and 4,5,6,1,2,3.
 - Determine the permutations of $\underline{6}$ with ranks 151 and 300.
 - What permutation immediately follows 9,8,7,1,2,3,4,5,6? 6,5,4,1,2,3,7,8,9?
 - What permutation immediately precedes 9,8,7,1,2,3,4,5,6? 6,5,4,1,2,3,7,8,9?
- 3.2.3. Consider the three strictly decreasing functions from \underline{k} to \underline{n} :
- $k, k-1, \dots, 2, 1$
 - $k+1, k, \dots, 3, 2$
 - $k+2, k+1, \dots, 4, 3$
- Obtain simple formulas for their ranks.
- 3.2.4. This problem concerns nonincreasing functions listed in lex order.
- Prove that

$$\text{RANK}(f) = \sum_{i=1}^k \binom{f(i) + k - i - 1}{k + 1 - i}.$$

Hint. Example 2.11 (p. 52) may be useful.

- Compute the ranks of 5,5,4,2,1,1 and 6,3,3.
 - What nonincreasing function on $\underline{4}$ has rank 35? 400?
- 3.2.5. This problem concerns the permutations listed in lex order.
- Obtain a formula for $\text{RANK}(f)$ in terms of the decisions D_1, \dots, D_n (or D_1, \dots, D_{n-1} if $f(n-1)$ and $f(n)$ are considered as being specified at the same time).
 - Describe a method for going from a permutation in one line form to the decision sequence.
 - Compute the ranks of 5,6,1,3,4,2 and 6,2,4,1,3,5.
 - Compute the sequence of decisions for permutations of $\underline{6}$ which have ranks 151 and 300. What are the one line forms of the permutations?

3.2.6. Write computer code for procedures to calculate RANK and UNRANK. You might store the function being ranked as an array of integers where the first component is the size of the domain. The classes of functions for which you should write procedures are:

- (a) the strictly decreasing functions in \underline{n}^k in lex order;
- (b) the nonincreasing functions in \underline{n}^k in lex order.
- (c) the permutations of \underline{n} in insertion order;
- (d) the permutations of \underline{n} in lex order;

Hint. For the first two, it might be useful to have a procedure for calculating the binomial coefficients $C(a, b)$.

3.2.7. Returning to Example 3.12, draw the decision tree for $n = 4$ and list the Gray code for $n = 5$.

3.2.8. Returning to Example 3.12, compute the ranks of the following sequences in the Gray code for subsets of \underline{n} :

0110; 1001; 10101; 1000010; 01010101.

(The value of n is just the length of the sequence.)

3.2.9. Returning to Example 3.12 for each of the following values of n and k , find the n -string of rank k in the Gray code for subsets of \underline{n} :

$n = 20, k = 0$; $n = 20, k = 2^{19}$; $n = 4, k = 7$; $n = 8, k = 200$.

*3.2.10. We will write the permutations of \underline{n} in one line form. Two permutations will be considered adjacent if one can be obtained from the other by interchanging the elements in two adjacent positions. We want a Gray code for all permutations. Here is such a code for $n = 4$. You should read down the first column, then the second and finally the third.

1,2,3,4	3,1,2,4	2,3,1,4
1,2,4,3	3,1,4,2	2,3,4,1
1,4,2,3	3,4,1,2	2,4,3,1
4,1,2,3	4,3,1,2	4,2,3,1
4,1,3,2	4,3,2,1	4,2,1,3
1,4,3,2	3,4,2,1	2,4,1,3
1,3,4,2	3,2,4,1	2,1,4,3
1,3,2,4	3,2,1,4	2,1,3,4

List a Gray code for $n = 5$. As a challenge, describe a method for listing a Gray code for general n .

*3.3 Backtracking

In many computer algorithms it is necessary to systematically inspect all the vertices of a decision tree. A procedure that systematically inspects all the vertices is called a *traversal of the tree*.

How can we create such a procedure? One way to imagine doing this is to walk around the tree. An example is shown in Figure 9.2 (p. 249), where we study the subject in more depth. “Walking around the tree” is not a very good program description. We can describe our traversal more precisely by giving an algorithm. Here is one which traverses a tree whose leaves are associated with functions and lists the functions in the order of their rank.

Theorem 3.5 Systematic traversal algorithm *The following procedure systematically visits the leaves in a tree from left to right by “walking” around the tree.*

1. **Start:** *Mark all edges as unused and position yourself at the root.*
2. **Leaf:** *If you are at a leaf, list the function.*
3. **Decide case:** *If there are no unused edges leading out from the vertex, go to Step 4; otherwise, go to Step 5.*
4. **Backtrack:** *If you are at the root, STOP; otherwise, return to the vertex just above this one and go to Step 3.*
5. **Decision:** *Select the leftmost unused edge out of this vertex, mark it used, follow it to a new vertex and go to Step 2.*

If you cannot easily visualize the entire route followed by this algorithm, take the time now to apply this algorithm to the decision tree for $\underline{2}^3$ shown in Figure 3.1 and verify that it produces all eight functions in lex order.

Because Step 1 refers to all leaves, the algorithm may be impractical in its present form. This can be overcome by keeping a “decision sequence” which is updated as we move through the tree. Here’s the modified algorithm.

Theorem 3.6 Systematic traversal algorithm (programming version) *The following procedure systematically visits the leaves in a tree from left to right by “walking” around the tree.*

1. **Start:** *Initialize the decision sequence to -1 and position yourself at the root.*
2. **Leaf:** *If you are at a leaf, list the function.*
3. **Decide case:** *Increase the last entry in the decision sequence by 1. If the new value equals the number of decisions that can be made at the present vertex, go to Step 4; otherwise go to Step 5.*
4. **Backtrack:** *Remove the last entry from the decision sequence. If the decision sequence is empty, STOP; otherwise go to Step 3.*
5. **Decision:** *Make the decision indicated by the last entry in the decision sequence, append -1 to the decision sequence and go to Step 2.*

In both versions of the algorithm, Step 4 is labeled “Backtrack.” What does this mean? If you move your pencil around a tree, this step would correspond to going toward the root on an edge that has already been traversed in the opposite direction. In other words, backtracking refers to the process of moving along an edge *back* toward the root of the tree. Thinking in terms of the decision sequence, backtracking corresponds to undoing (i.e., backtracking on) what is currently the last decision made.

If we understand the structure of the decision tree well enough, we can avoid parts of the algorithm. In the next example, we eliminate the descent since only one element in the Gray code changes each time. In the example after that, we eliminate the decision sequence for strictly decreasing functions. In both cases, we use `empty` to keep track of the state of the decision sequence. Also, Step 2 is avoided until we know we are at a leaf.

Example 3.13 Listing Gray coded subsets In Example 3.12 we looked at a Gray code for listing all elements of an n element set. Since there are only two decisions at each vertex, the entries in the decision sequence will be 0 or 1 (but they usually *do not* equal the entries in the Gray code). Here is the code with s_i being the decision sequence and g_i the Gray code.

```

Procedure GraySubsets( $n$ )
  For  $i = 1$  to  $n$ :                               /* Set up first leaf */
     $s_i = 0$ 
     $g_i = 0$ 
  End for
  empty = FALSE
  While (empty=FALSE):
    Output  $g_1, \dots, g_n$ .                       /* Step 2 */
    For  $i$  from  $n$  to 1 by  $-1$ :                       /* Steps 3 & 4 (moving up) */
      If ( $s_i = 0$ ) then
         $s_i = 1$ 
         $g_i = 1 - g_i$                                /* Step 5 (moving right) */
        For  $j$  from  $i + 1$  to  $n$ :
           $s_j = n + 1 - j$                            /* Steps 3 & 5 (moving down) */
        End for
        Goto ENDCASE
      End if
      empty = TRUE
    Label ENDCASE
  End for
End while
End

```

The statement $g_i = 1 - g_i$ changes 0 to 1 and vice versa. Since the Gray code changes only one entry, we are able to move down without changing any of the g_j values.

You may find this easier to understand than the presentation in Example 3.12. In that case, why don't we just throw that example out of the text? Although the code may be easy to follow, there is no reason to believe that it lists all subsets. Example 3.12 contains the proof that the algorithm does list all subsets, and the proof given there requires the recursive construction of the gray code based on reversing the order of the output for subsets of an $n - 1$ element set. Once we know that it works, we do the backtracking using the fact that only one g_i is changed at each output. Thus, we can forget about the construction of the Gray code as soon as we know that it works and a little about the decision tree. \square

Example 3.14 Listing strictly decreasing functions In Example 3.8 we studied ranking and unranking for strictly decreasing functions from \underline{k} to \underline{n} , which correspond to the k element subsets of \underline{n} . Now we want to list the subsets by traversing the decision tree.

Suppose we are at the subset given by $d_1 > \dots > d_k$. Recall that, since the d_i are strictly decreasing and $d_k \geq 1$, we must have $d_i \geq k + 1 - i$. We must back up from the leaf until we find a vertex that has unvisited children. If this is the $(i - 1)$ st vertex in the list, its next child after d_i will be $d_i - 1$ and the condition just mentioned requires that $d_i - 1 \geq k + 1 - i$. Here is the code for the traversal

```

Procedure Subsets( $n, k$ )
  For  $i$  from 1 to  $k$ :  $d_i = n + 1 - i$  End for
  empty = FALSE
  While (empty=FALSE):
    Output  $d_1, \dots, d_k$ .                /* Step 2 */
    For  $i$  from  $k$  to 1 by  $-1$ :              /* Steps 3 & 4 (moving up) */
      If ( $d_i > k + 1 - i$ ) then
         $d_i = d_i - 1$                       /* Step 5 (moving right) */
        For  $j$  from  $i + 1$  to  $k$ :
           $d_j = n + 1 - j$                   /* Steps 3 & 5 (moving down) */
        End for
        Goto ENDCASE
      End if
      empty = TRUE
    Label ENDCASE
  End for
End while
End   □

```

So far in our use of decision trees, it has always been clear what decisions are reasonable; i.e., lead, after further decisions, to a solution of our problem. This is because we've looked only at simple problems such as generating *all* permutations of \underline{n} or generating *all* strictly decreasing functions in \underline{n}^k . Consider the following problem.

How many permutations f of \underline{n} are such that

$$|f(i) - f(i + 1)| \leq 3 \quad \text{for } 1 \leq i < n? \quad 3.3$$

It's not at all obvious what decisions are reasonable in this case. For instance, when $n = 9$, the partially specified one line function 124586 cannot be completed to a permutation.

There is a simple cure for this problem: We will allow ourselves to make decisions which lead to "dead ends," situations where we cannot continue on to a solution. With this expanded notion of a decision tree, there are often many possible decision trees that appear reasonable for doing something. We'll look at this a bit in a minute. For now, let's look at our problem (3.3). Suppose that we're generating things in lex order and we've reached the vertex 12458. What do we do now? We'll simply continue to generate more of the permutation, making sure that (3.3) is satisfied for that portion of the permutation we have generated so far. The resulting portion of the tree that starts at 1,2,4,5,8 is shown in Figure 3.12. Each vertex is labeled with the part of the permutation after 12458. The circled leaves are solutions.

Our tree traversal algorithm given at the start of this section requires a slight modification to cover our extended decision tree concept where a leaf need not be a solution: Change Step 2 to

2'. Solution?: If you are at a solution, take appropriate action.

How can there be more than one decision tree for generating solutions in a specified order? Suppose someone who was not very clever wanted to generate all permutations of \underline{n} in lex order.

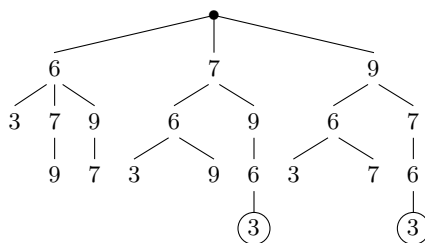


Figure 3.12 The portion of the decision tree for (3.3) that begins 1,2,4,5,8. The decision that led to a vertex is placed at the vertex rather than on the edge. The circled leaves are solutions.

He might program a computer to generate all functions in $\underline{n}^{\underline{n}}$ in lex order and to then discard those functions which are not permutations. This leads to a much bigger tree because n^n is much bigger than $n!$, even when n is as small as 3. A somewhat cleverer friend might suggest that he have the program check to see that $f(k) \neq f(k-1)$ for each $k > 1$. This won't slow down the program very much and will lead to only $n(n-1)^{n-1}$ functions. Thus the program should run faster. Someone else might suggest that the programmer check at each step to see that the function produced so far is an injection. If this is done, nothing but permutations will be produced, but the program may be much slower. Someone more knowledgeable might suggest a way to convert a decision sequence to a permutation using the ideas of the previous section. This would give the smallest possible tree and would not take very long to run. You might try to figure out how to do that.

The lesson to be learned from the previous paragraph is that there is often a trade off between the size of the decision tree and the time that must be spent at each vertex determining what decisions to allow. (For example, the decision to allow only those values for $f(k)$ which satisfy $f(k) \neq f(k-1)$.) Because of this, different people may develop different decision trees for the same problem. The differences between computer run times for different decision trees can be truly enormous. By carefully limiting the decisions, people have changed problems that were too long to run on a supercomputer into problems that could be easily run on a personal computer.

We'll conclude this section with two examples of backtracking of the type just discussed.

Example 3.15 Latin squares An $n \times n$ *Latin square* is an $n \times n$ array in which each element of \underline{n} appears exactly once in each row and column. Let $L(n)$ be the number of $n \times n$ Latin Squares. Finding a simple formula, or even a good estimate, for $L(n)$ is an unsolved problem. How can we use backtracking to compute $L(n)$ for small values of n ?

The number of Latin Squares increases rapidly with n , so anything we can do to reduce the size of the decision tree will be a help. Here's a way to cut our work by a factor of $n!$. Let's agree to rearrange the columns of a Latin Square so that the first row always reads $1, 2, 3, \dots, n$. We'll say such a square is "first row ordered." Given a first row ordered square, we can permute the columns in $n!$ ways, each of which leads to a different Latin Square. Hence $L(n)$ is $n!$ times the number of first row ordered Latin Squares.

By next rearranging the rows, we can get the entries in the first column in order, too. If we're sloppy, we might think this gives us another factor of $n!$. This is not true because 1 is already at the top of the first column due to our ordering of the first row. Hence only the second through n th positions are arbitrary and so we have a factor of $(n-1)!$.

Let's organize what we've got now. We'll call a Latin Square in *standard form* if the entries in the first row are in order and the entries in the first column are in order. Each $n \times n$ Latin Square in standard form is associated with $n!(n-1)!$ Latin Squares which can be obtained from it by permuting all rows but the first and then permuting all columns. The standard form Latin Squares

1	1 2	1 2 3	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
	2 1	2 3 1	2 1 4 3	2 1 4 3	2 3 4 1	2 4 1 3
		3 1 2	3 4 1 2	3 4 2 1	3 4 1 2	3 1 4 2
			4 3 2 1	4 3 1 2	4 1 2 3	4 3 2 1

Figure 3.13 The standard Latin squares for $n \leq 4$.

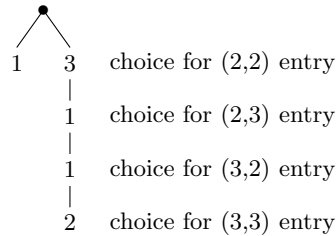


Figure 3.14 A decision tree for 3×3 Latin squares.

for $n \leq 4$ are shown in Figure 3.13. It follows that $L(1) = 1 \times 1! 0! = 1$, $L(2) = 1 \times 2! 1! = 2$, $L(3) = 1 \times 3! 2! = 12$ and $L(4) = 4 \times 4! 3! = 576$.

We'll set up the decision tree for generating Latin Squares in standard form. Fill in the entries in the array in the order $(2,2)$, $(2,3)$, \dots , $(2,n)$, $(3,1)$, \dots , $(3,n)$, \dots , (n,n) ; that is, in the order we read—left to right and top to bottom. The decision tree for $n = 3$ is shown in Figure 3.14, where each vertex is labeled with the decision that leads to it. The leaf on the bottom right corresponds to the standard form Latin Square for $n = 3$. The leaf on the left is the end of a path that has died because there is no way to choose entry $(2,3)$. Why is this? At this point, the second row contains 1 and 2, and the third column contains 3. The $(2,3)$ entry must be different from all of these, which is impossible. \square

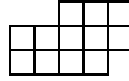
Example 3.16 Arranging dominoes A backtracking problem is stated in Example 4 at the beginning of this part: We asked for the number of ways to systematically arrange 32 dominoes on a chessboard so that each one covers exactly two squares and every square is covered.


If the squares of the board are numbered systematically from 1 to 64, we can describe any placement of dominoes by a sequence of 32 h 's and v 's: Place dominoes sequentially as follows. If the first unused element in the sequence is h , place a horizontal domino on the first unoccupied square and the square to its right. If the first unused element in the sequence is v , place a vertical domino on the first unoccupied square and the square just below it. Not all sequences correspond to legal arrangements because some lead to overlaps or to dominoes off the board. For a 2×2 board, the only legal sequences are hh and vv . For a 2×3 board, the legal sequences are hvh , vhh and vvv . For a 3×4 board, there are eleven legal sequences as shown in Figure 3.15.

To find these sequences in lex order we used a decision tree for generating sequences of h 's and v 's in lex order. Each decision is required to lead to a domino that lies entirely on the board and does not overlap another domino. The tree is shown in Figure 3.16. Each vertex is labeled with the choice that led to the vertex. The leaf associated with the path $vhvvv$ does not correspond to a covering. It has been abandoned because there is no way to place a domino on the lower left square of the board, which is the first free square. Draw a picture of the board to see what is happening.

Our systematic traversal algorithm can be used to traverse the decision tree without actually drawing it; however, there is a slight difficulty: It is not immediately apparent what the possible decisions at a given vertex are. This is typical in backtracking problems. (We slid over it in the

3.3.3. Draw a decision tree for covering the following board with dominoes.



3.3.4. Draw a decision tree for covering a 3×3 board using the domino and the L-shaped 3×3 square pattern: .

Notes and References

Although listing is a natural adjunct to enumeration, it has not received as much attention as enumeration until recently. Listing, ranking and unranking have emerged as important topics in connection with the development of algorithms in computer science. These are now active research areas in combinatorics.

The text [1] discusses decision trees from a more elementary viewpoint and includes applications to conditional probability calculations. The text by Stanton and White [3] is at the level of this chapter. Williamson [5; Chs.1,3] follows the same approach as we do, but explores the subject more deeply. A different point of view is taken by Nijenhuis and Wilf [2, 4].

1. Edward A. Bender and S. Gill Williamson, *Mathematics for Algorithm and Systems Analysis*, Dover (2005).
2. Albert Nijenhuis and Herbert S. Wilf, *Combinatorial Algorithms*, Academic Press (1975).
3. Dennis Stanton and Dennis White, *Constructive Combinatorics*, Springer-Verlag (1986).
4. Herbert S. Wilf, *Combinatorial Algorithms: An Update*, SIAM (1989).
5. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).