"A nickel ain't worth a dime anymore."

— Yogi Berra

"You know that I write slowly. This is chiefly because I am never satisfied until I have said as much as possible in a few words, and writing briefly takes far more time than writing at length."

— Karl Friedrich Gauss
(1777-1855)

Lecture V
THE GREEDY APPROACH

An algorithmic approach is called "greedy" when it makes decisions for each step based on what seems best at the current step. Moreover, once a decision is made, it is never revoked. It may seem that this approach is rather limited. Nevertheless, many important problems have special features that allow optimal solutions using this approach. Since we do not revoke our greedy decisions, such algorithms tend to be simple and efficient.

greedy with no regrets

To make this concept of "greedy decisions" concrete, suppose we have some "gain" function $G(x)$ which quantifies the gain we expect with each possible decision x. View the algorithm as making a sequence $x_1, x_2, \ldots, x_n$ of decisions, where each $x_i \in X_i$ for some set X_i of feasible choices at the ith step. Greediness amounts to choosing the $x_i \in X_i$ which maximizes the value $G(x)$.

The greedy method is supposed to exemplify the idea of "local search". But closer examination of greedy algorithms will reveal some global information being used. Such global information is usually minimal. Typically it amounts to some global sorting step. Indeed, the preferred data structure for delivering this global information is the priority queue.

We begin with three simple classes of greedy problems: a toy version of bin packing, the coin changing problem and problems involving intervals. Next we discuss the Huffman coding problem and minimum spanning trees. An abstract setting for the minimum spanning tree problem is based on **matroid theory** and the associated **maximum independent set problem**. This abstract framework captures the essence of a large class of problems with greedy solutions.

§1. Joy Rides and Bin Packing

We start with an example of a greedy algorithm to solve a simple problem which we call **linear bin packing**. The problem is, however, related to the major topic of bin packing in combinatorial algorithms.

¶1. Amusement Park Problem. Suppose we have a joy ride in an amusement park where riders arrive in a queue. We want to assign riders into cars, where the cars are empty as they arrive and we can only load one car at a time. Each car has a weight limit $M > 0$. The number of riders in a car is immaterial, as long as their total weight is $\leq M$ pounds. We may assume that no rider has weight $> M$. A key constraint in this problem is that we must make a decision for rider as they arrive at the head of the queue. This is called the **online requirement**. For instance, if $M = 400$ and the weights (in pounds) of the riders in the queue are

$$(30, 190, 80, 210, 100, 80, 50), \tag{1}$$

then we can put the riders into cars in the following groups:

$$S_1 : (30, 190, 80), (210, 100, 80), (50).$$

Solution S_1 uses three cars (the first two cars have the 3 riders each, and the last car has 1 rider). It is the solution given by the "greedy algorithm" which fills each car with as many riders as possible before loading the next car. Here are two other non-greedy solutions that uses the same number of cars:

$$S_2 : (30, 190), (80, 210), (100, 80, 50).$$

$$S_3 : (30, 190)(80, 210, 100), (80, 50).$$

¶2. General Bin Packing. The joy ride problem is an instance of the following prototype bin packing problem: *given a collection of items, to place them into as few bins as possible.* Each item is characterized by its weight (a positive real number) and the bins are identical, with a limited capacity. More precisely, we are given a multiset set $S = \{w_1, \ldots, w_n\}$ of positive weights, and a bin capacity $M > 0$. We want to partition S into a minimum number of subsets such that the total weight in each subset is at most M. We may assume that each $w_i \le M$. Unlike the joy ride problem, the weights can be reordered in any way we like. A solution to this general bin packing problem is also called a **globally optimal solution**.

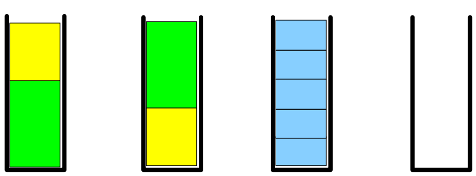


Figure 1: Bin packing solution.

E.g., if $S = \{1, 1, 1, 3, 2, 2, 1, 3, 1\}$ and $M = 5$ then one solution is $\{3, 2\}, \{2, 3\}, \{1, 1, 1, 1, 1\}$, illustrated in Figure 1. This solution uses 3 bins, and it is clearly optimal since each bin is filled to capacity. Unlike the joy ride problem, there is no linear constraint on the order of packing the weights, and no online constraint. Optimality in this general setting is very hard to achieve: *there are no known polynomial time algorithms for the general bin packing problem.* Put another way, we have converted a hard problem into a simple problem by introducing additional constraints on the possible solutions.

Formally, the joy ride problem is given an integer M and a sequence $w = (w_1, w_2, \ldots, w_n)$ of non-negative weights. We assume that the input (M, w) is **valid** in the sense that $M \ge w_i > 0$ for each $i = 1, \ldots, n$. A **linear solution** simply breaks w into some $m \ge 1$ groups,

$$(w_1, w_2, \ldots, w_{t(1)}),\ (w_{t(1)+1}, \ldots, w_{t(2)}),\ \ldots,\ (w_{t(m-1)+1}, \ldots, w_n)$$

where the groups are determined by a sequence of **breakpoints**

$$0 = t(0) < t(1) < t(2) < \cdots < t(m) = n. \tag{2}$$

The solution is **feasible** if each group satisfies

$$M \ge w_{t(i-1)+1} + w_{t(i-1)+2} + \cdots + w_{t(i)}$$

(for $i = 1, \ldots, m$). A feasible solution is **optimal** if m is minimum over all feasible solutions. The **linear bin packing problem** is to compute an optimal linear solution for any input M, w. For instance, the greedy algorithm on the input $w = (1, 1, 1, 3, 2, 2, 1, 3, 1), M = 5$ leads to the solution

the ith group corresponds to the ith car in the joy ride.

$$C_1 = \{1, 1, 1\}, \quad C_2 = \{3, 2\}, \quad C_3 = \{2, 1\}, \quad C_4 = \{3, 1\}.$$

Since this solution uses more than 3 bins, it is suboptimal for the general bin packing problem. Nevertheless, this solution is optimal for linear bin packing.

¶3. Greedy Algorithm. Let us code up the Greedy Algorithm for the linear bin packing problem (a.k.a. joy ride problem). Let $w = (w_1, w_2, \ldots, w_n)$ be the input sequence of weights. Let C denote a container (or car) that is being filled with elements of w, and W be the sum of the weights in C. Initially, $W \leftarrow 0$ and $C \leftarrow \emptyset$.

> Greedy Algorithm for Linear Bin Packing:
> Input: valid input (M, w) where $w = (w_1, \ldots, w_n)$.
> Output: A sequence of containers $C_1, C_2, \ldots, C_m$ representing an optimal linear solution.
> ▷ *Initialization*
> $C \leftarrow \emptyset$, $W \leftarrow 0$
> ▷ *Loop*
>   for $i = 1$ to n
>   ▷ *INVARIANT:* $M \ge W = \sum_{w \in C} x$
>   if $(i = n$ or $M < W + w_i)$
>   Output C
> $C \leftarrow \emptyset$, $W \leftarrow 0$
> $W \leftarrow W + w_i$; $C \leftarrow C \cup \{w_i\}$.

This greedy algorithm may be called the **first fit algorithm**.

¶4. Optimality of Greedy Algorithm. It may not be obvious why the greedy algorithm produces an optimal linear solution. In any case, it is instructive to prove that this is so. We use natural induction. Suppose the greedy algorithm outputs k cars with the weights

$$(w_1, \ldots, w_{t(1)}), (w_{t(1)+1}, \ldots, w_{t(2)}), \ldots, (w_{t(k-1)+1}, \ldots, w_{t(k)})$$

where $t(k) = n$. This defines the following sequence of breakpoints

$$1 \le t(1) < t(2) < \cdots < t(k) = n.$$

We will compare the greedy solution with some optimal solution with breakpoints

$$1 \le s(1) < s(2) < \cdots < s(\ell) = n.$$

By assumption of optimality, we have

$$\ell \le k. \tag{3}$$

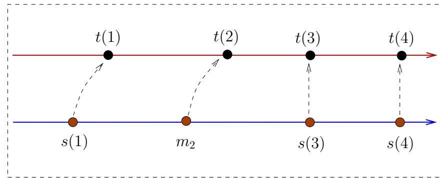
Using induction on i, we claim that

$$s(i) \le t(i) \tag{4}$$

for $i = 1, \ldots, \ell$. The base case $i = 1$ is clearly true (greediness ensures $t(1)$ achieves the maximum among first break points of feasible solutions).

For $i > 1$, suppose $s(i-1) \le t(i-1)$ holds by induction hypothesis. By way of contradiction, suppose that $s(i) > t(i)$. Hence

$$s(i-1) \le t(i-1) < t(i) < s(i). \tag{5}$$



The i-th car in the optimal solution has weight equal to $w_{s(i-1)+1} + \cdots + w_{s(i)}$, but this weight (according to (5)) is at least

$$w_{t(i-1)+1} + \cdots + w_{t(i)} + w_{t(i)+1}. \tag{6}$$

Note that this expression is meaningful since $t(i)+1 \le s(i) \le n$ for $i \le \ell$. By definition of the greedy algorithm, the sum in (6) must exceed M (otherwise the greedy algorithm would have added $w_{t(i)+1}$ to its ith car). This contradiction concludes our proof of (4).

From (4), we have $s(\ell) \le t(\ell)$. But since $s(\ell) = n \ge t(\ell)$, we conclude that $s(\ell) = t(\ell) = n$. Thus $k = \ell$, i.e., the greedy method is optimal.

¶5. How good is Linear Bin Packing? Given a sequence $w = (w_1, \ldots, w_n)$ of positive weights, we want to compare its optimal solutions when viewed as a linear bin packing instance, and when viewed as a general bin packing instance (¶2). Let $G_1(w)$ be the number of bins used by the greedy algorithm and $Opt(w)$ be the number of bins used by an optimal algorithm that does not have to respect the linear ordering in w.

Theorem 1. *For any weight sequence w,*

$$Opt(w) \ge 1 + \lfloor G_1(w)/2 \rfloor \tag{7}$$

For each n, there is a sequence of n weights for which (7) is an equality.

Proof. Suppose $G_1(w) = k$. Let the weight of ith output bin be W_i for $i = 1, \ldots, k$. The following inequality holds:

$$W_i + W_{i+1} > M \tag{8}$$

where M is the bin capacity. To see this, note that the first weight v to be put into the $i+1$st bin by the greedy algorithm must satisfy $W_i + v > M$. This implies (8) since $W_{i+1} \ge v$. Summing up (8) for $i = 1, 3, 5, \ldots, 2\lfloor k/2 \rfloor - 1$, we obtain $\sum_{i=1}^{k} W_i > \lfloor k/2 \rfloor M$. The last inequality implies that the optimal solution needs more than $\lfloor k/2 \rfloor$ bins, i.e., $Opt(w) \ge 1 + \lfloor k/2 \rfloor$. This proves (7).

To see that the inequality (7) is sharp, consider the following[1] weight sequence of length n:

$$w = \begin{cases} (1, \frac{1}{n}, 1, \frac{1}{n}, \ldots, 1, \frac{1}{n}) & \text{if } n = even, \\ (\frac{1}{n}, 1, \frac{1}{n}, \ldots, 1, \frac{1}{n}) & \text{if } n = odd. \end{cases}$$

If the bin capacity is $M = 1$, then the greedy solution uses n bins, but clearly $Opt(w) = 1 + \lfloor n/2 \rfloor$.

Q.E.D.

Thus linear bin packing can be optimally solved in $O(n)$ time. If the weights are arbitrary real numbers, this $O(n)$ bound is based on the real RAM computational model of Chapter 1.

[1] Thanks to my student Jason Y. Lee (2008) for simplifying my example.

¶6. General Bin Packing. Let us consider the general bin packing problem. The preceding lemma says that the optimal solution for linear bin packing is a 2-approximation for general bin packing: $G_1(w) \le 2 \cdot Opt(w)$.

We now derive a 1.5-approximation. It is still a greedy idea, but with two differences: first, this will be an **offline algorithm** because we must first sort the input weights w. Second, the bins are no longer packed linearly.

Greedy Algorithm for General Bin Packing:
  Input: valid input (M, w) where $w = (w_1, \ldots, w_n)$.
  Output: Bin packing solution $(B_1, B_2, \ldots, B_m)$ for (w, M)
1. Sort the weights w so that (after renumbering), we have
    $w_1 \ge w_2 \ge \cdots \ge w_n$.
2. Initialize n empty bins, $B_1, B_2, \ldots, B_n$.
3. For $i = 1, \ldots, n$,
4.   *place w_i into the first bin B_j that it fits into.*
5. Return the set of non-empty bins $(B_1, \ldots, B_m)$.

Refer to this as the **First Fit Algorithm** for General Bin Packing. To implement Step 4, we could use a while loop, checking $B_1, B_2, B_3, \ldots$ in this order until we find a bin that can accept w_i. Each iteration of the while-loop is $O(n)$ time, with overall complexity $O(n^2)$. But let us improve Step 4 to $O(\log n)$, giving an overall complexity of $O(n \log n)$. We represent the bins as leaves of a (static) binary tree T of height $O(\lg n)$. The ith leaf stores the set of weights in bin B_i. Define the **residual capacity** of B_i to be $R_i := M - W_i$ where W_i is the sum of the weights in B_i. Each node u of T stores a key $u.\mathtt{key}$ which is the maximum residual capacity of the bins in the subtree at u. Initially, each W_i is zero, and so $u.\mathtt{key} = M$ for each node u. For each w_i, we can find the first bin B_j whose residual R_j is at least as large as w_i as follows:

Insert(w_i):
1. Initialize u to the root of T.
2. While (u is not a leaf)
3.   If ($u.\mathtt{left.key} \ge w_i$)
      $u \leftarrow u.\mathtt{left}$
4.   else
      $u \leftarrow u.\mathtt{right}$
5. $u.\mathtt{key} \leftarrow u.\mathtt{key} - w_i$ ◃ *u is a leaf*
6. While (u is not the root)
7. $u \leftarrow u.p$ ◃ *move to the parent*
8. $u.\mathtt{key} \leftarrow \max\{u.\mathtt{left.key}, u.\mathtt{right.key}\}$

The first while-loop (Line 2) finds the leftmost bin B_j where w_i can be inserted; the second while-loop (Line 6) updates the keys along the path from B_j to the root. The proof that this algorithm achieves a 1.5 approximation is left as an Exercise.

what if we really want to determine the exact value of $Opt(w)$? This can be done by using the linear bin packing solution as a subroutine: we just cycle through each of the $n!$ permutations of $w = (w_1, \ldots, w_n)$, and for each compute the greedy solution in $O(n)$ time. The optimal solution is among them. This yields an $\Theta(n \cdot n!) = \Theta((n/e)^{n+(3/2)})$ time algorithm. Here, we assume that we can generate all n-permutations in $O(n!)$ time. This is a nontrivial assumption; see §8 for how to do this.

The Θ-form of Stirling's approximation

We can improve the preceding algorithm by a factor of n, since without loss of generality, we may restrict to permutations that begins with an arbitrary w_1 (why?). Since there are $(n-1)!$ such permutations, we obtain:

Lemma 2. *The bin packing problem can be solved in $O(n!) = O((n/e)^{n+(1/2)})$ time in the real RAM model.*

We can further extend this idea to improve this complexity by additional factors of n (Exercise). Observe that by imposing restrictions on the space of possible solutions, we have turned a difficult problem like general bin packing into a feasible one like linear bin packing. The latter problem may be interesting on its own merit, but we see that it can also be used as a subroutine for solving the original problem.

¶7. Two-Car Loading. Consider an extension of linear bin packing where we simultaneously load two cars. Call these two cars the **front** and **rear cars**. This is a realistic scenario for joy rides in a Ferris wheel. This allows us to mildly violate the first-come first-serve policy: a rider may be assigned to the rear car, while the next rider in the queue may be assigned to the front car. But this is the worst that can happen (people coming behind you in the queue can never be ahead by more than one car). If neither car can accommodate the new rider, we must **dispatch** the front car, so that the rear car comes to the front position and a new car empty becomes the rear car. We continue to have the "online restriction", i.e., we must make the decision for each rider in the queue without knowledge of who comes afterward. As usual, we assume that decisions are **irrevocable**. Once a rider has been assigned a car, it cannot be changed.

We want to design a new policy G_2 for 2-car loading. The goal, as usual, is to minimize the number of cars used for any given input sequence w. If G_1 is the car loading policy represented by the original greedy algorithm (¶3), we want to compare G_2 to G_1. More precisely, we want to show that for all $w = (w_1, \ldots, w_n)$, we have

$$G_2(w) \le G_1(w) \tag{9}$$

where $G_i(w)$ is the number of cars used by policy G_i on input w. There is a trivial way to design G_2 to satisfy (9): just imitate G_1! This means (9) is actually an equality, but it is not very interesting. What we want is a policy G_2 where, in addition to (9), there are many inputs w where $G_2(w) < G_1(w)$. Hopefully, G_2 has quantifiable advantages.

We propose two possible loading policies for G_2. First, let us introduce a useful concept: if the total weight of the riders already in a car is W, then we say the car has **residual capacity** of $M - W$. Thus an empty car has residual capacity of M. If a new passenger has weight that is at most the residual capacity of a car, then we say the rider **fits** in that car.

- **(FF) First Fit Policy**: *Load each rider into the front car if it fits, otherwise load into the rear car if it fits. If neither fits, dispatch the front car.*
- **(BF) Best Fit Policy**: *Load each rider into the car with the "best fit", i.e., the car with the minimum residual capacity that fits the rider. if neither car fits, then dispatch the front car.*

Let $G_2(w)$ and $G_2'(w)$ denote (respectively) the number of cars loaded by the first fit and best fit policies. In the Exercise, we will compare G_2 and G_2'.

Let us focus on G_2, mainly because it is easier to analyze than G_2'. First we note that it can be strictly better than G_1: For instance, if $w = (30, 190, 80, 210, 90, 80, 50)$ and $M = 400$ is our original example in (1), then the first fit policy is an improvement: $G_2(w) = 2 < 3 = G_1(w)$.

To prove that (9), we need to generalize it to a stronger statement about "subsequences". Normally, w' is called a subsequence of $w = (w_1, \ldots, w_n)$ if w' can be obtained from w by dropping zero or more entries from w. E.g., $w' = (2, 3, 1)$ is a subsequence of $w = (2, 2, 1, 3, 1)$. But instead of dropping an entry, we can imagine replacing it by 0: thus $w' = (2, 3, 1)$ can be regarded as $(2, 0, 0, 3, 1)$ or $(0, 2, 0, 3, 1)$. For the following proof, we therefore define a **subsequence** of $w = (w_1, \ldots, w_n)$ to be any sequence $w' = (w_1', \ldots, w_n')$ where $0 \le w_i' \le w_i$ for each i.

Lemma 3. *If w' is a subsequence of a w,*

$$G_2(w') \le G_1(w). \tag{10}$$

Proof. We use induction on the number $G_1(w)$. Let $w = (w_1, \ldots, w_n)$ and $w' = (w_1', \ldots, w_n')$. If $G_1(w) = 1$, then clearly (10) holds (it actually holds with equality unless w' is identically 0). Suppose $G_1(w) \ge 2$, and let the first car load in the G_1 solution be the multiset $C = \{w_1, w_2, \ldots, w_i\}$ for some $i \ge 1$. So

$$G_1(w) = 1 + G_1(w_{i+1}, w_{i+2}, \ldots, w_n). \tag{11}$$

The first car load C' in the $G_2(w')$ solution clearly contains the multiset $\{w_1', \ldots, w_i'\}$. But C' might also contain additional elements from the sequence w'. To account for these elements, let w'' be the weight sequence that is obtained from $(w_{i+1}', w_{i+2}', \ldots, w_n')$ by setting to 0 any weight w_j' that is in C'. Thus, we have

$$G_2(w') = 1 + G_2(w''). \tag{12}$$

Clearly, w'' is a subsequence of $(w_{i+1}', \ldots, w_n')$, and hence w'' is a subsequence of $(w_{i+1}, \ldots, w_n)$. By induction hypothesis, we conclude that

$$G_2(w'') \le G_1(w_{i+1}, \ldots, w_n). \tag{13}$$

Thus inequality (10) now follows from (11), (12), and (13). **Q.E.D.**

What about lower bounds? Consider the following weight sequence w of length $3n$,

$$w = (a_1, b_1, c_1; a_2, b_2, c_2; a_3 \cdots c_{n-1}; a_n, b_n, c_n)$$

where $a_i = c_i = 0.5 + i\epsilon$, $b_i = 0.5 - i\epsilon + \delta$. In the Exercise, we ask you to show that for suitable small values for $\delta, \epsilon > 0$, we have $G_2(w)/Opt(w) \ge 1.5 - O(1/n)$. Here, Opt is the optimal solution for general bin packing of w.

We may further extend the 2-car loading framework:

- One extension is to allow decisions to be revoked. This means that, upon seeing the next rider in the queue, we are allowed to move one or more riders between the front and rear cars. An even stronger notion of revoking is to exchange a rider in one of the two loading cars with the rider at the head of the queue. Note that this stronger notion can be applied even in 1-car loading.

- Another extension is to assume that two loading cars are in "parallel tracks" (left or right tracks). That means we can dispatch either car first. Note that this is extension permits loading policies which are arbitrarily unfair in the sense that a rider may be placed into a car that is arbitrarily far ahead of someone who arrived earlier in the queue. So we might want to restrict the admissible loading policies.

As far as I know, these problems have not been analyzed.

Exercises

Exercise 1.1: Give a counter example to the greedy algorithm in case the w_i's can be negative. NOTE: the greedy algorithm is exactly as it was before; in particular it is an online algorithm. $\diamondsuit$

Exercise 1.2: We consider linear bin packing problem in which the weights w_i's can be negative. From the previous problem, we know that the greedy algorithm does not produce the optimal solution. How bad can the greedy solution be, compared to an offline solution for linear bin packing? Please quantify "badness" in some reasonable way, and construct examples that illustrates how bad it could be. $\diamondsuit$

Exercise 1.3: There are two places where our optimality proof for the greedy algorithm breaks down when there are negative weights. What are they? $\diamondsuit$

Exercise 1.4: Consider the following "generalized greedy algorithm" in case w_i's can be negative. A solution to linear bin packing be characterized by the sequence of breakpoints, $0 = n_0 < n_1 < n_2 < \cdots < n_k = n$ where the ith car holds the weights

$$[w_{n_{i-1}+1}, w_{n_i+2}, \ldots, w_{n_i}].$$

Here is a greedy way to define these indices: let n_1 to be the largest index such that $\sum_{j=1}^{n_1} w_j \le M$. For $i > 1$, define n_i to be the largest index such that $\sum_{j=n_{i-1}+1}^{n_i} w_j \le M$. Note that this algorithm is no longer "online". Either prove that this solution is optimal, or give a counter example. $\diamondsuit$

Exercise 1.5: Give an $O(n^2)$ algorithm for linear bin packing when there are negative weights. HINT: Assume that when you solve the problem for (M, w), you also solve it for each (M, w') where w' is a suffix of w. This is really the idea of dynamic programming (Chapter 7). $\diamondsuit$

Exercise 1.6: Improve the bin packing upper bound in Lemma 2 to $O((n/e)^{n-(1/2)})$. HINT: Repeat the trick which saved us a factor of n in the first place. Fix two weights w_1, w_2. Consider two cases: either w_1, w_2 belong to the same bin or they do not. $\diamondsuit$

Exercise 1.7: The previous exercise fixes two weights w_1, w_2.
(a) Generalize the idea by fixing any fixed number $k \ge 3$ of weights.
(b) Further generalize the idea by fixing a non-constant number $k = k(n)$ of weights. $\diamondsuit$

Exercise 1.8: We have the 2-car loading problem, but now imagine the 2 cars move along two independent tracks, say the left track and right track. Either car could be sent off before the other. We still make decision for each rider in an online manner, but our ith decision x_i now comes from the set $\{L, R, L^+, R^+\}$. The choice $x_i = L$ or $x_i = R$ means we load the ith rider into the left or right car (resp.), but $x_i = L^+$ means that we send off the left car, and put the i-th rider into a new car in its place. Similarly for $x_i = R^+$. Consider the following heuristic: let $C_0 > 0$ and $C_1 > 0$ be the residual capacities of the two open cars. Try to put w_i into the car with the smaller residual capacity. If neither fits, we send off the car with the smaller residual capacity (and put w_i into its replacement car). Call this the **best fit dual-track 2-car strategy**, and let $G''_2(w)$ be the number of cars used by this strategy on input w (say capacity is 1). This is open ended: compare $G''_2(w)$ to G_1 and G_2. ♢

Exercise 1.9: Construct examples $w = (w_1, \ldots, w_n)$ (for arbitrarily large n) such that $G_2(w)/Opt(w) > 1.5 - O(1/n)$. Here, $Opt(w)$ is the minimum number of bins used if you have no constraints (you can put any weight into any bin).

HINT: Use the example suggested in the text where $n = 3m$ and $w = (a_1, b_1, c_1; a_2, b_2, c_2; a_3, b_3, c_3; \cdots)$ where we have arranged the weights into groups of 3 separated by semi-colons (;). Also, suppose $a_i = c_i = 0.5 + i\epsilon$ and $b_i = 0.5 - i\epsilon + \delta$. You must choose suitable values of ϵ, δ so that $G_2(w) = n$, and $Opt(w) \le 2n + O(1)$. ♢

Exercise 1.10: Let $G_2(w)$ and $G'_2(w)$ denote (respectively) the number of cars used when loading according to the First Fit and Best Fit Policies.
(a) Show an example where $G_2(w) > G'_2(w)$.
(b) Show an example where $G_2(w) < G'_2(w)$. ♢

Exercise 1.11: (Open) Can you improve the lower bound of 1.5 in the previous exercise? One idea is to use an adaptive adversary. ♢

Exercise 1.12: In the text, we compare our 2-car loading policy against an optimal bin packing solution. Now we want to compare our 2-car loading policy with the performance of a **clairvoyant 2-car algorithm**. Clairvoyant means that the algorithm can see into the future (and thus knows the entire queue). However, it must still respect the online requirement – each rider must be assigned a car and this cannot be revoked later. ♢

Exercise 1.13: (Open ended) Explore the revoking of decisions in 1- or 2-car loading. ♢

Exercise 1.14: (Open ended) Quantify the improvements possible when loading 2 cars in parallel tracks instead of loading 2 cars in a single track. ♢

Exercise 1.15: Recall the 1.5 Approximation Algorithm for the general bin packing problem described in the text. The algorithmic idea was to first sort the input $w = (w_1, \ldots, w_n)$ so that $w_1 \ge w_2 \ge \cdots \ge w_n$, and then use a greedy packing method. For simplicity, we may assume the bin capacity to be 1 and $0 < w_i < 1$. These weights are placed into bins $B_1, B_2, \ldots, B_m$ for some $m \ge Opt(w)$. Our goal is to prove that $m \le 1.5 \cdot Opt(w)$.
(a) Give a careful implementation of the algorithm. Make it as simple as possible; you are allowed to use time $O(n^2)$.

(b) Assume that this algorithm uses $m > Opt(w)$ many bins. There is a critical index i $(2 \le i \le n)$ such that w_i is placed input bin B_j where $j = Opt(w) + 1$. Let i_0 denote this critical index. Show that $w_{i_0} \le 1/3$.
(c) Show that $n - i_0 \le Opt(w)$ where i_0 is the critical index of part(b).
(d) Conclude that $m \le 1.5 \cdot Opt(w)$.
(e) For each n, produce examples $w_1, \ldots, w_n$ that shows that this factor of 1.5 is the best possible. ♢

Exercise 1.16: Weights with structure: suppose that the input weights are of the form $w_{i,j} = u_i + v_j$ and $(u_1, \ldots, u_m)$ and $(v_1, \ldots, v_n)$ are two given sequences. So w has mn numbers. Moreover, each group must have the form $w(i, i', j, j')$ comprising all $w_{k,\ell}$ such that $i \le k \le i'$ and $j \le \ell \le j'$. Call this a "rectangular group". We want the sum of the weights in each group to be at most M, the bin capacity. Give a greedy algorithm to form the smallest possible number of rectangular groups. Prove its correctness. ♢

Exercise 1.17: Two Dimensional Bin Packing: suppose the bins are unit squares, and the weights are boxes of dimensions $x_i \times y_i$ $(i = 1, \ldots, n)$. Assume $0 \le x_i \le 1$ and $0 \le y_i \le 1$. We write $w_i = (x_i, y_i)$ in this case. Give a heuristic for greedy bin packing, where box w_i must be assigned to the bin without knowing the later boxes w_j $(j > i)$. Moreover, once we place w_i, we are not allowed to rearrange it's placement If w_i cannot be placed, we must close the current bin and get a new bin for w_i. NOTE: this is a difficult problem. ♢

Exercise 1.18: For $0 < r < 1$, let $Opt(r)$ denote the maximal number of identical disks of radius r that can be packed into a unit disk.
(a) Device a greedy online algorithm for this problem, and analyze its performance relative to $Opt(r)$,
(b) How good is your algorithm when $r = 1/2$? ♢

Exercise 1.19: A **vertex cover** for a bigraph $G = (V, E)$ is a subset $C \subseteq V$ such that for each edge $e \in E$, at least one of its two vertices is contained in C. A **minimum vertex cover** is one of minimum size. Here is a greedy algorithm to finds a vertex cover C:

> Greedy Vertex Cover($G(V, E)$):
> 1. Initialize C to the empty set.
> 2. Choose a vertex $v \in V$ with the largest degree.
>   Add vertex v to the set C, and remove vertex v and all edges that are incident on it from graph G.
> 3. Repeat step 2 until the edge set is empty.
> 4. The final set C is a vertex cover of the original G.

(a) Show a graph G for which this greedy algorithm fails to give a minimum vertex cover.
(b) Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ where each x_i is associated with vertex $i \in V = \{1, \ldots, n\}$. Consider the following set of inequalities:

- For each $i \in V$, introduce the inequality
$$0 \le x_i \le 1.$$
- For each edge $i{-}j \in E$, introduce the inequality
$$x_i + x_j \ge 1.$$

If $\boldsymbol{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ satisfies these inequalities, we call $\boldsymbol{a}$ a **feasible solution**. If each a_i is either 0 or 1, we call $\boldsymbol{a}$ a $0-1$ feasible solution. Clearly, there is a bijective correspondence between the set of vertex covers and the set of $0-1$ feasible solutions. denote the corresponding $0-1$ feasible solution. We also write $|\boldsymbol{a}|$ for the sum $a_1 + \cdots + a_n$. Suppose $\boldsymbol{x}^* = (x_1^*, \ldots, x_n^*) \in \mathbb{R}^n$ is a feasible solution that minimizes the $|\boldsymbol{x}|$, i.e., for all feasible $\boldsymbol{x}$,

$$|\boldsymbol{x}^*| \le |\boldsymbol{x}|. \tag{14}$$

Call $\boldsymbol{x}^*$ an **optimum vector**. Construct a graph $G = (V, E)$ where the optimum vector $\boldsymbol{x}^*$ is not a $0-1$ feasible solution.
(c) Show that $\boldsymbol{x}^*$ is half-integer valued.
(d) Given a vector $\boldsymbol{x}$, define the **rounded vector** $\lfloor \boldsymbol{x} \rceil$ where each component x_i is rounded to $\lfloor x_i \rceil \in \{0, 1\}$. Note that rounding is usually defined up to some tie-breaking rule, viz., how to round 0.5. Show that, with a suitable tie-breaking rule, if $\boldsymbol{x}$ is feasible solution then $\lfloor \boldsymbol{x} \rceil$ is feasible $0-1$ solution.
(e) Suppose C^* is a minimum vertex cover. Show that $|\lfloor \boldsymbol{x}^* \rceil| \le 2|C^*|$.
(f) Consider the following trivial vertex cover algorithm:

```
TRIVIAL VERTEX COVER(G(V,E)):
  1. Initialize C to the empty set
  2. While E ≠ ∅
  3.    Remove u−v from E
  4.    C ← C ∪ {u}      ◁ arbitrarily pick u or v
  5.    Remove from E any edge incident on u or v
  6. Return C
```

Show that the output C is at most twice the size of the minimum vertex cover.
(g) Given that the trivial algorithm in (f) is also a 2-approximation algorithm for Vertex Cover, is there a reason to use the rounding algorithm $\lfloor \boldsymbol{x}^* \rceil$ of (e)? ♢

Exercise 1.20: Let $\boldsymbol{x}^*$ (14) be the optimum solution in the previous question. Show that coordinates of $\boldsymbol{x}^*$ must be be half-integers $(0, \frac{1}{2}, 1)$. ♢

Exercise 1.21: For $k \ge 1$, a k**-coloring** of a bigraph $G = (V, E)$ is a function $C : V \to \{1, \ldots, k\}$. The coloring is **proper** if $u - v \in E$ implies $C(u) \ne C(v)$. The **chromatic number** of G is the smallest k such that there exists a proper k-coloring of G; this number is denoted $\chi(G)$. Computing the chromatic number of bigraphs is one of the pure graph problems for which we do not know any polynomial time solution. But like the bin packing problem we can "linearize it": Given an enumeration $\boldsymbol{v} = (v_1, \ldots, v_n)$ of the vertices of G, we define the following **greedy coloring** of G relative to $\boldsymbol{v}$. For $i = 1, \ldots, n$, let $N_i = \{v_j : j < i, v_j - v_i \in E\}$ denote the set of vertices with index less than i that are adjacent to v_i. For each $i = 1, \ldots, n$, in order, we color v_i using the following rule: $C(v_1) = 1$. For $i \ge 2$, $C(v_i)$ is the smallest integer $k \ge 1$ such that $k \notin C(N_i) := \{C(v) : v \in N_i\}$.
(a) Show that if the enumeration is arbitrary, then the greedy coloring may be arbitrarily bad compared to $\chi(G)$.
(b) Suppose we first sort the vertices in order of increasing degrees. How bad can the greedy coloring be in this case?
(c) Show that there exists an enumeration whose greedy coloring is optimal.
(d) Using (d), establish an upper bound on the complexity of computing chromatic numbers. ♢

End Exercises

§2. Coin Changing Problem

In the US, if you bought an item for \$5.75, and you paid with a ten dollar bill, you are very likely to get back four singles (\$1.00 bills) and a quarter (25 cents) as change. You would be surprised if you got three singles and 5 quarters, for example. This expectation suggests that there is a deterministic algorithm that all cashiers use. In fact, it is a greedy algorithm! What problem does this greedy algorithm solve? It is popularly known as the **coin changing problem** although it is clear that bills are also counted as coins here.

greed and money, they go together

Let us explore some questions associated with coin changing. But first, we need some terminology. A **currency system** is a vector of increasing positive integers, $D = [d_1, d_2, \ldots, d_m]$ where each $d_i \in D$ is called a **denomination**. The number m of denominations is the **order** of the system. Notice that we use square brackets $[\cdots]$ for currency systems. For instance, the US currency system in daily use may be given by: $D = [1, 5, 10, 25, 100, 500, 1000, 2000, 5000, 10000]$ or more compactly,

$$D = [1, 5, 10, 25, \$1, \$5, \$10, \$20, \$50, \$100]. \tag{15}$$

The 2 dollar bill was introduced in 1862, discontinued in 1966, and reintroduced in 1976 for the US Bicentennial.

In (15), we have omitted the rare two-dollar bill which has been a US denomination since 1976. For $x \in \mathbb{N}$, a ***D*-solution** for x is any vector $s \in \mathbb{N}^m$ such that the dot product $\langle s, D\rangle := \sum_{i=1}^m s_i D_i$ equals x. Call x the **value** of s. We say D is **complete** if every positive integer has a D-solution. It is easy to see that D is complete iff $d_1 = 1$. In other words, the humble penny is what makes our currency system complete.

Two D-solutions s, s' are **equivalent** denoted $s \equiv s'$, if they have the same value. For example, if $s = (0, 0, 0, 1, 4, 0, 0, 0, 0, 0)$ is a solution in the US currency system (15), then its value is $\langle s, D\rangle = \$4.25$ (the change we expected in the initial example). An equivalent solution would be $s' = (0, 0, 0, 5, 3, 0, 0, 0, 0, 0)$, i.e.,

$$(0, 0, 0, 1, 4, 0, 0, 0, 0, 0) \equiv (0, 0, 0, 5, 3, 0, 0, 0, 0, 0),$$

Henceforth, we omit reference to D if it is understood (so we speak of "solution" instead of "D-solution", etc).

Call s a **greedy solution for** x if s is lexicographically the largest among solutions for x: that means that if s' is another solution, then the *last* non-zero entry in the vector difference $s - s'$ is positive. For example, $s' = (0, 0, 0, 5, 3, 0, 0, 0, 0, 0)$ then s is lexicographically larger than s'. Note that we look at last (not first) non-zero entry entry because we order the vector D from smallest to largest denomination.

¶8. The Canonicity Problem. The **size** of a solution $s = (s_1, \ldots, s_m)$ is given by $|s| := \sum_{i=1}^d s_i$. Thus "size" generalizes the "number of coins" we get in change during a transaction. An **optimal solution** for x is any solution s for x of minimum size. Let $\mathrm{Opt}_D(x)$ (resp., $\mathrm{Grd}_D(x)$) denote any optimal (resp., the greedy) solution, assuming x is a multiple of d_1. Clearly, we have $|\mathrm{Opt}_D(x)| \le |\mathrm{Grd}_D(x)|$. We say D is **canonical** if $|\mathrm{Grd}_D(x)| = |\mathrm{Opt}_D(x)|$ for all $x \in \mathbb{N}$ that is a multiple of d_1. *The key open problem in coin changing is to characterize canonical systems.*

the problem to solve!

Besides canonicity, another desirable property is uniqueness: D has **uniqueness** if $\mathrm{Opt}_D(x)$ is unique for every x that is a multiple of d_1. For example, the system $(1, 2, 3)$ is a canonical system, but it is non-unique since $(1, 0, 1) \equiv (0, 2, 0)$.

To show non-canonicity, we need counter examples. If $|\mathrm{Grd}_D(x)| > |\mathrm{Opt}_D(x)|$, we call x a **counter example** for (the canonicity of) D. Suppose x is a minimum counter example for $D = [d_1, d_2, \ldots, d_m]$, i.e., any value less than x is no counter example. Tien and Hu noted that if $\mathrm{Opt}(x) = (s_1^*, \ldots, s_m^*)$ is an optimum solution for such an x then

$$s_i \cdot s_i^* = 0 \qquad \text{(for all } i = 1, \ldots, m) \tag{16}$$

where $\mathrm{Grd}_D(x) = (s_1, \ldots, s_m)$. Suppose not. Say $s_i \cdot s_i^* > 0$. Then we can replace x by $x' = x - d_i$ to get a strictly smaller counter example. Why? Clearly, $\mathrm{Grd}(x')$ is obtained from $\mathrm{Grd}(x)$ by subtracting 1 from the i-th component. Also, $|\mathrm{Opt}(x')| \le |\mathrm{Opt}(x)| - 1$. We know that $|\mathrm{Opt}(x)| < |\mathrm{Grd}(x)|$. Thus $|\mathrm{Opt}(x')| \le |\mathrm{Opt}(x)| - 1 < |\mathrm{Grd}(x)| - 1 = |\mathrm{Grd}(x')|$. Thus x' is also a counter example, contradicting the minimality of x.

Let $q(D) = (q_1, q_2, \ldots, q_m)$ where $q_i = \lceil d_{i+1}/d_i \rceil$ for each $i = 1, \ldots, m-1$. Also let $q_m = \infty$. Thus $q(D)$ is roughly the multiple by which successive denominations increase. Then

$$\mathrm{Grd}(x) \le q(D). \tag{17}$$

This means that if $\mathrm{Grd}(x) = (s_1, \ldots, s_m)$ then $s_i \le q_i$ for all i. By way of contradiction, suppose $s_i > q_i$ for some i. Clearly $i \le m-1$. Then $s_i d_i > q_i d_i \ge d_{i+1}$. We could have obtained a solution $s' = (s'_1, \ldots, s'_m)$ for x in which

$$s'_j = \begin{cases} s_i d_i - d_{i+1} & \text{if } \ j = i \\ 1 + s_{i+1} & \text{if } \ j = i+1 \\ s_j & \text{if } \ j > i+1. \end{cases}$$

Thus s' is lexicographically larger than s. This contradicts the assumption that $\mathrm{Grd}_D(x)$ is lexicographically largest among all solutions.

¶9. Why is the US Currency System canonical? Experience suggests that the US Currency system is canonical. But how do we prove this? Call the currency systems D' an **extension** of D if D is a prefix of D'. Consider two kinds of extensions of $D = [d_1, \ldots, d_m]$:

have you ever met a counter example?

- **Type A extension** is $D' = [d_1, \ldots, d_m, d_{m+1}]$ where $d_{m+1} = d_m \cdot q$ for some $q \ge 2$.
- **Type B extension** is $D' = [d_1, \ldots, d_m, d_{m+1}, d_{m+2}]$ where $d_{m+1} = d_m \cdot q$ for some $q \ge 2$, and $d_{m+2} = d_{m+1} \cdot q' + d_m \cdot r$ for some $q' \ge 1$ and $r \ge 1$.

Call (q, q', r) a set of **parameters** of the Type B extension. If $r > q$, then $(q, q'+1, r-q)$ is also a set of parameters for the extension. By repeating this process, we conclude that there is a set of parameters in which $r < q$. Call this the **reduced set of parameters** for the Type B extension.

Trivially, $D = [d_1]$ is canonical for any $d_1 \ge 1$. We see that $D' = [1, 5]$ is a Type A extension of $[1]$ and $D'' = [1, 5, 10, 25]$ is a Type B extension of $[1, 5]$ with parameters $(q, q', r) = (2, 2, 1)$. In fact, the US currency system (15) is a sequence of Type A and Type B extensions of $[1]$. Indeed, this remains true even if we add the two dollar bill to the system. We now characterize how canonicity is preserved under these extensions:

Theorem 4. *Let D be a canonical system, D' be a Type A extension of D, and D'' be a Type B extension of D with reduced parameters (q, q', r).*
(i) If D is canonical, then D' is canonical.
(ii) If D is canonical, then D'' is canonical iff $r < q \le r + q'$.
(iii) If D is uniquely canonical, then D'' is uniquely canonical iff $r < q < r + q'$.

Proof. Let us initially assume $D = [q_1]$ (we will see that this is without loss of generality). Then $D' = [d_1, qd_1]$ $(q \ge 2)$ and $D'' = [d_1, qd_1, (qq'+r)d_1]$ $(r < q)$.
(i) Suppose x is the minimum counter example for D' where $q \ge 2$. If the greedy D'-solution for x is $\mathrm{Grd}_{D'}(x) = (s'_1, s'_2)$ then $s'_2 \ge 1$. If $\mathrm{Opt}_{D'}(x) = (s^*_1, s^*_2)$ then by property (16), $s^*_2 = 0$. Thus, we can view $\mathrm{Opt}_{D'}(x)$ as a D-solution for x. Since D is canonical, this proves that $|\mathrm{Opt}_{D'}(x)| \ge |\mathrm{Grd}_D(x)|$. Thus,

$$\begin{array}{rcll} |\mathrm{Grd}_{D'}(x)| & > & |\mathrm{Opt}_{D'}(x)| & x \text{ is a counter example} \\ & \ge & |\mathrm{Grd}_D(x)| & \text{just noted} \\ & = & s'_1 + qs'_2 & \text{since } \mathrm{Grd}_D(x) = (s'_1 + qs'_2) \\ & = & |\mathrm{Grd}_{D'}(x)| + (q-1)s'_2 & \\ & > & |\mathrm{Grd}_{D'}(x)| & q \ge 2 \text{ and } s'_2 \ge 1 \end{array}$$

which is a contradiction.
(ii) Note that D' is canonical (by part(i)), and $r < q$ holds by definition of reduced parameters. We must show that $q \le r + q'$ holds iff D'' is canonical. There are two cases: **CASE 1**, $q > r + q'$: In this case, we must provide a counter example. This is readily provided by the x such that

$$\mathrm{Grd}_{D''}(x) = (q - r, 0, 1).$$

Then $(q-r, 0, 1) \equiv (0, q'+1, 0)$ and we see that $|(q-r,0,1)| = q-r+1 > q'+1 = |(0, q'+1, 0)|$.

CASE 2, $q \le r + q'$: by way of contradiction, assume there is a counter example. Kozen-Zaks (Exercise) shows that the minimum counter example x has the form $\mathrm{Grd}_{D''}(x) = (a, 0, 1)$ for some $a \ge 0$. But we know $(a, 0, 1) \equiv (a + r, q', 0)$. The optimal solution $s^* = (s^*_1, s^*_2, s^*_3)$ satisfies $s^*_3 = 0$ (by (16)). Since (s^*_1, s^*_2) is equal to $\mathrm{Grd}_{D'}(x)$ (by canonicity of D'), we deduce

$$s^* = \begin{cases} (a + r, q', 0) & \text{if } a + r < q \\ (a + r - q, 1 + q', 0) & \text{else.} \end{cases}$$

If $a + r < q$, $|s^*| = a + r + q' \ge a + 1 = |\mathrm{Grd}_{D''}(x)|$. If $a + r \le q$, $|s^*| = a + (r - q) + 1 + q' \ge a + 1 = |\mathrm{Grd}_{D''}(x)|$. In either case, we have shown that $|s^*| \ge |\mathrm{Grd}_{D''}(x)|$, i.e., the D''-greedy solution for x is optimal. This contradicts the assumption that x is a counter example.
(iii) This is similar to part(ii). We must show that $q < r+q'$ iff D'' is uniquely canonical. Again, consider two cases. **CASE 1**, $q \ge r + q'$. We see that $(q - r, 0, 1) \equiv (0, q' + 1, 0)$ implies that the greedy solution $(q-r, 0, 1)$ has size that is either greater than or equal to $(0, q'+1, 0)$. This shows it is either suboptimal or non-unique. This proves that D'' is not uniquely canonical. **CASE 2**, $q < r+q'$: We want to show that D'' is uniquely canonical. By way of contradiction, assume an x where $|\mathrm{Opt}_{D''}(x)| \le |\mathrm{Grd}_{D''}(x)|$. The same argument as before will lead to the contradiction that $|\mathrm{Opt}_{D''}(x)| > |\mathrm{Grd}_{D''}(x)|$.

The arguments for parts(i)-(iii) assume $D = [d_1]$. But it is easy to see that if D is an arbitrary canonical system, there is no change in any argument, except for slightly more tedious notations such as $s = (0, \ldots, 0, q - r, 0, q)$ instead of $s = (q - r, 0, q)$. **Q.E.D.**

As a result of this theorem, we conclude that the US currency system (15) is uniquely canonical. See also Kozen and Zaks [5, 2].

¶10. Historical Notes. Chang and Gill [3] seems to be the first to study the coin change problem algorithmically. Kozen and Zaks characterized canonical system of up to order $m = 3$ [5]; Cai and Zheng [2] characterized canonical systems of order up to $m = 5$.

Exercises

Exercise 2.1: In a certain country, its currency system was originally $D = [1, 5, 10, 25, 50, 100]$. A new king came along. As a mathematician, he wanted to honor $\pi = 3.1415\cdots$ and so he decreed a new denomination 314. Is the new currency system canonical? ♢

Exercise 2.2: Suppose you are a cashier at a checkout and has to give change to customers. You want to give out the minimum number of coins and notes. Assume that you have an infinite supply of coins/notes in each denomination.

nice assumption!

(a) What is the greedy algorithm for this?
(b) Assuming a US cashier giving change less than \$100. You have bills in denominations \$50, \$20, \$10, \$5, \$1 and common coins 25cents, 10cents, 5cents, 1cents. Can you prove the optimality of your greedy algorithm?
(c) Give a set of currency denominations in which your greedy algorithm is non-optimal. ♢

Exercise 2.3: A certain sovereign state had a complete and canonical currency system $D = (d_1, \ldots, d_m)$. After a period of hyper-inflation, the state decreed that its pennies ($d_1 = 1$) are no longer legal currency. Is the new currency still canonical? What if the next denomination d_2 is also no longer legal? ♢

Exercise 2.4: In 1971, the British denomination converted to a decimal system. The old system has these denominations:

$$\frac{1}{2}, 1, 3, 6, 12(\text{=shilling}), 24(\text{=florin}), 30(\text{=half-crown}), 60(\text{=crown}), 240(\text{=pound}).$$

(a) Show that the old system in non-canonical.
(b) Determine the largest possible value of $\mathrm{Grd}(x) - \mathrm{Opt}(x)$ in the old system. ♢

Exercise 2.5: (Cai-Zheng) Show that the coin system $D = [1, 2, 4, 5]$ is non-canonical but $D' = [1, 2, 4, 5, 8]$ is canonical. ♢

Yogi Berra was referring to this in the introductory quote!

Exercise 2.6: Show by a direct argument that the binary system $D = (1, 2, 4, \ldots, 2^m)$ is a canonical system that is also unique. ♢

Exercise 2.7: (Kozen-Zaks)
(a) Let $D = [1, d_2, d_3]$ where $d_3 = qd_2 + r$ and $0 \le r < d_2$. Show that $(q + 1)d_2$ is the minimum counter example.
(b) If $D = [1, d_2, \ldots, d_m]$ is non-canonical, the minimum counter example $s = (s_1, \ldots, s_m)$ satisfies $s_3 < x < s_{m-1} + s_m$. ♢

Exercise 2.8: (Panagiotis Karras) The following problem arises in "compressing databases". We are given a sequence $w = (w_1, \ldots, w_n)$ of numbers and some $\epsilon > 0$. We say a sequence $x = (x_1, \ldots, x_m)$ is an **ϵ-approximation** of w of **order** m if there is a sequence of m breakpoints (as in (2))

$$0 = t(0) < t(1) < t(2) < \cdots < t(m) = n$$

such that for each original number w_i, the unique x_j ($j = 1, \ldots, m$) such that $t(j-1) < i \le t(j)$ provides an ϵ-approximation to w_i in the sense that

$$|w_i - x_j| \le \epsilon.$$

Intuitively, this says that we can approximate the sequence w by a histogram with m steps. Let $Min(w, \epsilon)$ denote the minimum order of an ϵ-approximation of w. Design and prove an $O(n)$ greedy algorithm to compute the $Min(w, \epsilon)$. $\diamondsuit$

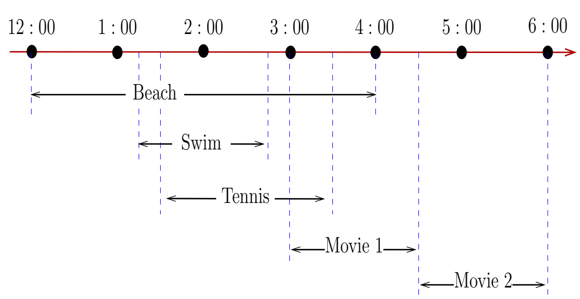
Exercise 2.9: Suppose that our supermarket checkout suddenly ran out of quarters. All the other denominations remain available.
(a) Is the greedy algorithm still optimal?
(b) More generally, we say a canonical currency system $D = [d_1, \ldots, d_m]$ is **robust** if it remains canonical under the loss of any single denomination d_i. Is the US currency system robust? $\diamondsuit$

End Exercises

§3. Interval Problems

An important class of greedy algorithms involves intervals. Typically, we think of an interval $I \subseteq \mathbb{R}$ as a time interval, representing the duration of some activity. We will use half-open intervals of the form $I = [s, f)$ where $s < f$ to represent an activity that starts at time s and finishes before time f. Recall that $[s, f)$ is the set $\{t \in \mathbb{R} : s \le t < f\}$, (we might say such intervals are "open ended"). Two activities **conflict** if their time intervals are not disjoint. We use half-open intervals instead of closed intervals so that the finish time of an activity can coincide with the start time of another activity without causing conflict. A set $S = \{I_1, \ldots, I_n\}$ of intervals is said to be **compatible** if the intervals in S are pairwise disjoint (i.e., the activities in S are mutually conflict-free).

We begin with the **activities selection problem**, originally studied by Gavril. Imagine you have the choice to do any number of the following fun activities in one afternoon:



beach	$12:00 - 4:00$,
swimming	$1:15 - 2:45$,
tennis	$1:30 - 3:20$,
movie	$3:00 - 4:30$,
movie	$4:30 - 6:00$.

You are not allowed to do two activities simultaneously. Assuming that your goal is to maximize your number of fun activities, which activities should you choose? Formally, the activities selection problem is this: *given a set*

$$A = \{I_1, I_2, \ldots, I_n\}$$

of intervals, to compute a compatible subset of S that is optimal. Here optimal means "of maximum cardinality". E.g., in the above fun activities example, an optimal solution would be to swim and to see two movies. It would be suboptimal to go to the beach. What would a greedy algorithm for this problem look like? Here is a generic version:

```
GENERIC GREEDY ACTIVITIES SELECTION:
Input: a set A of intervals
Output: S ⊆ A, a set of compatible intervals
▷ Initialization
    Sort A according to some numerical criterion.
        Let (I_1, ..., I_n) be the sorted sequence.
    Let S = ∅.
▷ Main Loop
    For i = 1 to n
        If S ∪ {I_i} is compatible, add I_i to S
    Return(S)
```

Thus, S is a partial solution that we are building up. At stage i, we consider interval I_i, to either **accept** or **reject** it. Accepting means to make it part of current solution S. Notice the difference and similarities between this greedy solution and the one for joy rides.

But what greedy criteria should we use for sorting? Here are four possibilities. Note that sometimes, a criterion leads to a tie. In this case, you break ties arbitrarily. Notice that we say "sort in increasing order", instead of the more accurate "sort in non-decreasing' order'. I prefer[2] this direct formulation, *always assuming the rule to break ties arbitrarily.*

(a) Sort I_i's in order of increasing finish times:
swim, tennis, beach, movie 1, movie 2

(b) Sort I_i's in order of increasing start times:
beach, swim, tennis, movie 1, movie 2

(c) Sort I_i's in order of duration where the duration of activity I_i is $f_i - s_i$. Note that *movie 1*, *movie 2* and *swim* are tied, but breaking ties arbitrarily:
movie 1, movie 2, swim, beach, tennis

(d) Sort I_i's in order of increasing conflict degree.
The conflict degree of I_i is the number of I_j's which conflict with I_i. If the conflict degree of I_i is zero, clearly we can always add I_i into our set. Thus the ordering is:
movie 2, movie 1 or swim, beach or tennis

My class (Fall 2011) voted on which of these criteria yields the optimal solution. Tally: (a) 11, (b) 0, (c) 16, (d) 20.

We can combine these criteria: e.g., if one criterion leads to a tie, we can use use another criterion to break ties. We now show that the first criterion (sorting by increasing finish times) leads to an optimal solution. In the Exercises, you will provide counter examples to the optimality of the other three criteria. Because of the possibility of ties, we distinguish two kinds of counter examples: "strong" counter examples do not depend on how ties are broken; "weak" counter example depends on how the ties are broken.

[2] The more accurate way of speaking slows down our processing of such statements.

Let us prove the optimality of sorting by increasing finish times. We use induction, reminiscent of the joy ride proof. Let $S = (I_1, I_2, \ldots, I_k)$ be the solution given by our greedy algorithm. If $I_i = [s_i, f_i)$, we may assume

$$f_1 < f_2 < \cdots < f_k.$$

Suppose $S' = (I'_1, I'_2, \ldots, I'_\ell)$ is an optimal solution where $I'_i = [s'_i, f'_i)$ and again $f'_1 < f'_2 < \cdots < f'_\ell$. By optimality of S', we have $k \le \ell$. CLAIM: We have the inequality $f_i \le f'_i$ for all $i = 1, \ldots, k$. We leave the proof of this CLAIM to the reader. Let us now derive a contradiction if the greedy solution is not optimal: assume $k < \ell$ so that I'_{k+1} is defined. Then

$$\begin{array}{rcll} f_k & \le & f'_k & \text{(by } CLAIM\text{)} \\ & \le & s'_{k+1} & \text{(since } I'_k, I'_{k+1} \text{ have no conflict)} \end{array}$$

and so I'_{k+1} is compatible with $\{I_1, \ldots, I_k\}$. This is a contradiction since the greedy algorithm halts after choosing I_k because there are no other compatible intervals.

What is the running time of this algorithm? In deciding if interval I_i is compatible with the current set S, it is enough to only look at the finish time f of the last accepted interval. This can be done in $O(1)$ time since this comparison takes $O(1)$ and f can be maintained in $O(1)$ time. Hence the algorithm takes linear time after the initial sorting.

¶11. Extensions, variations. There are many possible variations and generalizations of the activities selection problem. Some of these problems are explored in the Exercises.

- Suppose your objective is not to maximize the number of activities, but to maximize the total amount of time spent in doing activities. In that case, for our fun afternoon example, you should go to the beach and see the second movie.
- Suppose we generalize the objective function by adding a weight ("pleasure index") to each activity. Your goal now is to maximize the total weight of the activities in the compatible set.
- We can think of the activities to be selected as a uni-processor scheduling problem. (You happen to be the processor.) We can ask: what if you want to process as many activities as possible using two processors? Does our original greedy approach extend in the obvious way? (Find the greedy solution for processor 1, then find greedy solution for processor 2).
- Alternatively, suppose we ask: what is the minimum number of processors that suffices to do all the activities in the input set?
- Suppose that, in addition to the set A of activities, we have a set C of classrooms. We are given a bipartite graph with vertices $A \cup C$ and edges is $E \subseteq A \times C$. Intuitively, $(I, c) \in E$ means that activity I can be held in classroom c. We want to know whether there is an assignment $f : A \to C$ such that (1) $f(I) = c$ implies $(I, c) \in E$ and (2) $f^{-1}(c)$ is compatible. REMARK: scheduling of classrooms in a school is more complicated in many more ways. One additional twist is to do weekly scheduling, not daily scheduling.

Exercises

Exercise 3.1: We gave four different greedy criteria for the activities selection problem.
(a) We know one of them is optimal, but do not if sorting in increasing order degree-of-conflict is optimal. Show that the two of the other three criteria are suboptimal. EXTRA CREDIT: provide a *single* set of activities that serves as *strong* counter example to each of three criteria. It is possible to find a single counter example with 9 activities.
(b) Naturally, each of the above four criteria has an inverted version in which we sort in decreasing order. Again, one of these inverted criteria is actually optimal, and the other three suboptimal. Prove the optimality of one, and provide counter examples for the other three. ♢

Please draw figures as counter examples!

Exercise 3.2: Suppose the input $A = (I_1, \ldots, I_n)$ for the activities selection problem is already sorted, by increasing order of their start times, i.e., $s_1 \le s_2 \le \cdots \le s_n$. Give an algorithm to compute a optimal solution in $O(n)$ time. Show that your algorithm is correct. ♢

Exercise 3.3: Consider the activities selection problem with the following optimality criterion: to maximize the length $|A|$ of a set $A \subseteq S$ of activities. Define the **length** $|A|$ of a compatible set S to be the length of all the activities in S, where the length of an activity $I = [s, f)$ is just $|I| = f - s$. In case S is not compatible, its length is 0. Write $L(S)$ for the maximum length of any $A \subseteq S$. Let $A_{i,j} = \{I_i, I_{i+1}, \ldots, I_j\}$ for $i \le j$ and $L_{i,j} = L(A_{i,j})$.
(a) Show by a counter example that the following "dynamic programming principle" fails:

$$L_{i,j} = \max\{L_{i,k} + L_{k+1,j} : i \le k \le j-1\} \tag{18}$$

Would assuming that S is sorted by its start or finish times help?
(b) Give an $O(n \log n)$ algorithm for computing $L_{1,n}$. HINT: order the activities in the set S according to their finish times, say,

$$f_1 \le f_2 \le \cdots \le f_n.$$

For $i = 1, \ldots, n$, let L_i be the maximum length of a subset of $\{I_1, \ldots, I_i\}$. Use an incremental algorithm to compute $L_1, L_2, \ldots, L_n$ in this order. ♢

Exercise 3.4: Give a divide-and-conquer algorithm for the problem in previous exercise, to find the maximum length feasible solution for a set S of activities. (This approach is harder and less efficient!) ♢

Exercise 3.5: Interval problems often arises from scheduling.
(a) There is a 5 player game that lasts 48 minutes. In this game, any number of players can be swapped at any time. Suppose there are 8 friends what wants to play this game. Give a schedule for swapping players so that each of the 8 friends has the same amount of play time.
(b) Suppose there is a n player game that lasts t minutes. Again, any number of players can be swapped at any time. There are m friends who wants to play this game. Prove that there is always a schedule to let each friend have the same amount of playtime.
(c) Design an algorithm for (b) to schedule the swaps so that every one has the same amount of play time. ♢

End Exercises

§4. Huffman Code

The problem of compressing information is central to computing and information processing. We shall study one problem whose solution is based on the greedy paradigm. It is best to begin with an informally stated problem:

> (P) Given a string s of characters (or letters or symbols) taken from an alphabet Σ, choose a *variable length code* C for Σ so as to minimize the space to encode the string s.

Before making this problem precise, it is helpful to know some contexts for such a problem. A computer file may be regarded as a string s, so problem (P) can be called the **file compression problem**. Often, the characters in computer files are extended ASCII characters. This means the alphabet Σ has size $2^8 = 256$, and there is a standard way to represent each character by a 8-bit binary string, represented by a function $C_{asc} : \Sigma \to \{0,1\}^8$. The ASCII code is a *fixed-length binary code*, i.e., $|C_{asc}(x)| = 8$ for all $x \in \Sigma$, So the ASCII encoding of a file of m characters is a binary string of length $8m$. Can we do better?

Was Gauss referring to the difficulty of compression in the opening quotation?

The idea of Huffman coding is to use a **variable length code** in order to take advantage of the relative frequency of different characters. For instance, in typical English texts, the letters 'e' and 't' are most frequent and it is a good idea to use shorter length codes for them. On the other hand, infrequent letters like 'q' or 'z' could have longer length codes. An example of a variable length code is Morse code (see Notes at the end of this section). To see what additional properties are needed in variable-length codes, let us give some definitions:

A (binary) **code** for Σ is an injective function

$$C : \Sigma \to \{0,1\}^*.$$

A string of the form $C(x)$ $(x \in \Sigma)$ is called a **code word**. The string $s = x_1x_2 \cdots x_m \in \Sigma^*$ is then encoded as

$$C(s) := C(x_1)C(x_2)\cdots C(x_m) \in \{0,1\}^*. \tag{19}$$

This raises the problem of **decoding** $C(s)$, *i.e.*, recovering s from $C(s)$. For a general code C and s, one cannot expect unique decoding. Clearly, a basic requirement for decodability is that no code word $C(x)$ may be the empty string. Another requirement is that we should be able to detect the boundary between code words in $C(s)$. One solution is to introduce a new symbol '\$' and use it to separate successive $C(x_i)$'s. If we insist on using binary alphabet for the code, this forces us to convert, say, '0' to 00, '1' to 01 and '\$' to 11. This doubles the number of bits, and seems to be wasteful.

¶12. Prefix-free codes. Our preferred solution for unique decoding is to require that C be **prefix-free**. This means that if $a, b \in \Sigma$ are distinct letters then $C(a)$ is not a prefix of $C(b)$. It is not hard to see that the decoding problem has a unique solution under prefix-free codes. With suitable preprocessing (basically to construct the "code tree" for C, defined next) decoding can be done very simply in an on-line fashion.

We represent a prefix-free code C by a binary tree T_C with $|\Sigma|$ leaves. Each leaf in T_C is labeled by a character $b \in \Sigma$ such that the path from the root to b is encoded by $C(b)$ in the natural way: the path length has length $|C(b)|$, with the 0-bit (resp., 1-bit) representing a left (resp., right) branch. We call T_C a **code tree** for C. For $\Sigma| \ge 2$, if a code tree T_C has any internal node of degree 1, we can clearly "prune" that node to obtain a more efficient code. When T_C has no nodes to prune, we obtain a full binary tree. *For simplicity, we will henceforth assume that* $|\Sigma| \ge 2$ *and all code trees are full binary trees.* The case $|\Sigma| = 1$ needs special treatment since a full binary tree with 1 leaf would only admit the empty string as code word, contrary to our requirement for decodability. $|\Sigma| \ge 2!$

Figure 2 shows two such trees representing prefix codes for the alphabet $\Sigma = \{a, b, c, d\}$. The first code, for instance, corresponds to $C(a) = 00$, $C(b) = 010$, $C(c) = 011$ and $C(d) = 1$.

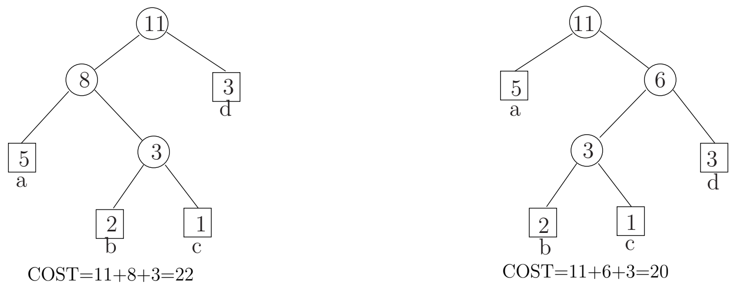


Figure 2: Two prefix-free codes and their code trees: assume $f(a) = 5, f(b) = 2, f(c) = 1, f(d) = 3$.

Returning to the informal problem (P), we can now interpret this problem as the construction of the best prefix-free code C for s, *i.e.*, the code that minimizes the length $|C(s)|$ of $C(s)$. It is easily seen that the only statistics important about s is the number, denoted $f_s(x)$, of occurrences of the character x in s. In general, call a function of the form

$$f : \Sigma \to \mathbb{N} \tag{20}$$

a **frequency function**. So we now regard the input data to our problem to be a frequency function $f = f_s$ rather than the string s. Relative to f, the **cost** of C is defined to be

$$COST(f, C) := \sum_{a \in \Sigma} |C(a)| \cdot f(a). \tag{21}$$

Clearly $COST(f_s, C)$ is the length of $C(s)$. Finally, the **cost** of f is defined by minimization over all choices of C:

$$COST(f) := \min_C COST(f, C)$$

over all prefix-free codes C on the alphabet Σ. A code C is **optimal** for f if $COST(f, C)$ attains this minimum. It is easy to see that an optimal code tree must be a *full* binary tree (i.e., non-leaves must have two children).

For the codes in Figure 2, assuming the frequencies of the characters a, b, c, d are $5, 2, 1, 3$ (respectively), the cost of the first code is $5 \cdot 2 + 2 \cdot 3 + 1 \cdot 3 + 3 \cdot 1 = 22$. The second code is better, with cost 20.

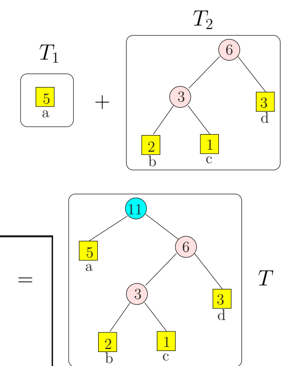
We now precisely state the informal problem (P) as the **Huffman coding problem**:

Given a frequency function $f : \Sigma \to \mathbb{N}$, find an optimal prefix-free code C for f.

Relative to a frequency function f on Σ, we associate a **weight** $W(u)$ with each node u of the code tree T_C: the weight of a leaf is just the frequency $f(x)$ of the character x at that leaf, and the weight of an internal node is the sum of the weights of its children. Let $T_{f,C}$ denote such a **weighted code tree**. In general, a weighted code tree is just a code tree together with weights on each node satisfying the property that the weight of an internal node is the sum of the weights of its children. For example, see Figure 2 where the weight of each node is written next to it. The **weight** of $T_{f,C}$ is the weight of its root, and its **cost** $COST(T_{f,C})$ defined as the sum of the weights of all its *internal* nodes. In Figure 2(a), the internal nodes have weights $3, 8, 11$ and so the $COST(T_{f,C}) = 3 + 8 + 11 = 22$. In general, the reader may verify that

$$COST(f, C) = COST(T_{f,C}). \tag{22}$$

We need the **merge** operation on code trees: if T_i is a code tree on the alphabet Σ_i $(i = 1, 2)$ and $\Sigma_1 \cap \Sigma_2$ is empty, then we can merge them into a code tree T on the alphabet $\Sigma_1 \cup \Sigma_2$ by introducing a new node as the root of T and with T_1, T_2 as the two children of the root. We may write $T = T_1 + T_2$. Note that we do not care whether T_1 or T_2 is the left of T. If T_1, T_2 are weighted code trees, the result T is also a weighted code tree.



tree merging

We now present a greedy algorithm for the Huffman coding problem:

Huffman Code Algorithm:
  Input: Frequency function $f : \Sigma \to \mathbb{N}$.
  Output: Optimal code tree T^* for f.
  1. Let S be a set of weighted code trees. Initially, S is the set of $n = |\Sigma|$ trivial trees, each tree having only one node representing a single character in Σ.
  2. While S has more than one tree,
     2.1. Choose $T, T' \in S$ with the minimum and the next-to-minimum weights, respectively.
     2.2. Merge T, T' and insert the result $T + T'$ into S.
     2.3. Delete T, T' from S.
  3. Now S has only one tree T^*. Output T^*.

A **Huffman tree** is defined as a weighted code tree that *could* be output by this algorithm. We say "could" because we regard the Huffman code algorithm as nondeterministic – when two trees have the same weight, the algorithm may pick either one in its choices. Let us illustrate the algorithm with a familiar 12-letter string, `hello world!`. The alphabet Σ for this string and its frequency function may be represented by the following two arrays:

arguably the famous string in computing

letter	h	e	l	o	⊔	w	r	d	!
frequency	1	1	3	2	1	1	1	1	1

Note that the exclamation mark (!) and blank space (⊔) are counted as letters in the alphabet Σ. The final Huffman tree is shown in Figure 3. The number shown inside a node u of the tree is the **weight** of the node. This is just sum of the frequencies of the leaves in the subtree at u. Each leaf of the Huffman tree is labeled with a letter from Σ.

Figure 3 shows the Huffman tree produced by our algorithm on our famous string. In addition, we display, next to each node, its "rank" $(0, 1, 2, \ldots, 16)$. The rank of a node specifies

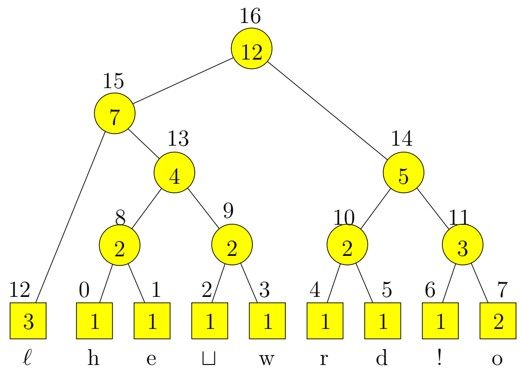


Figure 3: Huffman Tree for "`hello world!`": weights are written inside each node, but ranks $(0, 1, \ldots, 16)$ are beside the nodes.

the order in which nodes were extracted from the priority queue. For instance, the leaves `h` (rank 0) and `e` (rank 0) were the first two to be extracted in the queue. Their merge produced a node of rank 8. Note that the root is the last (rank 16) to be extracted from the queue. With this rank information, we can re-trace the step-by-step execution of the Huffman code algorithm. In the next section, we will exploit rank information in a more significant way. For the time being, we draw Huffman trees without paying much attention to its "orientation" i.e., the ordering of the 2 children of an internal node. When we treat dynamic Huffman tree in the next section, we will start paying attention to this issue. E.g., in Figure 3, the left child of the root has rank 15. Thus the Huffman code for the letter ℓ is 00. But we could also let the node of rank 14 be the left child of the root (this will be required in dynamic Huffman trees). The Huffman code of ℓ is then 10.

¶13. Implementation and complexity. The input for the Huffman algorithm may be implemented as an array $f[1..n]$ where $f[i]$ is the frequency of the ith letter and $|\Sigma| = n$. We construct the Huffman code tree T whose leaves are weighted by frequencies $f[a]$ $(a \in \Sigma)$, and internal nodes have weights that are the sum of the weights of the children. This algorithm can be implemented using a priority queue on a set S of binary tree nodes. Recall (§III.2) that a priority queue supports two operations, (a) inserting a keyed item and (b) deleting the item with smallest key. The frequency of the code tree serves as its key. Any balanced binary tree scheme (such as AVL trees in Lecture IV) will give an implementation in which each queue operation takes $O(\log n)$ time. Hence the overall algorithm takes $O(n \log n)$.

Now that we have constructed the code tree, let us see how to use it for coding and decoding. Decoding is easy: just start from the root of T, we follow a path to a leaf by turning left or right according to the scanned bits. Once we read a leaf, we can output the character stored at the leaf, and start again at the root. What about using T for encoding a string $s \in \Sigma^*$? For each $a \in \Sigma$, we need to determine the code word $C(a) \in \{0,1\}^*$. How could we do it? We must assume a separate "character map" `Cm` that takes each $a \in \Sigma$ to the corresponding leaf in T. Then by following parent pointers from `Cm`(a) to the root, we obtain the code word (in reverse).

¶14. Correctness. We show that the produced code C has minimum cost. This depends on the following simple lemma. Let us say that a pair of nodes in T_C is a **deepest pair** if they are siblings and their depth is equal to the depth of T_C. In a full binary tree, there is always a deepest pair.

Lemma 5 (Deepest Pair Property). *For any frequency function f, there exists a code tree T that is optimal for f, with the further property that T has a deepest pair b, c where b is some least frequent character, and c is some next-to-least frequent character.*

Proof. Suppose b, c are two characters at depths $D(b), D(c)$ (respectively) in a weighted code tree T. If we exchange the labels (b, c) and corresponding weights $(f(b), f(c))$ of these two nodes, we get a new code tree T' where

$$\begin{aligned} COST(T) - COST(T') &= f(b)D(b) + f(c)D(c) - f(b)D(c) - f(c)D(b) \\ &= [f(b) - f(c)][D(b) - D(c)] \end{aligned}$$

where f is the frequency function. If b has the least frequency then $f(b) - f(c) \leq 0$. And if $D(c)$ is the depth of the tree T then $D(b) - D(c) \leq 0$. Together, they imply

$$COST(T) - COST(T') \geq 0.$$

That is, the cost of the tree cannot increase when we move a least frequent characters to the deepest leaf. Hence if c, c' are the two characters labeling a deepest pair and and b, b' are the two least frequent characters, then our preceding argument shows that the exchanges $b \leftrightarrow b'$ and $c \leftrightarrow c'$ will not increase the cost of the tree. If the tree is already optimal, then this exchange preserves optimality. This proves that there exists an optimal tree with a deepest pair formed by two least frequent characters. **Q.E.D.**

Note that this lemma does not claim that every optimal code tree has the deepest pair property. See Exercise for a counter example.

We are ready to prove the correctness of Huffman's algorithm. Suppose by induction hypothesis that our algorithm produces an optimal code whenever the alphabet size $|\Sigma|$ is less than n. The basis case, $n = 1$, is trivial. Now suppose $|\Sigma| = n > 1$. After the first step of the algorithm in which we merge the two least frequent characters b, b', we can regard the algorithm as constructing a code for a modified alphabet Σ' in which b, b' are replaced by a new character $[bb']$ with modified frequency f' such that $f'([bb']) = f(b) + f(b')$, and $f'(x) = f(x)$ otherwise. By induction hypothesis, the algorithm produces the optimal code C' for f':

$$COST(f') = COST(f', C'). \tag{23}$$

From the code C', we can obtain a code C for Σ as follows: the code tree T_C is obtained from $T_{C'}$ by replacing the leaf $[bb']$ by an internal node with two children representing b and b', respectively. Thus we have

$$COST(f, C) = COST(f', C') + f(b) + f(b'). \tag{24}$$

By our deepest pair lemma, and using the fact that the COST is a sum over the weights of internal nodes, we conclude that

$$COST(f) = COST(f') + f(b) + f(b'). \tag{25}$$

[More explicitly, this equation says that if T is the optimal weighted code tree for f and T has the deepest pair property, then by removing the deepest pair with weights $f(b)$ and $f(b')$, we get an optimal weighted code tree for f'.] From equations (23), (24) and (25), we conclude $COST(f) = COST(f, C)$, *i.e.*, C is optimal. ■

¶15. Compact Coding for Transmission. We address the representation of a Huffman code C for the purposes of transmission. It is assumed that the representation will be a binary string α_C, and the main consideration is compactness, i.e., minimality of the length of α_C.

It is important to spell out some further assumptions about $C : \Sigma \to \{0,1\}^*$. First of all, the set Σ is only partially known. We must rely on some standard character set such as the ASCII set or Unicode (see Figure 8 and notes below). Let Σ_0 be[3] such a standard character set. *We assume that Σ is some subset of Σ_0*. For simplicity, we further assume that $\Sigma_0 \subseteq \{0,1\}^N$ for some fixed N. This N is the common knowledge used by both the transmitter and receiver. We stress that we allow Σ to be a proper subset of Σ_0. Typically, Σ is just the set of characters that actually occur in the string we are trying to compress and transmit.

To represent C, we first encode the shape of the code tree T_C. Once the shape of T_C is known, we just need to list the elements of Σ in the order they appear in the left-to-right listing of the leaves of T_C.

We give a progression of ideas that lead to the final compact coding of the shape of T_C. The initial idea is simple: let us prescribe a systematic way to traverse T. Starting from the root, we use a depth-first traversal, always go down the left child first. Each edge is traversed twice, initially downward and later upward. Then if we "spit" out a 0 for going down an edge and "spit" out a 1 for going up an edge, we would have faithfully output a description of the shape of T by the time we return to the root for the second time. Figure 4 illustrates this traversal of the Huffman tree of Figure 2(a), and shows resulting binary sequence

$$0010'0101'1101. \tag{26}$$

$4n - 4 = 12$ *bits!*

The $'$-marks here are only decorative, to help in parsing into blocks of 4 bits each. This scheme uses 2 bits per edge. If there are n leaves, there are $n-1$ internal nodes. Each internal nodes are associated with 2 outgoing edges, giving $2n-2$ edges. Each edge gives rise to 2 bits, and so the representation has $4n-4$ bits. We emphasize: *this representation depends on knowing that T is a full binary tree.*

Where have we exploited this fact?

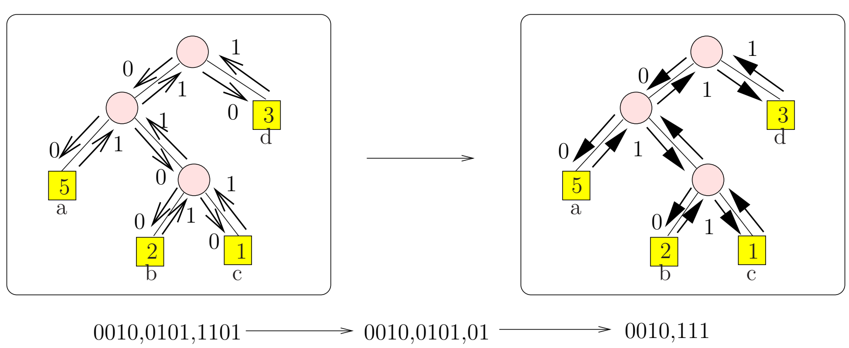


Figure 4: Compressed bit representation for the Huffman tree Figure 2a

Another remark is this: this encoding is **self-limiting** in the sense that, if we know the beginning of the encoding, then we would know when we reach the end of the encoding. This property is useful for many applications. This implies that the set of all such encodings is a prefix-free set.

[3] What does this really mean? Take the example of the English alphabet, 'A' to 'Z'. Each letter has many layers of meaning and variations that goes beyond its recognizable symbol. For our purposes however, it suffices to choose one canonical set of 26 symbols to represent this alphabet.

To improve this representation, observe that a contiguous sequence of ones can be replaced by a single 1 since we know where to stop when going upward from a leaf (we stop at the first node whose right child has not been visited). This also takes advantage of the fact that we have a full binary tree. Previously we used $2n-2$ ones. With this improvement, we only use n ones (corresponding to the leaves). The representation now has only $3n-2$ bits. Then (26) is now represented by

$$0010'0101'01. \tag{27}$$

$3n-2=10$ bits!

Finally, we note that each 1 is immediately followed by a 0 (since the 1 always leads us to a node whose right child has not been visited, and we must immediately go down to that child). The only exception to this rule is the final 1 when we return to the root; this final 1 is not followed by a 0. We propose to replace all such 10 sequences by a plain 1. Since there are n ones (corresponding to the n leaves), we would have eliminated $n-1$ zeros in this way. This gives us the final representation with $2n-1$ bits. The scheme (27) is now shortened to:

How about $01 \to 1$?

$$0010'111. \tag{28}$$

$2n-1=7$ bits!

See the final illustration in Figure 4. The final scheme (28) will be known as the **compressed bit representation** α_T of a full binary tree T. In case T is the code tree for a Huffman code C, we may denote the binary string α_T by α_C.

The above description assumes T has more than one node: in this case, α_T always begins with a 0 and ends with a 1. So the shortest such string is 011, represent a full binary tree with two leaves. If T has only one node, it is natural to represent it as $\alpha_T = 1$. Some additional simple properties are summarized as follows:

Lemma 6. *Let T be a full binary tree T with $n \ge 1$ leaves.*
(i) $|\alpha_T| = 2n-1$ with $n-1$ zeros and n ones.
(ii) The number of zeros is at least the number of ones in any proper prefix of α_T.
(iii) The set $S \subseteq \{0,1\}^*$ of all such compressed bit representation α_T forms a prefix-free set. This is an immediate consequence of the self-limiting property of our encodings.
(iv) There is a linear time algorithm that checks whether a given binary string s belongs to the set S, i.e., if s has the form α_T for some T.

We leave the proof as an Exercise. This has the following application:

Theorem 7. *There is a protocol to transmit a binary string β_C representing any Huffman code $C : \Sigma \to \{0,1\}^*$ on $|\Sigma| = n$ letters such that*
(i) The length of β_C is $(2n-1) + Nn = n(N+2) - 1$.
(ii) A receiver can recover the code C from β_C in linear time, without prior knowledge of Σ except that $\Sigma \subseteq \Sigma_0 \subseteq \{0,1\}^N$.

Proof. The string β_C has two parts: the first part is the compressed bit representation α_C. The second part is a list of the elements in Σ. The elements in this list are N-bit binary strings, and they appear in their order as labels of the n leaves of T_C. We have $|\alpha_C| = 2n-1$, and the listing of Σ uses nN bits. This proves (i). For part (ii), the receiver can use the prefix-free property of α_C to detect the end of α_C while processing β_C. In linear time, it could also reconstruct the shape of T_C and thus knows n. Since the receiver knows N, it can also parse each symbol of Σ in the rest of β_C. **Q.E.D.**

The encoding of Huffman trees in this lemma can be improved if we are allowed to swap subtrees. We can swap the children of internal nodes so that, in each level, the leaves all appear before the internal nodes. A full binary tree such that in each level, all the leaves appear to the

Letter	Code	Letter	Code
A	· —	B	— · · ·
C	— · — ·	D	— · ·
E	·	F	· · — ·
G	— — ·	H	· · · ·
I	· ·	J	· — — —
K	— · —	L	· — · ·
M	— —	N	— ·
O	— — —	P	· — — ·
Q	— — · —	R	· — ·
S	· · ·	T	—
U	· · —	V	· · · —
W	· — —	X	— · · —
Y	— · — —	Z	— — · ·
0	— — — — —	1	· — — — —
2	· · — — —	3	· · · — —
4	· · · · —	5	· · · · ·
6	— · · · ·	7	— — · · ·
8	— — — · ·	9	— — — — ·
Fullstop (.)	· — · — · —	Comma (,)	— — · · — —
Query (?)	· · — — · ·	Slash (/)	— · · — ·
BT (pause)	— · · · —	AR (end message)	· — · — ·
SK (end contact)	· · · — · —		

Table 1: Morse Code

left of all the internal nodes is said to be **canonical**. Such trees have a self-limiting encoding that uses only $n+2$ bits where n is the number of leaves (Exercise).

Remarks: The publication of the Huffman algorithm in 1952 by D. A. Huffman was considered a major achievement. This algorithm is clearly useful for compressing binary files. See "Conditions for optimality of the Huffman Algorithm", D.S. Parker (*SIAM J.Comp.*, 9:3(1980)470–489, *Erratum* 27:1(1998)317), for a variant notion of cost of a Huffman tree and characterizations of the cost functions for which the Huffman algorithm remains valid.

¶16. Notes on Morse Code. In the Morse[4] code, letters are represented by a sequence of dots and dashes: $a = \cdot -$, $b = - \ \cdot\cdot\cdot$ and $z = - \ - \ \cdot\cdot$. The code is also meant to be sounded: dot is pronounced '*dit*' (or '*di-*' when non-terminal), dash is pronounced '*dah*' (or '*da-*' when non-terminal). So the famous distress signal "S.O.S" is `di-di-di-da-da-da-di-di-dit`. Thus 'a' is $di - dah$, 'z' is $da - da - di - dit$. The code does not use capital or small letters. Here is the full alphabet:

Note that Morse code assigns a dot to `e` and a dash to `t`, the two most frequent English letters. These two assignments dash any hope for a prefix-free code. So how can do you send or decode messages in Morse code? Spaces! Since spaces are not part of the Morse alphabet, they have an informal status as an explicit character (so Morse code is not strictly a binary code). There are 3 kinds of spaces: space between *dit*'s and *dah*'s within a letter, space between letters, and space between words. Let us assume some **unit space**. Then the above three types of spaces are worth 1, 3 and 7 units, respectively. These units can also be interpreted as "unit time" when the code is sounded. Hence we simply say **unit** without prejudice. Next, the system of dots and dashes can also be brought into this system. We say that spaces are just "empty units", while *dit*'s and *dah*'s are "filled units". *dit* is one filled unit, and *dah* is 3 filled units. Of course, this brings in the question: why 3 and 7 instead of 2 and 4 in the above? Today, Morse code is still required of HAM radio operators and is useful in emergencies.

Exercises

[4] Samuel Finley Breese Morse (1791-1872) was Professor of the Literature of the Arts of Design in the University of the City of New York (now New York University) 1832-72. It was in the university building on Washington Square where he completed his experiments on the telegraph.

Exercise 4.1: Give a Huffman code for the string "`hello! this is my little world!`". ♢

Exercise 4.2: What is the length of the Huffman code for the string s = "`please compress me`". Show your hand computation. Do not forget the empty space character. ♢

Exercise 4.3: Consider the following letter frequencies:

$$a = 5, b = 1, c = 3, d = 3, e = 7, f = 0, g = 2, h = 1, i = 5, j = 0, k = 1, l = 2, m = 0,$$
$$n = 5, o = 3, p = 0, q = 0, r = 6, s = 3, t = 4, u = 1, v = 0, w = 0, x = 0, y = 1, z = 1.$$

Please determine the cost of the optimal tree. NOTE: you may ignore letters with the zero frequency. ♢

Exercise 4.4: Give an example of a prefix-free code $C : \Sigma \to \{0,1\}^*$ *and* a frequency function $f : \Sigma \to \mathbb{N}$ with the property that (i) $COST(C, f)$ is optimal, but (ii) C could not have arisen from the Huffman algorithm. HINT: you can choose $|\Sigma| = 4$. ♢

Exercise 4.5: True or False? If T and T' are two optimal prefix-free code for the frequency function $f : \Sigma \to \mathbb{N}$, then T and T' are isomorphic as unordered trees. Prove or show counter example. NOTE: a binary tree is an ordered tree because the two children of a node are ordered. ♢

Exercise 4.6: In the text, we prove that for any frequency function f, there is an optimal code tree in which there is a deepest pair of leaves whose frequencies are the least frequent and the next-to-least frequent. Consider this stronger statement: *if T is any optimal code tree for f, there must be a deepest pair whose frequencies are least frequent and next-to-least frequent.* Prove it or show a counter example. ♢

Exercise 4.7: Let $C : \Sigma \to \{0,1\}^*$ be any prefix-free code whose code tree T_C is a full-binary tree. Prove that there exists a frequency function $f : \Sigma \to \mathbb{N}$ such that C is optimal. ♢

Exercise 4.8:
(a) Draw the full binary tree corresponding to its compressed bit representations:

$$\alpha_1 = 0010'1100'1011'1 \qquad \alpha_2 = 0100'1001'0011'111$$

(b) What is α_T where T is the full binary tree with 6 leaves and every right child is a leaf.
(c) What is α_T where T is the full binary tree with 6 leaves and every left child is a leaf.
(d) What is α_T where T is the complete binary tree with 8 leaves. ♢

Exercise 4.9: Joe Smart suggested that we can slightly improve the compressed bit representation of full binary trees on n leaves as follows: since the first bit is always 0 and the last bit is always 1, we can use only $2n - 3$ bits instead of $2n - 1$. What are some issues that might arise with this improvement? ♢

Exercise 4.10: The text gave a method to represent any full binary tree T on n leaves using a binary string α_T with $2n-1$ bits. Clearly, not every binary string of length $2n-1$ represents a full binary tree. For instance, the first and last bits must be 0 and 1, respectively. Give a necessary and sufficient condition for a binary string to be a valid representation. ♢

Exercise 4.11: For any binary full tree T, we have given two representations: the array A_T and the bit string α_T. Give detailed algorithms for the following conversion problems:
(a) To construct the string α_T from the array A_T.
(b) To construct the array A_T from the string α_T.

♢

Exercise 4.12: Let T be a full binary tree on n leaves. Give an algorithm to convert its compressed bit representation $\alpha_T[1..2n-1]$ to a $4n-4$ array $B[1..4n-4]$ representing the traversal of T.

♢

Exercise 4.13: Suppose we want to represent an arbitrary binary tree, not necessarily full. HINT: there is a bijection between arbitrary binary trees and full binary trees. Exploit our compressed bit-representation of full binary trees. ♢

Exercise 4.14: (a) Prove (22).
(b) It is important to note that we defined $COST(T_{f,C})$ to be the sum of $f(u)$ where u range over the *internal* nodes of $T_{f,C}$. That means that if $|\Sigma| = 1$ (or $T_{f,C}$ has only one node which is also the root) then $COST(T_{f,C}) = 0$. Why does Huffman code theory break down at this point?
(c) Suppose we (accidentally) defined $COST(T_{f,C})$ to be the sum of $f(u)$ where u range over the *all* nodes of $T_{f,C}$. Where in your proof in (a) would the argument fail? ♢

Exercise 4.15: (Kraft Inequality) Let $0 \le d_1 \le d_2 \le \cdots \le d_n$ be the depths of the leaves in a binary tree with n leaves. Prove that

$$1 \ge \sum_{i=1}^{n} 2^{-d_i}. \tag{29}$$

Moreover, if the binary tree is a full binary tree, then the inequality is an equality. ♢

Exercise 4.16: (Elias) Let $bin(n)$ denote the standard binary encoding of $n \in \mathbb{N}$ and $len(n)$ be the length of this encoding. E.g., $bin(0,1,2,3,4) = (\epsilon, 1, 10, 11, 100)$ and $len(0,1,2,3,4) = (0,1,2,2,3)$. Here, ϵ is the empty string and we use the compact notation $bin(n_1, n_2, \ldots, n_k) := (bin(n_1), bin(n_2), \ldots, bin(n_k))$, etc.

We want a binary encoding of natural numbers, $rep : \mathbb{N} \to \{0,1\}^*$, with the following property: if $n_1, n_2, \ldots$ is any sequence of natural numbers, the binary string $rep(n_1)rep(n_2)\cdots$ is uniquely decodable. Such an encoding $rep(n)$ is **self-limiting** in the sense that whenever we know the start of $rep(n)$, we can also determine where it ends. Alternatively, the representation is prefix-free: if $n \ne n'$ then $rep(n)$ is not a prefix of $rep(n')$.

(a) Consider the following encoding scheme $rep_1 : \mathbb{N} \to \{0,1\}^*$ for natural numbers: $rep_1(0) = 1$ and for $n \geq 1$, $rep_1(n) = 0^{len(n)} bin(n)$. E.g., $rep_1(0,1,2,3,4) = (1, 0'1, 00'10, 00'11, 000'100)$. Note we use the prime mark ($'$) for decoration. What is $rep_1(99)$? What is the length of $\mathtt{rep}_1(n)$ as a function of n?

(b) Now consider $rep_2 : \mathbb{N} \to \{0,1\}^*$ where $rep_2(n) = rep_1(len(n))bin(n)$ for $n \geq 0$. E.g., $rep_2(0,1,2,3,4) = (1, 01'1, 0010'10, 0010'11, 0011'100)$. What is $rep_2(99)$? What is the length of $\mathtt{rep}_2(n)$ as a function of n?

(c) What is wrong with the suggestion to define $rep(n)$ recursively as follows:

$$rep(n) = rep(len(n))bin(n)$$

for all $n \geq n_0$ (for some n_0)? How can you fix this issue? How small can n_0 be, and what can you do for $n < n_0$? What is the length of your representation as a function of n? What is your representation of 99? For what values of n will your representation be shorter than $rep_2(n)$?

$\diamondsuit$

Exercise 4.17: Here is an idea to improve the compressed bit representation of Huffman code trees given in our text. Say a full binary tree T is **canonical** if at every level, every leaf must appear before any internal node in the left-right listing of the level. A Huffman code tree is canonical if its underlying tree is canonical.
(a) Show that every full binary tree can transformed into a canonical one by a sequence of swaps between pairs of nodes at the same level. NOTE: when we swap two nodes, we are really swapping the entire subtrees rooted at these two nodes.
(b) Show that swaps do not change the cost of a Huffman tree.
(c) Give an **improved compressed bit representation** for canonical Huffman trees.
(d) What is the range of sizes for your improved compressed bit representation for a canonical tree with n leaves? $\diamondsuit$

Exercise 4.18: (Gashlin 2012) The previous exercise gave an "improved compressed bit representation" for canonical Huffman trees. We now give a further improvement, but using a radically different approach: instead of focusing on the leaves, we focus on the internal nodes. An integer sequence of the form

$$\boldsymbol{n} = (n_0, n_1, \ldots, n_{d-1}) \tag{30}$$

is called the **profile** of a full binary tree of depth d and at level $i = 0, \ldots, d-1$, it has n_i internal nodes.
(a) Show that there is a bijection between canonical trees with N internal nodes and integer sequences of the form (30) that satisfies $N = \sum_{i=0}^{d-1} n_i$, $n_0 = 1$ and $1 \leq n_i \leq 2n_{i-1}$ $(i = 1, \ldots, d-1)$.
(b) Let the **leaf profile** of a full binary tree of depth d be the sequence $(\ell_1, \ell_2, \ldots, \ell_d)$ where ℓ_i is the number of leaves at level $i = 1, \ldots, d$. Given the profile (30) of a full binary tree, what is the corresponding leaf profile?
(c) Give a self-limiting encoding of profiles of canonical trees. HINT: use simple self-limiting encodings, but be sure we can detect the end of the encoding.
(d) Give an upper bound on the number $T(N)$ of canonical Huffman trees with N internal nodes.
(e) Give a lower bound on $T(N)$. $\diamondsuit$

Exercise 4.19: Below is President Lincoln's address at Gettysburg, Pennsylvania on November 19, 1863.

(a) Give the Huffman code for the string S comprising the first two sentences of the address. Also state the length of the Huffman code for S, and the percentage of compression so obtained (assume that the original string uses 7 bits per character). View caps and small letters as distinct letters, and introduce symbols for space and punctuation marks. But ignore the newline characters.
(b) The previous part was meant to be done by hand. Now write a program in your favorite programming language to compute the Huffman code for the entire Gettysburg address. What is the compression obtained?

```
Four score and seven years ago our fathers brought forth on this continent
a new nation, conceived in liberty and dedicated to the proposition that
all men are created equal.  Now we are engaged in a great civil war, testing
whether that nation or any nation so conceived and so dedicated can long
endure. We are met on a great battlefield of that war. We have come to
dedicate a portion of that field as a final resting-place for those who here
gave their lives that that nation might live. It is altogether fitting and
proper that we should do this.  But in a larger sense, we cannot dedicate,
we cannot consecrate, we cannot hallow this ground.  The brave men, living
and dead who struggled here have consecrated it far above our poor power to
add or detract. The world will little note nor long remember what we say
here, but it can never forget what they did here.  It is for us the living
rather to be dedicated here to the unfinished work which they who fought
here have thus far so nobly advanced. It is rather for us to be here
dedicated to the great task remaining before us -- that from these honored
dead we take increased devotion to that cause for which they gave the last
full measure of devotion -- that we here highly resolve that these dead
shall not have died in vain, that this nation under God shall have a new
birth of freedom, and that government of the people, by the people, for the
people shall not perish from the earth.
```

$\diamondsuit$

Exercise 4.20: Let $(f_0, f_1, \ldots, f_n)$ be the frequencies of $n+1$ symbols (assuming $|\Sigma| = n+1$). Consider the Huffman code in which the symbol with frequency f_i is represented by the ith code word in the following sequence

$$1, 01, 001, 0001, \ldots, \underbrace{00\cdots01}_{n-1}, \underbrace{00\cdots001}_{n}, \underbrace{00\cdots000}_{n}.$$

(a) Show that a sufficient condition for optimality of this code is

$$\begin{array}{rcl} f_0 & \ge & f_1 + f_2 + f_3 + \cdots + f_n, \\ f_1 & \ge & f_2 + f_3 + \cdots + f_n, \\ f_2 & \ge & f_3 + \cdots + f_n, \\ & \cdots & \\ f_{n-2} & \ge & f_{n-1} + f_n. \end{array}$$

(b) Suppose the frequencies are distinct. Give a set of sufficient and necessary conditions. $\diamondsuit$

Exercise 4.21: Suppose you are given the frequencies f_i in sorted order. Show that you can construct the Huffman tree in linear time. $\diamondsuit$

Exercise 4.22: (Representation of Binary Trees) In the text, we showed that a full binary tree on n leaves can be represented using $2n-1$ bits. Suppose T is an arbitrary binary tree, not necessarily full. With how many bits can you represent T? HINT: by extending T into a full binary tree T', then we could use the previous encoding on T'. ♢

Exercise 4.23: Huffman code is based on transmitting bits. Suppose we transmit in 'trits' (a base-3 digit). Then the corresponding 3-ary Huffman code $C : \Sigma \to \{0,1,2\}^*$ is represented by a 3-ary code tree T where each leaf is associated with a unique letter in Σ and each internal node has degree at most 3. If $f : \Sigma \to \mathbb{N}$ is a frequency function, this assigns a weight to each node of T: the leaf associated with $x \in \Sigma$ has weight $f(x)$, and each internal node has a weight equal to the sum of the weights of its children. The cost of T is defined as usual, as the sum of the weights of the internal nodes of T. We are interested in optimal trees T, i.e., whose cost cost is minimum.
(a) Show that in an optimal 3-ary code tree, there are no nodes of degree 1 and at most one node of degree 2. Furthermore, if a node has degree 2, it must have leaves as both of its children.
(b) Let T be a tree whose internal nodes have degrees 2 or 3. If there are d_i nodes of degree i $(i = 0, 2, 3)$ in T show that $d_0 = 1 + d_2 + 2d_3$.
(c) Show that there are optimal 3-ary code trees with this property: if $|\Sigma|$ is odd, there are no degree 2 nodes, and if $|\Sigma|$ is even, there is one degree 2 node. Moreover, if the unique node u of degree 2 we may assume its children have minimum frequencies among all the leaves.
(d) Give an algorithm for constructing an optimal 3-ary code tree and prove its correctness. ♢

Exercise 4.24: We consider the 4-ary version of the previous question. Let T be an optimum 4-ary code tree for some frequency function $f : \Sigma \to \mathbb{N}$.
(a) Give a short inductive proof of the following fact: Suppose T is *any* 4-ary tree on $n \geq 1$ leaves, and let N_d be the number of nodes with d children $(d = 0, 1, 2, 3, 4)$. Thus, $n = N_0$. Give a short inductive proof for the following formula: $n = 1 + N_2 + 2N_3 + 3N_4$.
(b) Show that if T is an optimal code tree, then $N_1 = 0$ and $3N_2 + 2N_3 \leq 4$, and every non-full internal node has only leaves as children and the depth of these leaves must equal the height of of T.
(c) Moreover, we can always transform T from part (b) into T' such that the corresponding degrees satisfy $N'_1 = 0$ and $N'_2 + N'_3 \leq 1$. Also, for any non-full internal node of T', its children have weights no larger than any other leaves.
(d) Suppose $r = (n-1) \,\mathbf{mod}\, 3$. So $r \in \{0, 1, 2\}$. Show how N'_2, N'_3 in part (b) is determined by r.
(e) Describe an algorithm to construct an optimal code tree from a frequency function f.
(f) Show the optimal 4-ary Huffman tree for the input string `hello world!`. Please state the cost of this optimal tree. ♢

Exercise 4.25: Further generalize the 3-ary Huffman tree construction to arbitrary k-ary codes for $k \geq 4$. ♢

Exercise 4.26: Suppose that the cost of a binary code word w is $z + 2o$ where z (resp. o) is the number of zeros (resp. ones) in w. Call this the **skew cost**. So ones are twice as expensive as zeros (this cost model might be realistic if a code word is converted into a sequence of dots and dashes as in Morse code). We extend this definition to the **skew cost** of a code C or of a code tree. A code or code tree is **skew Huffman** if it is optimum with respect to this skew cost. For example, see Figure 5 for a skew Huffman tree for alphabet $\{a, b, c\}$ and $f(a) = 3$, $f(b) = 1$ and $f(c) = 6$.

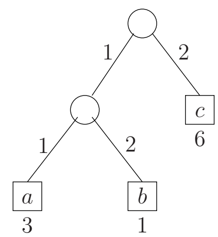


Figure 5: A skew Huffman tree with skew cost of 21.

(a) Argue that in some sense, there is no greedy solution that makes its greedy decisions based on a linear ordering of the frequencies.
(b) Consider the special case where all letters of the alphabet has equal frequencies. Describe the shape of such code trees. For any n, is the skew Huffman tree unique?
(c) Give an algorithm for the special case considered in (b). Be sure to argue its correctness and analyze its complexity. HINT: use an "incremental algorithm" in which you extend the solution for n letters to one for $n + 1$ letters. $\diamondsuit$

Exercise 4.27: (Golin-Rote) Further generalize the problem in the previous exercise. Fix $0 < \alpha < \beta$ and let the cost of a code word w be $\alpha \cdot z + \beta \cdot o$. Suppose α/β is a rational number. Show a dynamic programming method that takes $O(n^{\beta+2})$ time. NOTE: The best result currently known gets rid of the "+2" in the exponent, at the cost of two non-trivial ideas. $\diamondsuit$

Exercise 4.28: (Open) Give a non-trivial algorithm for the problem in the previous exercise where α/β is not rational. An algorithm is "trivial" here if it essentially checks all binary trees with n leaves. $\diamondsuit$

Exercise 4.29: The range of the frequency function f was assumed to be natural numbers. If the range is arbitrary integers, is the Huffman theory still meaningful? Is there fix? What if the range is the set of non-negative real numbers? $\diamondsuit$

Exercise 4.30: (Shift Key in Huffman Code) We want to encode small as well as capital letters in our alphabet. Thus 'a' and 'A' are to be distinguished. There are two methods to do this. (I) View the small and capital letters as distinct symbols. (II) Introduce a special "shift" symbol, and each letter is assumed to be small unless it is preceded by a shift symbol, in which case the following letter is capitalized. As input string for this problem, use the text of this question. Punctuation marks are part of this string, but there is only one SPACE character. Newlines and tabs are regarded as instances of SPACE. Two or more consecutive SPACE characters are replace by a single SPACE.
(a) What is the length of the Huffman code for our input string using method (I). Note that the input string begins with "We want to en..." and ends with "...ngle SPACE.".
(b) Same as part (a) but using method (II).
(c) Discuss the pros and cons of (I) and (II).
(d) There are clearly many generalizations of shift keys, as seen in modern computer keyboards. The general problem arises when our letters or characters are no longer indivisible units, but exhibit structure (as in Chinese characters). Give a general formulation of such extensions. $\diamondsuit$

End Exercises

§5. Dynamic Huffman Code

Here is the typical sequence of steps for compressing and transmitting a string s using the Huffman code algorithm:

(i) First make a pass over the string s to compute its frequency function.

(ii) Next compute a Huffman code tree T_C corresponding to some code C.

(iii) Using T_C, compute the compressed string $C(s)$.

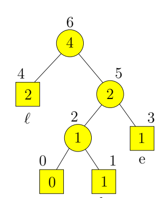
(iv) Finally, transmit the tree T_C (Theorem 7), together with the compressed string $C(s)$, to the receiver.

The receiver receives T_C and $C(s)$, and hence can recover the string s. Since the sender must process the string s in two passes (steps (i) and (iii)), the original Huffman tree algorithm is sometimes called the "2-pass Huffman encoding algorithm". There are two deficiencies with this 2-pass process: (a) Multiple passes over the input string s makes the algorithm unsuitable for realtime data transmissions. Note that if s is a large file, this require extra buffer space. (b) The Huffman code tree must be explicitly transmitted before the decoding can begin. We need some way to encode T_C. This calls for a separate algorithm to handle T_C in the encoding and decoding process.

An approach called "Dynamic Huffman coding" (or adaptive Huffman coding) overcomes these problems: there is no need to explicitly transmit the code tree, and it passes over the string s only once. In fact, it does not even have to pass over the entire string even once, but can transmit as much of the string as has been read! This property is important for transmitting continuous stream of data that has no apparent end (e.g., ticker tape, satellite signals). Two known algorithms for dynamic Huffman coding [10] are the **FGK Algorithm** (Faller 1973, Gallager 1978, Knuth 1985) and the **Lambda Algorithm** (Vitter 1987). The dynamic Huffman code algorithm can be used for data compression: for example, it is used[5] in the Unix utility `compress/uncompress`.

¶17. Sibling Property. In Dynamic Huffman Coding, the weighted code tree T must evolve as characters from the input string is read. It must evolve in two ways: not only does the frequency of letters in Σ increase over time, but Σ itself can grow as new letters are encountered. We need to update our representation of T as this happens. The key idea is the "sibling property" of Gallagher.

Assume T has $k \geq 0$ internal nodes. So it has $k+1$ leaves or $2k+1$ nodes in all. We say T has the **sibling property** if its nodes can be **ranked** from 0 to $2k$ satisfying:



siblings and ranks

(S1) (Weights are increasing with rank) If w_i is the weight of node with rank i, then $w_{i-1} \leq w_i$ for $i = 1, \ldots, 2k$.

[5] This particular utility has been replaced by better compression schemes.

(S2) (Siblings have consecutive ranks) The nodes with ranks $2j$ and $2j+1$ are siblings (for $j = 0, \ldots, k-1$).

For example, the weighted code tree in Figure 3 has been given the rankings $0, 1, 2, \ldots, 16$. We check that this ranking satisfies the sibling property. Note that the node with rank $2k$ is necessarily the root, and it has no siblings. In general, let $r(u)$ denote the rank of node u. If the weights of nodes are all distinct, then the rank $r(u)$ is uniquely determined by Property (S1).

Lemma 8. *Let T be weighted code tree. Then T is Huffman iff it has the sibling property.*

Proof. If T is Huffman then by definition, it is constructed by the Huffman code algorithm. We can rank the nodes in the order that nodes are extracted from the priority queue, and this ordering implies the sibling property. Conversely, the sibling property of T determines an obvious order for merging pairs of nodes to form a Huffman tree. **Q.E.D.**

¶18. Sibling Representation of Huffman Tree. We provide an array representation of Huffman trees that exploits the sibling property. Let T be a Huffman tree with $k+1 \ge 1$ leaves. Each of its $2k+1$ nodes may be identified by its rank, *i.e.*, a number from 0 to $2k$. Hence node i has rank i. We use two arrays

$$\mathtt{Wt}[0..2k], \qquad \mathtt{Lc}[0..2k]$$

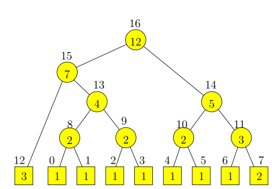
of length $2k+1$ where $\mathtt{Wt}[i]$ is the *weight* of node i, and $\mathtt{Lc}[i]$ is an even integer indicating the *left child* of node i. In case node i is a leaf, we may let $\mathtt{Lc}[i] = -1$. Alternatively, as will be done in these notes, we let $\mathtt{Lc}[i]$ store a character from the alphabet Σ; of course, this convention assumes that one can distinguish between elements in the set $0, \ldots, 2k$ and the characters in Σ. Thus, for a non-leaf i, its right child is given by $\mathtt{Lc}[i] + 1$, and $\mathtt{Lc}[i]$ is always an even integer. This, the left and right child of any node is a pair of the form $(2j, 2j+1)$ (for some j). We ensure that the root is node $2k$.

We stress that storing elements of Σ in $\mathtt{Lc}$ is is not essential, but it makes the examples more transparent. What *is* essential for our algorithms is the *inverse* representation that tells us, for each letter $x \in \Sigma$, which leaf in T contains x. Moreover, because of the dynamic nature of Σ, we need a more general mapping,

$$\mathtt{Cm} : \Sigma_0 \to \{-1, 0, 1, 2, \ldots, 2k\}$$

such that $\mathtt{Cm}[x] = i$ iff $Lc[i] = x \in \Sigma$, and $\mathtt{Cm}[x] = -1$ if $x \notin \Sigma$. Call $\mathtt{Cm}$ the **character map array**. Initially, $\mathtt{Cm}[x] = -1$ for all $x \in \Sigma_0$ (i.e., initially $\Sigma = \emptyset$). As new letters in Σ_0 are encountered, they are added to Σ and the corresponding entry $\mathtt{Cm}[x]$ updated.

In summary, our Huffman tree is represented by three arrays $\mathtt{Lc}, \mathtt{Wt}, \mathtt{Cm}$. For example, the Huffman tree in Figure 3 is[6] illustrated by the arrays in Table 2: Two of these arrays, $\mathtt{Lc}$ and $\mathtt{Wt}$, are explicitly shown. But the array $\mathtt{Cm}[x \in \Sigma_0]$ is easily inferred from the leaf entries of $\mathtt{Lc}$. E.g., $\mathtt{Cm}[h] = 0$, $\mathtt{Cm}[e] = 1$ and $\mathtt{Cm}[a] = -1$. There is no "Rank" array in this representation because, trivially, $Rank[v] = v$ for all $v \in \{0, \ldots, 2k\}$.



[6] If you compare Table 2 with the Huffman tree in Figure 3, you might be surprised that the left child of 16 is 14 and not 15. Until now, we have not taken the oriented-ness of Huffman trees seriously (since the length of the compressed string did not depend on the ordering of the 2 children of an internal node). According to the displayed binary tree in Figure 3, node 15 is the left child of the root. But the sibling property requires node 14 to be the left child.

Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Lc	h	e	⊔	w	r	d	!	o	0	2	4	6	ℓ	8	10	12	14
Wt	1	1	1	1	1	1	1	2	2	2	2	3	3	4	5	7	12

Table 2: Compact representation of Huffman tree in Figure 3

Here is a simple application of the Sibling representation. Suppose we are given a letter $x \in \Sigma$, and we want to determine the corresponding Huffman code $C(x)$. We need to first go to the leaf u of T corresponding to x. This is of course given by $u = \mathtt{Cm}[x]$. The last bit of $C(x)$ is therefore equal to the "parity" of u. (The parity of a natural number u is equal to 0 if u is even, and equal to 1 otherwise.) Then we replace u by *parent*(u), and thereby determine the next bit of $C(x)$. Iterating this process, we stop when u eventually becomes the root. So the macro to compute the bits of $C(x)$ (in reverse order) is given by

```
C(x):
    u ← Cm[x]
    Output(parity(u))
    While u < 2k
        u ← parent(u)
        Output(parity(u))
```

The parent of u is computed by a simple for-loop:

```
parent(u):
    ℓ ← 2 ⌊u/2⌋
    for p ← u + 1 to 2k
        if (Lc[p] = ℓ), Return(p).
```

The for-loop is sure to terminate when u is non-root. Moreover, the number of times that the p variable is updated (over repeated calls to *parent*(u) by $C(x)$) is at most $2k$ values since the value of p is strictly increasing with each assignment. This ensures the overall complexity of $C(x)$ is $O(k)$. Ideally, we would like to generate $C(x)$ in time $O(|C(x)|)$ (Exercise).

¶19. The Restoration Problem. The key problem of dynamic Huffman tree is how to restore Huffman-ness under a particular kind of perturbation: let T be Huffman and suppose the weight of a leaf u is incremented by 1. So weight of each node along the path from u to the root must be similarly incremented. The result is a weighted code tree T', but it may no longer be Huffman. *Informally, our problem is to restore Huffman-ness in such a tree T'.*

Let us first give some intuition of what has to be done, using our example of `he`$\ell\ell$`o wor`ℓ`d!`. Begin with the Huffman tree after having transmitted the prefix `he`ℓ. Assume that, somehow, we managed to construct a Huffman tree for this string as shown in Figure 6(a). The letters h, e and ℓ are stored in nodes $4, 3$ and 1 (respectively). Note that there is a leaf with weight 0, but we ignore this for now. Each letter has frequency (=weight) of 1. The next transmitted letter is ℓ, and if we simply increase the frequency of node 1 (which

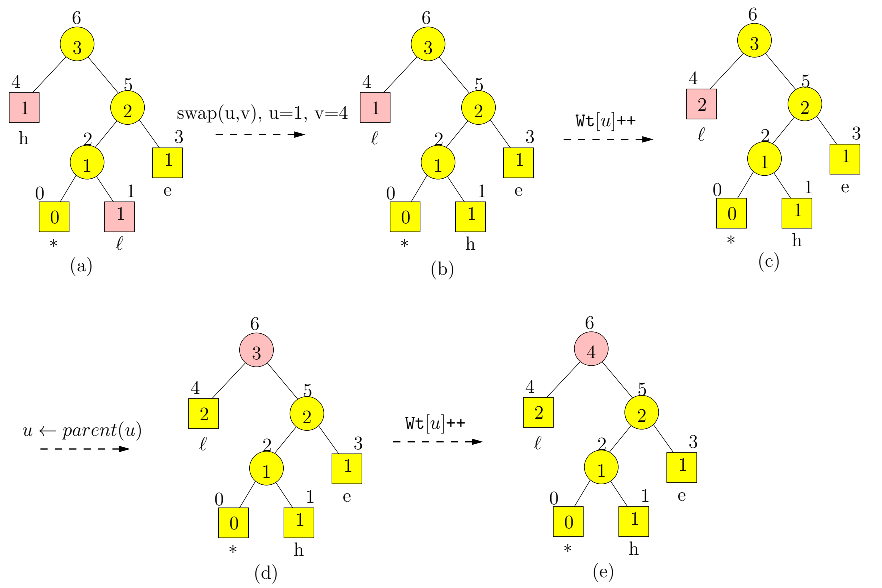


Figure 6: Restoring Huffmanness after incrementing the frequency of letter ℓ

represents ℓ) to $2 = 1 + 1$, we would violate the ranking property (S1) of ¶17. This is because the weight of a node of rank 1 would now be greater than the weights of nodes with greater rank (3 and 4). The key idea is to first *swap node* 1 *with node* 4. This is shown in Figure 6(b). Now, the letter ℓ is represented by node 4, and incrementing its weight by 1 is no longer a problem. The result is seen in Figure 6(c). We must next increment the weight of the parent of node 4, namely node 6. So the focus moves to node 6, as indicated by Figure 6(d). We can simply increment the weight of node 6 because it is the root. But if it is not the root, we may have to do a swap first. The result is Figure 6(e). The process stops since we have reached the root of the tree.

Consider the following algorithm for restoring Huffman-ness in T. For each node v in T, let $R(v)$ denote its rank in the original tree T. But our usual convention is that v is identified with its rank, i.e., $R(v) = v$. Let u be the current node. Initially, u is the leaf whose weight was incremented. We use the following iterative process:

Restore (u)
  ▷ *u is a node whose weight is to be incremented*
  While (u is not the root) do
1.   ▷ *Find the node v with the largest rank $R(v)$ subject*
     ▷ *to the constraint $\mathtt{Wt}[v] = \mathtt{Wt}[u]$. Specifically:*
     $v \leftarrow u$
     While $(\mathtt{Wt}[v+1] = \mathtt{Wt}[u])$
       v++
2.   If $(v \neq u)$
3.   *Swap*(u, v). ◁ *This swaps the subtrees rooted at u and v.*
4. $\mathtt{Wt}[u]$++. ◁ *Increment the weight of u*
5. $u \leftarrow parent(u)$. ◁ *Reset u*
6. $\mathtt{Wt}[u]$++. ◁ *Now, u is the root*

We need to explain one detail in the Restore routine. The swap operation in Line 3 needs to be explained: conceptually, swapping u and v means the subtree rooted at u and the subtree rooted at v exchange places. This can be confusing since our encoding identifies the nodes u and v with their rank. So for the moment, imagine that u is a node in a tree where nodes have parent, left child and right child pointers, etc. Suppose u' and v' were the parents (respectively) of u and v before the swap. Then after the swap, v' (resp., u') becomes the parent of u (resp., v). Coming back to our representation using the `Lc` array, we only have to exchange the values in the array entries $\texttt{Lc}[u]$ and $\texttt{Lc}[v]$. But note that this swap may involve leaves, in which case we have to update the character map `Cm`:

SWAP(u, v)
  $(((tmp \leftarrow \texttt{Lc}[u]) \leftarrow \texttt{Lc}[v]) \leftarrow tmp)$
  If $(\texttt{Lc}[u] \in \Sigma_0)$ then $\texttt{Cm}[\texttt{Lc}[u]] \leftarrow u$
  If $(\texttt{Lc}[v] \in \Sigma_0)$ then $\texttt{Cm}[\texttt{Lc}[v]] \leftarrow v$

Note that we write $((a \leftarrow b) \leftarrow c)$ to mean we first assign b to a and then assign c to b. We do not have to exchange $\texttt{Wt}[u]$ and $\texttt{Wt}[v]$ since these have the same values! A swap is done only if $v > u$ (Line 2). Thus the rank of the current node u is strictly increased by such swaps. After swapping u and v, their siblings will change (recall that rank $2j$ and rank $2j+1$ nodes are siblings).

The reader may verify that the informal example of Figure 6 is really an operation of the Restore routine.

But let us walk through an example of the operations of Restore, this time seeing its transformation on the $\texttt{Lc}, \texttt{Wt}$ arrays. Suppose we have just completely processed our famous string "`he`$\ell\ell$`o world!`", and assume that the resulting Huffman tree T is given by Figure 3. Let the next character to be transmitted be ⊔ (space character), and set u to the node corresponding to ⊔. So $\texttt{Wt}[u]$ is to be incremented, and we call Restore(u). We use the representation of T by the arrays $\texttt{Lc}, \texttt{Wt}$ above: in this case u is the node (whose rank is) 2 (or v_2, for clarity). It has weight $\texttt{Wt}[v_2] = 1$ and so we must find the largest ranked node with weight 1, namely node v_6. Swapping v_2 with v_6, and then incrementing the weight of v_6, we get:

Rank	0	1	<u>2</u>	3	4	5	<u>6</u>	7	8	9	10	11	12	13	14	15	16
Lc	h	e	!	w	r	d	⊔	o	0	2	4	6	ℓ	8	10	12	15
Wt	1	1	1	1	1	1	1+1	2	2	2	2	3	3	4	5	7	12
After first swap			v				u										

Next, u is set to the parent of node of rank 6, namely v_{11}. This has weight 3, and so we must swap it with the element v_{12} which is the highest ranked node with weight 3. After swapping v_{11} and v_{12}, we increment the new v_{12}. The following table illustrates the remaining changes:

Rank	0	1	2	3	4	5	6	7	8	9	10	$\underline{11}$	$\underline{12}$	13	14	$\underline{15}$	$\underline{16}$
Lc	h	e	!	w	r	d	$\sqcup$	o	0	2	4	ℓ	6	8	10	12	15
Wt	1	1	1	1	1	1	1+1	2	2	2	2	3	3+1	4	5	7	12
After second swap												v	u				
Lc	h	e	!	w	r	d	$\sqcup$	o	0	2	4	ℓ	6	8	10	12	15
Wt	1	1	1	1	1	1	1+1	2	2	2	2	3	3+1	4	5	7+1	12
No third swap																$u = v$	
Lc	h	e	!	w	r	d	$\sqcup$	o	0	2	4	ℓ	6	8	10	12	15
Wt	1	1	1	1	1	1	1+1	2	2	2	2	3	3+1	4	5	7+1	12+1
No final swap																	$u = v$

¶20. How to add a new letter: the 0-Node. Our dynamic Huffman code tree T must be capable of expanding its alphabet. E.g., if the current alphabet is $\Sigma = \{\mathtt{h}, \mathtt{e}\}$ and we next encounter the letter `l`, we want to expand the alphabet to $\Sigma = \{\mathtt{h}, \mathtt{e}, \mathtt{l}\}$. For this purpose, we introduce in T a special leaf with weight 0. Call this the 0**-node**. This node does not represent any letters of the alphabet, but in another sense, it represents all the yet unseen letters. We might say that the 0-node represents the character '$*$'. Upon seeing a new letter like `l`, we take three steps to update T:

1. First, we "expand" the 0-node by giving it two children. Its left child is the new 0-node, and its right child u is a new leaf representing the letter `l`.
2. Next, we must give ranks to all the nodes: the new 0-node has rank 0, the new leaf u has rank 1, and all the previous nodes have their ranks incremented by 2. In particular, the original 0-node will have rank 2.
3. Finally, we must update the weights. The weight of the new 0-node is 0, and the weight of u is 1. We must now increase the weight of all the nodes along the path from the old 0-node to the root: this is done by calling Restore on the old 0-node.

Observe that our algorithm for expanding the 0-node maintains the following useful invariant: *the* 0*-node has rank* 0 *and its parent has rank* 2. The operations of the restore function using this 0-node convention is illustrated in Figure 7. Here, we begin with an initial Huffman tree containing just the 0-node, and show successive Huffman trees on inserting the first five letters of our hello example.

Note that the transition from `hel` to `hell` is already described in detail in Figure 6.

The existence of the 0-node actually causes a wrinkle in the Restore routine. We leave the fix to an Exercise.

¶21. Interface between Huffman Code and Canonical Encoding. Let Σ denote the set of characters in the current Huffman code. We view Σ as a subset of a fixed universal set U where $U \subseteq \{0,1\}^N$. Call U the **canonical encoding**. In reality, U might be the set of ASCII characters with $N = 8$. A more complicated example is where U is some unicode set. We assume the transmitter and receiver both know this global parameter N and the set U. In the encoding process, we assume that each character of the string comes from U. Upon seeing a letter x, we must decide whether $x \in \Sigma$ (i.e., in our current Huffman tree), and if so, what

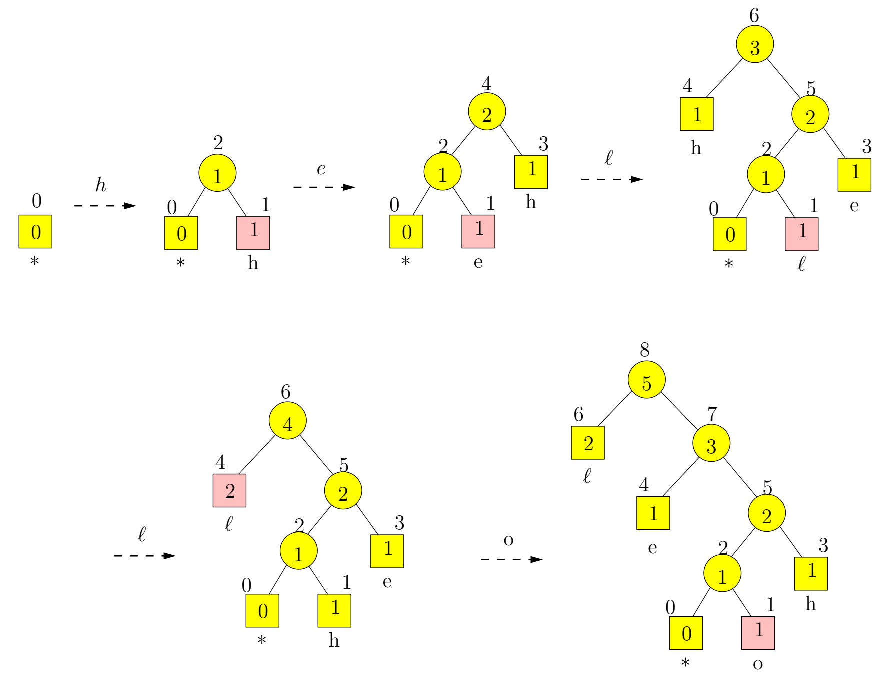


Figure 7: Evolving Huffman tree on inserting the string `hello`

is its current Huffman code. If $|U|$ is not too large (e.g., $|U| = 2^8$), we can provide an array $C[1..2^N]$ such that $C[x]$ maps to a leaf of the Huffman tree. To be specific, suppose $|\Sigma| = k$ and the current Huffman tree T is represented by the arrays $\texttt{Lc}[0..2k], \texttt{Wt}[0..2k]$. If $x \in \{0,1\}^N$, let $C[x] = i$ if node i (of rank i) is the leaf of T representing the letter x. Initially, let $C[x] = -1$ for all x. Hence, the array C is a representation of the alphabet Σ.

For instance, if $C[x]$ is the 0-node, this means x is not in Σ. If $|U|$ is large, we can use hashing techniques.

Even though we know the leaf, it requires some work to obtain the corresponding Huffman code. [This is the encoding problem – but the Huffman code tree is specially designed for the inverse problem, i.e., decoding problem.] One way to solve this encoding problem is assume that our Huffman tree has parent pointer. In terms of our $\texttt{Lc}, \texttt{Wt}$ array representation, we now add another array $P[0..2k]$ for parent pointers.

Here now is the dynamic Huffman coding method for transmitting a string s:

Dynamic Huffman Transmission Algorithm:
Input: A string s of indefinite length.
Output: The dynamically encoded sequence representing s.
▷ *Initialization*
Initialize T to contain just the 0-node.
▷ *Main Loop*
While s is non-empty
1. Remove the next character x from the front of string s.
2. Let $u = C[x]$ be the leaf of T that corresponds to x.
3. Using u, transmit the code word for x.
4. If u is the 0-node ◁ *x is a new character*
5. Expand the 0-node to have two children, both with weight 0;
6. Let u be the right sibling, representing the character x and the left sibling represent the new 0-node.
7. Call Restore(u).
Signal termination, using some convention.

Decoding is also relatively straightforward. We are processing a continuous binary sequence, but we know where the implicit "breaks" are in this continuous sequence. Call the binary sequence between these breaks a **word**. We know how to recognize these words by maintaining the same dynamic Huffman code tree T as the transmission algorithm. For each received word, we know whether it is (a) a code word for some character, (b) signal to add a new letter to the alphabet Σ, or (c) the canonical representation of a letter. Thus the receiver can faithfully reproduce the original string s.

Another practical issue is that whenever we insert a new node, the rank of each existing node implicitly increases by 2. A literal implementation requires updating the entire array for `Lc` and `Wt`. There is a simple solution to this. Let us store the array in reverse order. All invocations of `Lc`$[i]$ is really an invocation of `Lc`$[2k-i]$. Similarly for `Wt`$[i]$. We leave it to the student to work out this detail.

REMARKS: It can be shown that the FGK Algorithm transmits at most $2H_2(s)+4|s|$ bits. The Lambda Algorithm of Vitter ensures that the transmitted string length is $\le H_2(s)+|s|-1$ where $H_2(s)$ is the number of bits transmitted by the 2-pass algorithm for s, independent of alphabet size. In Chapter VI, we will show another approach to dynamic compression of strings based on the move-to-front heuristic and splay trees [1].

¶22. Beyond Huffman Coding: Lempel-Ziv Coding. Although Huffman codding is optimal, we can go beyond its character-by-character encoding assumption. In other words, we can look for blocks of characters that occur with high frequency. For example, in English, certain digraphs like `in, at, th, ng`, etc, occur with high frequency. Certain trigraphs like `the, ing, ion, and`, etc, are also very common. But more generally, there is no need to restrict attention to blocks of any fixed size, but to let the input string determine this. This is the basis of another highly successful encoding scheme due to J.Ziv and A.Lempel [4].

¶23. Notes on the ASCII Character Set. ASCII stands for *The American Standard Code for Information Interchange*, and refers to the 7-bit encoding of 128 characters. See Figure 8(a) for this character set. 95 of these are **printable characters** such as `0, 1,..., 9, a, b, c,..., x, y, z`, and `A, B, C,..., X, Y, Z`, including the space character which we denote by '␣'. The remaining 33 are non-printing or **control characters** such as backspace (BS), carriage-return (CR), bell (BEL),

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

(a) ASCII set

	8	9	A	B	C	D	E	F
0	€			°	À	Ð	à	ð
1		'	¡	±	Á	Ñ	á	ñ
2	‚	'	¢	²	Â	Ò	â	ò
3	ƒ	"	£	³	Ã	Ó	ã	ó
4	„	"	¤	´	Ä	Ô	ä	ô
5	…	•	¥	µ	Å	Õ	å	õ
6	†	–	¦	¶	Æ	Ö	æ	ö
7	‡	—	§	·	Ç	×	ç	÷
8	ˆ	˜	¨	¸	È	Ø	è	ø
9	‰	™	©	¹	É	Ù	é	ù
A	Š	š	ª	º	Ê	Ú	ê	ú
B	‹	›	«	»	Ë	Û	ë	û
C	Œ	œ	¬	¼	Ì	Ü	ì	ü
D				½	Í	Ý	í	ý
E	Ž	ž	®	¾	Î	Þ	î	þ
F		Ÿ	¯	¿	Ï	ß	ï	ÿ

(b) ANSI extension

Figure 8: Extended ASCII character set from `http://ascii-table.com/`

etc. They include many obsolete ones associated with technology such as teletype from a bygone era. There is a natural sorting order associated with these characters, so we may speak of the first, second, and last characters. For instance, the table in Figure 8(a) shows that the first and last characters of these first 128 characters are called NUL and DEL, respectively. Indeed, the first 32 characters (i.e., the first two columns in Figure 8(a)) are control characters, as is the last DEL.

A tele... wha??

The basic ASCII character set has been extended into a set of 256 characters; Figure 8(b) shows this so-called ANSI extension (ANSI stands for *American National Standards Institute*). For example, two special symbols we use throughout in this book comes from the extension: the section symbol '§' and paragraph symbol '¶'. The 256 characters is naturally associated with an 8-bit binary string called its **ASCII code**. We shall write $ASCII(\mathtt{x})$ to denote the ASCII code of a character `x`. The original 128 characters in the ASCII character set, naturally, occupy the first 128 positions; thus a character `x` belongs to this original set iff the most significant bit in $ASCII(\mathtt{x})$ is 0. Of course, the 8-bits can be broken up into two groups of 4-bits which are then interpreted as hexadecimal digits. This much more compact notation is preferred. The 16 hexadecimal digits are conventionally written as $0, 1, 2, \ldots, 9, A, B, C, \ldots, F$. Because of the overlap between standard decimal digits and hexadecimal digits, a sequence like '10' would be ambiguous. To indicate that the hexadecimal digits are meant, we typically prefix the digits by '$0x$'. Thus, the ASCII code for the character `A` is $(01000001)_2$ in binary or $0x41$ in hexadecimal notation. Here are some ASCII codes:

You know the joke: there are 10 kinds of people in the world — those who count in binary and those who count in decimal.

$$\begin{aligned} ASCII(\mathtt{A}) &= 0x41, & ASCII(\mathtt{Z}) &= 0x5A, \\ ASCII(\mathtt{a}) &= 0x61, & ASCII(\mathtt{z}) &= 0x71, \\ ASCII(\mathtt{0}) &= 0x30, & ASCII(\mathtt{9}) &= 0x39, \\ ASCII(\sqcup) &= 0x20, & ASCII(*) &= 0x2A. \end{aligned}$$

Note that $\sqcup$ (space) is considered a printing character. These are all part of the original ASCII set since the first hexadecimal digit are all in the range 0 to 7. In contrast, the extended characters begin with a digit in the range 8 to F. E.g., $ASCII(\S) = 0xA7$ and $ASCII(\P) = 0xB6$. Actually, not all of the available codes in the extension has been assigned to characters. A recent assignment is $ASCII(\text{Euros}) = 0x80$ for the Euro symbol.

There are many variants of the ASCII encoding, as many countries adapted ASCII to their unique requirements. Today, the ASCII encoding is re-interpreted as the first 128 characters of the UTF-8 encoding. The latter is part of the **Unicode**, a character encoding system of amazing scope and generality. This system will be briefly described next.

The original ASCII set together with the ANSI extension is often called the **extended ASCII set**. However, unless otherwise noted in this book, *we will use "ASCII set" to refer to the extended ASCII set.*.

¶24. Notes on Unicode. The Unicode is an evolving standard for encoding the character sets of most human languages, including dead ones like Egyptian hieroglyphs. Here, we must make a basic distinction between **characters** (or graphemes) and their many **glyphs** (or graphical renderings). The idea is to assign a unique number, called a **code point**, to each character. Typically, we write such a number as U+XXXXXX where the X's are hexadecimal. As usual, leading zeros are insignificant. For instance the first 128 code points in Unicode, U+0000 to U+007F, correspond to the ASCII code. The code points below U+0020 are control characters in ASCII code. But there are many subtle points because human languages and writing are remarkably diverse. Characters are not always atomic objects, but may have internal structure. Thus, should we regard "`é`" as a single Unicode character, or as the character "`e`" with a combining acute "´"? Answer: *both solutions are provided in unicode.* If combined, what kinds of combinations do we allow? Coupled with this, we must meet the needs of computer applications: computers use unprintable or control characters, but should these be characters for Unicode? Answer: *of course, this is already part of ASCII.*

There are other international standards (ISO) and these have some compatibility with Unicode. For instance, the first 256 code points corresponds to ISO 8859-1. There are two methods for encoding in Unicode called Unicode Transformation Format (UTF) and Universal Character Set (UCS). These leads to UTF-n, UCS-n for various values of n. Let us just focus on one of these, UTF-8. This was created by K.Thompson and R.Pike, which is a de facto standard in many applications (e.g., electronic mail). It has a basic 8-bit format with variable length extensions that uses up to 4 bytes (32 bits). It is particularly compact for ASCII characters: only 1 byte suffices for the 127 US-ASCII characters. A major advantage of UTF-8 is that a plain ASCII string is also a valid UTF-8 string (with the same meaning of course). Here is UTF-8 in brief:

1. Any code point below U+0080 is encoded by a single byte. Of course, 080 in hex is just 128 in decimal. Thus, U+00XY where $X < 8$ can be represented by the single byte XY that has a leading 0-bit.
2. Code points between U+0080 to U+07FF uses two bytes. The first byte begins with 110, second byte begins with 10.
3. Code points between U+0800 to U+FFFF uses three bytes. The first byte begins with 1110, remaining two bytes begin with 10.
4. Code points between U+100000 to U+10FFFF uses four bytes. The first byte begins with 11110, remaining three bytes begin with 10.

Observe that each code point is self-limiting, i.e., you can tell when you have reached the end of a code point.

Exercises

Exercise 5.1: In this question, we are asking for three numbers. But you must summarize to show intermediate results of your computations. Assume that the alphabet Σ is a subset of $\{0,1\}^8$ (i.e., ASCII code).
(a) What is the length of the (static) Huffman code of the string "`Hello, world!`"? The quotation marks are not part of the string, but the space and punctuation marks are.
(b) How many bits does it take to transmit the Huffman code for the string of (a)?
(c) How many bits would be transmitted by the Dynamic Huffman code algorithm in sending the string "`Hello, world!`"? Compare this number with (a)+(b). $\diamondsuit$

Exercise 5.2: (Gashlin, 2012) Our description of the RESTORE(u) routine is correct only if every leaf has positive weight. In the presence of the 0-node, the routine is incorrect.
(a) Describe the precise situation where the routine fails.
(b) Provide the correct solution (be sure to justify it). ♢

Exercise 5.3: What binary string would you transmit in order to send the string "`now is the time`", under the dynamic Huffman algorithm? Show your working. Note: you would have to transmit ascii codes for the letters `n`, `o`, `w`, etc. Just write $ASCII$(`n`), $ASCII$(`o`), $ASCII$(`w`), etc. ♢

Exercise 5.4: Natural languages are highly redundant (for good reasons). Here is one way to test this.
(a) Please transmit the following string using dynamic Huffman coding, and state the bit length of your transmission.

```
Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer
in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is
taht the frist and lsat ltteer be at the rghit pclae.  The rset can be
a total mses and you can sitll raed it wouthit porbelm.  Tihs is
bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the
wrod as a wlohe.
```

(b) Now repeat part (a) but using similar string in which the words are now properly spelled. Should we expect a drop in the number of transmitted bits? Note that this experiment could not be done using standard Huffman coding since the frequency function in (a) and (b) are identical. ♢

Exercise 5.5:
(a) Please reconstruct the Huffman code tree T from the following representation:

$$r(T) = 0000, 1111, 0011, 011d, mrit, yo$$

CONVENTIONS: the commas in $r(T)$ are just decorative, and meant to help you parse the string. Other than 0/1 symbols, the letters d, m, i, etc, stands for 8-bit ASCII codes. The leftmost leaf in the tree is the 0-node, and its label (namely '$*$') is implicit. NOTE that the '$*$' letter actually has ASCII code $0x2A$ in hex, so it is not literally the smallest letter. The smallest letter in the ASCII code is $0x00$ and is usually called the NUL character. The remaining leaves are labeled by 8-bit ASCII codes for d, m, r, i, t, y, o, in left-to-right order.
(b) Here is a string encoded using this Huffman code:

$$0001, 1110, 1001, 1001, 0111, 1011, 10$$

Decode the string.
(c) Assume that the leaves of the Huffman tree in (a) has the following frequencies (or weights):

$$f(*) = 0, \qquad f(d) = f(m) = f(i) = f(t) = d(y) = 1, \qquad f(r) = f(o) = 2.$$

Assign a rank (i.e., numbers from $0, 1, \ldots, 14$) to the nodes of the tree in (a) so that the sibling property is obeyed. Redraw this tree with the ranking listed next to each node. Also, write the arrays `Lc`[0..14] and `Wt`[0..14] which encodes this ranking of the Huffman tree. Recall that these arrays encode the left-child relation and weights (frequencies),

respectively.
(d) Suppose that we now insert a new letter ⊔ (blank space) into the weighted Huffman code tree of (c). Draw the new Huffman tree with updated ranking. Also, show the updated arrays `Lc`[0..16] and `Wt`[0..16].
(e) Give the Huffman code for the string "`dirty room`" (this string has is a blank character ⊔, but the quotes are not part of the string). What is the relation between this string and the one in (d)? ♢

Exercise 5.6: Give the dynamic Huffman coding for the following anagrams: (1) `the morse code`
(2) `here come dots` ♢

Exercise 5.7: Assume the Sibling Representation of the Huffman code $C : \Sigma \to \{0,1\}^*$. Give the routine to compute the code word $C(x) \in \{0,1\}^*$ of any given $x \in \Sigma$. ♢

Exercise 5.8: Give a careful and efficient implementation of the dynamic Huffman code. Assume the compact representation of Huffman tree using the arrays `Wt` and `Lc` described in the text. ♢

Exercise 5.9: Consider 3-ary Huffman tree code. State and prove the Sibling property for this code. ♢

Exercise 5.10: A previous Exercise asks you to construct the standard Huffman code of Lincoln's speech at Gettysburg.
(a) Construct the optimal Huffman code tree for this speech. Please give the length of Lincoln's coded speech, and also the size of the code tree.
(b) Please give the length of the dynamic Huffman code for this speech. How does it compare to part (a)? Also, compare the code tree at the end of the dynamic coding process with the one in part (a). ♢

Exercise 5.11: In the text, we have represented the Huffman code tree ways: as an binary code tree and as the arrays `Wt`, `Lc`, `Cm`. The former gives $O(|C(x)|)$ complexity for finding the code of $x \in \Sigma$, but the latter is only $O(|\Sigma|)$. Show how to improve the latter complexity in the setting of dynamic Huffman coding. ♢

Exercise 5.12: The correctness of the dynamic Huffman code depends on the fact that the weight at the leaves are integral and the change is $+1$.
(a) Suppose the leave weights can be any positive real number, and the change in weight is also by an arbitrary positive number. Modify the algorithm.
(b) What if the weight change can be negative? ♢

End Exercises

§6. Minimum Spanning Tree

In the **minimum spanning forest problem** we are given a costed bigraph

$$G = (V, E; C)$$

where $C : E \to \mathbb{R}$. An acyclic set $T \subseteq E$ of maximum cardinality is called a **spanning forest**; in this case, $|T| = |V| - c$ where G has $c \geq 1$ components. The **cost** $C(T)$ of any subset $T \subseteq E$ is given by $C(T) = \sum_{e \in T} C(e)$. An acyclic set is **minimum** if its cost is minimum. It is conventional to make the following simplification:

The input bigraph G is connected.

With this assumption, a spanning forest T is actually a tree, and the problem is known as the **minimum spanning tree (MST) problem**. The simplification is not too severe: if our graph is not connected, we can first compute its connected components (we saw efficient solutions to this basic graph problem in Chapter IV). Then we apply the MST algorithm to each component. Alternatively, it is not hard to modify most MST algorithms so that they apply to non-connected graphs.

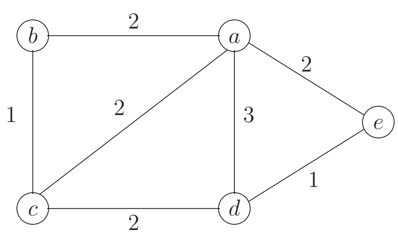


Figure 9: A bigraph with edge costs.

Consider the costed bigraph in Figure 9 with vertices $V = \{a, b, c, d, e\}$. One such MST is $\{a\text{–}b, b\text{–}c, c\text{–}d, d\text{–}e\}$, with cost 6. It is easy to verify that there are a total five MST's for this bigraph, as shown in Figure 10.

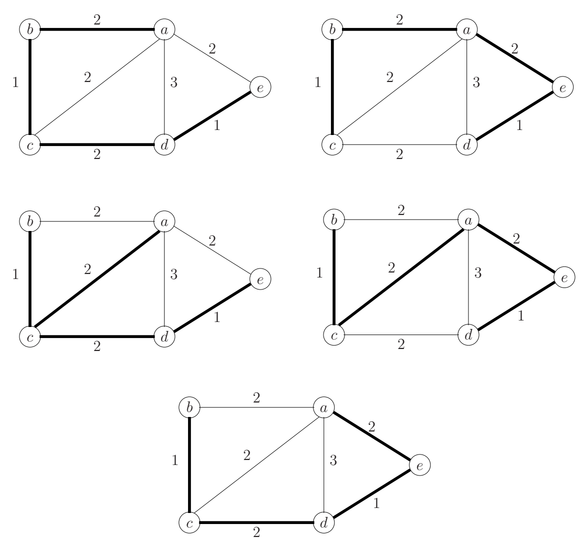


Figure 10: MST's of a bigraph.

¶25. Generic MST Algorithm. There several distinct algorithms for MST. They all fit into the following framework:

```
GENERIC GREEDY MST ALGORITHM
Input: G = (V, E; C) a connected bigraph with edge costs.
Output: S ⊆ E, a MST for G.
    S ← ∅.
    for i = 1 to |V| − 1 do
    1.  Greedy Step: find an e ∈ E \ S that is "good for S".
    2.  S ← S + e.
    Output S as the minimum spanning tree.
```

NOTATION: as illustrated in Line 2, we shall write "$S+e$" for "$S \cup \{e\}$". Likewise, "$S-e$" shall denote the set "$S \setminus \{e\}$".

What does it mean for "e to be good for S"? This will be made specific next.

¶26. Some Greedy MST Criteria. Let us say that e is a **candidate** for S if $S+e$ is acyclic. Consider the graph $G_S = (V, S)$: to understand MST algorithms, we need to look at the connected components of G_S. If U is a connected component of G_S, and $e = (u, v)$ is a candidate such that $u \in U$ or $v \in U$ then we say that e **extends** the component U. Among the candidates for S, only some are "good". The following are 4 notions of what "good" means:

- (Simple) $S + e$ is extendible to some MST.
- (Kruskal) Edge e has the least cost among all the candidates.
- (Boruvka) There is a connected component U of $G' = (V, S)$ such that e has the least cost among all the candidates that extend U. Let us expand this a bit: suppose G' has k components $U_1, \ldots, U_k$. Then for each U_i, there is at least one e_i that extends U_i with least cost. The fact that there are many choices for e can be exploited in parallel algorithms. Note that it is possible that $e_i = e_j$ with $i \neq j$, but we still have at least $\lceil k/2 \rceil$ choices.
- (Prim) This has, in addition to Boruvka's condition, the requirement that the graph $G'' = (V, S+e)$ has only one non-trivial component. [A component is trivial if it has only a single vertex.]

This first criterion is computational ineffective. The remaining three criteria are effective and are named after the inventors of three well-known MST algorithms. There are additional algorithmic techniques that are needed before we finally achieve the best realization of these ideas:

- (Kruskal) Kruskal tells us to sort the edges first, and then consider each edge e in order of increasing cost. How can we quickly tell if $S+e$ is acyclic? If e is $u{-}v$, this amounts to checking if u, v are in the same connected component of the graph $G' = (V, S)$. A simple method is to do this is have a linked list for each connected component of $G' = (V, S)$, with the nodes of the linked list representing vertices of the connected component. Given a vertex u, assume we have a pointer from u to the representative node for u in such a linked list. To decide if two vertices u, v are in the same connected component, we go to the linked lists nodes that represent u and v, and follow the links till the end of their respective linked lists. *The ends of these two linked list are equal iff $S + e$ has a cycle.*

The elaborations of this linked list idea will ultimately lead us to the union-find data structure which is studied in Chapter XIII. An Exercise below will explore some of these ideas.

- (Boruvka) We must maintain for each connected component of $G' = (V, S)$, the least cost edge that extends it. Again we need some form of union-find data structure. A key feature of Boruvka's algorithm is that we can select the good edges in "phases" where each phase calls for a pass through the set of remaining edges. This feature can be exploited in parallel algorithms. We explore these ideas in the Exercise.
- (Prim) Because of its additional restriction to one non-trivial connected component, Prim's algorithm is easier to implement than Boruvka's. We shall do this below. But the ultimate version of Prim's algorithm can only be taken up in Chapter VI (amortization techniques).

Let us call those sets $S \subseteq E$ that may arise during the execution of the generic MST algorithm **simply-good**, **Boruvka-good**, **Kruskal-good** or **Prim-good**, depending on which of the above criteria is used. The correctness of these algorithms amounts to showing that "*X-good implies simply-good*" where X = Kruskal, Boruvka or Prim. Let us now show the correctness of the algorithm of Boruvka. By definition, Prim-good implies Boruvka-good, and so Prim's algorithm is also correct. Indeed, Kruskal-good also implies Boruvka-good, so this also show the correctness of Kruskal's algorithm.

Lemma 9 (Correctness of Boruvka's Algorithm). *Boruvka-good sets are simply-good.*

Proof. We use induction on the size $|S|$ of Boruvka-good sets S. Clearly if $S = \emptyset$, then S is Boruvka-good and this is clearly simply-good. Next suppose $S = S' + e$ where S' is Boruvka-good. We need to prove that S is simply-good. By the Boruvka-goodness of S', there is a component U of the graph $G' = (V, S')$ such that e has the least cost among all edges that extend U. By induction hypothesis, we may assume S' is simply-good. Hence there is a MST T' that contains S'. If $e \in T'$, then we are done (as T' would be a witness to the fact that $S = S' + e$ is simply-good). So assume $e \notin T'$.

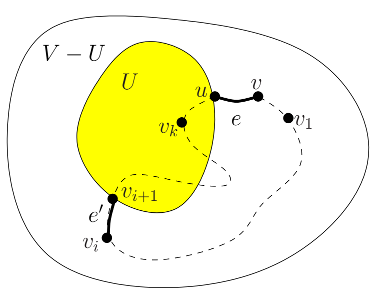


Figure 11: Extending a component U by $e = (u, v)$.

Write $e = u{-}v$ such that $u \in U$ and $v \notin U$. Hence $T' + e$ contains a unique closed path of the form

$$Z := (u{-}v{-}v_1{-}v_2{-}\cdots{-}v_k{-}u).$$

There exists some $i = 0, \ldots, k$ such that $v_i \notin U$ and $v_{i+1} \in U$. Write

$$Z = (u{-}v{-}v_1{-}\cdots{-}v_i{-}v_{i+1}{-}\cdots{-}u)$$

(where $v = v_0$ and $u = v_{k+1}$ in this notation). Let $e' := (v_i{-}v_{i+1})$. Note that $T := T' + e - e'$ is acyclic and is a spanning tree. Moreover, $C(e) \le C(e')$, by our choice of e. Hence $C(T) \le C(T')$. Since $C(T')$ is minimum, so is $C(T)$. This shows that S is simply-good, as S is contained in T.

Q.E.D.

¶27. Good sets of vertices. Let us extend the notion of "goodness" to sets of vertices. For any set $S \subseteq E$ of edges, let $V(S)$ denote the set of vertices that are incident on some edge of S. We say a set $U \subseteq V$ is X**-good** if there exists an X-good set $S \subseteq E$ such that $U = V(S)$. Here, X is equal to 'simply', 'Prim', 'Kruskal' or 'Boruvka'. We also declare any singleton set with only one vertex to be X-good.

¶28. Hand Simulation of MST Algorithms. Students are expected to understand those aspects of Kruskal's and Prim's algorithms that are independent of their ultimate realizations via efficient data structures. That is, you must do "hand simulations" where you act as the oracle for queries to the data structures. For Kruskal's algorithm, this is easy – we just list the edges by increasing weight order and indicate the acceptance/rejection of successive edges.

For Prim's algorithm, we just maintain an array $d[1..n]$ assuming the vertex set is $V = \{1, \ldots, n\}$. We shall maintain a subset $S \subseteq V$ representing the set of vertices which we know how to connect to the source node 1 in a MST. The set S is "Prim good". Initially, let $S = \emptyset$ and $d[1] = 0$ and $d[v] = \infty$ for $v = 2, \ldots, n$. In general, the entry $d[v]$ $(v \in V \setminus S)$ represents the "cheapest" cost to connect vertex v to the MST on the set S. Our simulation consists in building up a matrix M which is a $n \times n$ matrix, where the 0th row representing the initial array d. Each time the array d is updated, we rewrite it as a new row of a matrix M.

At stage $i \ge 1$, suppose we pick a node $v_i \in V \setminus S$ where $d[v_i] = \min\{d[j] : j \in V \setminus S\}$. We add v_i to S, and update all the values $d[u]$ for each $u \in V \setminus S$ that is adjacent to v_i. The update rule is this:

$$d[u] = \min\{d[u], C[v_i, u]\}.$$

The resulting array is written as row i in our matrix.

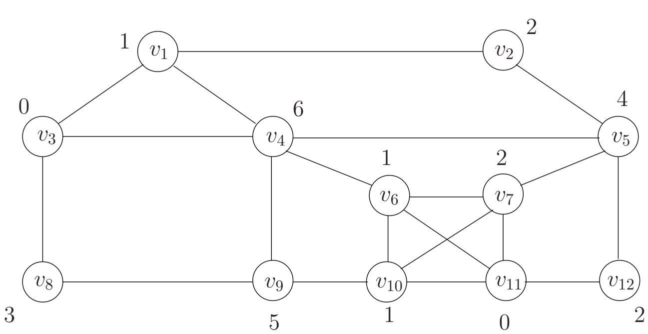


Figure 12: The house graph: The cost of edge $v_i{-}v_j$ is defined as $C(v_i) + C(v_j)$, where $C(v)$ is the value indicated next to v. E.g. $C(v_1{-}v_4) = 1 + 6 = 7$.

Let us illustrate the process on the graph of Figure 12. The vertex set is $V = \{v_1, v_2, \ldots, v_{11}, v_{12}\}$. The cost of an edge is the sum of the costs associated to each vertex. E.g., $C(v_1, v_4) = C(v_1) + C(v_4) = 1 + 6 = 7$. The final matrix is the following:

Stage	1	2	3	4	5	6	7	8	9	10	11	12
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	X	3	1	7	∞	∞	∞	∞	∞	∞	∞	∞
2			X	6				3				
3		X			6							
4								X	8			
5				X		7						
6					X		6					6
7						3	X			3	2	
8						1				1	X	2
9						X						
10									6	X		
11												X
12									X			

Conventions in this matrix: We mark the newly picked node in each stage with an 'X'. Also, any value that is unchanged from the previous row may be left blank. Thus, in stage 2, the node 3 is picked and we update $d[v_4]$ using $d[v_4] = \min\{d[v_4], C[v_3, v_4]\} = \min\{7, 6\} = 6$.

The final cost of the MST is 37. To see this, each X corresponds to a vertex v that was picked, and the last value of $d[v]$ contributes to the cost of the MST. E.g., the X corresponding to vertex 1 has cost 0, the X corresponding to vertex 2 has cost 3, etc. Summing up over all X's, we get 37.

¶29. Boruvka's MST Algorithm. Let $S \subseteq E$ be Boruvka-good, and $G_S = (V, S)$ is the subgraph of G with S as edge set. If C is a connected component of G_S, an edge $e = (u-v)$ where $u \in C$ and $v \notin C$ is called an **outgoing edge** of C. Recall that if e has minimum cost among all outgoing edges of C, then the extension $S+e$ is Boruvka-good. If G_S has k connected components, we pick one such minimum outgoing edge per component. Let e_i be the picked edge from component C_i $(i = 1, \ldots, k)$. There are two issues. First, it may happen that two distinct components $C \neq C'$ share the same minimum outgoing edges, $e_i = e_j$ for $i \neq j$. But no more than two components may share the same minimum outgoing edge. Thus, the number k' of distinct edges edges among $e_1, \ldots, e_k$ at least $k/2$ (and hence at least $\lceil k/2 \rceil$). Second, we want to ensure that $S \cup \{e_1, e_2, \ldots, e_{k'}\}$ is acyclic. To do this, we must break ties carefully in choosing the e_i's. Assume there is a global ordering "$<^*$" on the edge set E such that among the outgoing edges in a component with the same minimum weight, e_i is the minimum under this $<^*$-order. It is now easy to see that there are no cycles. In particular, k' cannot be equal to k (otherwise $S \cup \{e_1, e_2, \ldots, e_k\}$ has a cycle). In summary:

$$\left.\begin{array}{l}\text{the number } k' \text{ of distinct edge in the set } \{e_1, e_2, \ldots, e_k\} \\ \text{is between } k/2 \text{ and } k-1, \text{ and } S \cup \{e_1, \ldots, e_k\} \text{ is acyclic.}\end{array}\right\} \tag{31}$$

Thus, we can simultaneously extend S by adding these k' edges to obtain a Boruvka-good set S'. The key difference between Boruvka's algorithm and, say Prim's or Kruskal's algorithms, is that we can operate in "phases": if S is the Boruvka-good set at the beginning of a phase, then at the end of the phase we extend S by the k' distinct edges, forming a new acyclic set S'. These k' edges could be chosen "in parallel" if we were using parallel computers. But since we only consider on sequential algorithms, here is the conventional form of this algorithm.

```
Boruvka's MST Algorithm
    Input: G = (V, E; C) a connected graph
    Output: MST S ⊆ E of G
    S ← ∅    ◁ Initialize a Boruvka-good set of edges
    While (|S| < n − 1)
        for each connected component C of G_S,    ◁ Do Phase
            e ← argmin {C(v, u) : v ∈ C, u ∉ C}
            S ← S + {e}
```

How many phases can there be? If G_S has k components, and we form S' by adding k' distinct edges to S, then $G_{S'}$ has exactly $k-k'$ connected components. From (31), we conclude that $1 \le k-k' \le k/2$. In other words, the number of connected components it at least halved. Since we started with n components, the number of phases is at most $\lg n$. We next show that each phase can be computed in time $O(m+n)$, and so the overall complexity of Boruvka's algorithm is $O((m+n)\log n)$.

¶30. Implementation of a Phase. We now show how to implement a phase of Boruvka's algorithm in time $O(m+n)$. The key subroutine is a method to compute the connected components of a bigraph. In Chap.IV, ¶23, we have provided such an algorithm based on BFS.

- The main data structure is an array $CC[1..n]$ to keep track of the connected components of G_S. For each $i \in \{1, \ldots, n\} = V$, let $CC[i]$ refer to an arbitrary but fixed vertex j in the connected component of i. This j is called the **representative** of that connected component. In particular, we always have $CC[j] = j$. Initially, $S = \emptyset$ and hence we have $CC[i] = i$ for all i. Assume inductively that the array $CC[1..n]$ is available at the beginning of the phase. If $CC[i] = j$, then we say i belongs to component j.
- Next, we shall compute the minimum outgoing edge for each connected component with the help of an array, $A[1..n]$. Let $A[j]$ store the current minimum outgoing edge from component j. We initialize $A[j] \leftarrow$ `nil`, reflecting the fact that no outgoing edges are initially known. Note that if there are k components, then only k entries of the arrays A are in use. We incrementally update A as follows:

```
For each edge (i−j) ∈ E,
    If CC[i] ≠ CC[j]    ◁ (i−j) is an outgoing edge of component of i
        If the cost of A[CC[i]] is greater than C(i, j)
            A[CC[i]] ← (i−j).
```

  At the end of this for-loop, it is clear that the array A has the desired information.
- We must now extend the set S to S': this amounts to updating the array $CC[1..n]$ for the next phase. Imagine the reduced graph $G^r = (V^r, E^r)$ whose vertices are the set of representatives of connected components in G_S, and whose edges are the edges in $A[1..n]$. Viewed as a bigraph, our goal is to compute the connected components of G^r. For each $i \in V$, we retrieve the edge $(j-k) \leftarrow A[CC[i]]$. We add $CC[i]$ to V^r and $(CC[j]-CC[k])$ to E^r. Then we use a BFS Driver to compute the connected components of G^r. Assume (see Chap. IV, ¶23) that the BFS Driver returns an array CC^r such that for each $j \in V^r$, we have $CC^r[j]$ is a representative vertex in the connected component of j. Now we update CC with the help of CC^r:

For each $i \in V$,
$CC[i] \leftarrow CC^r[CC[i]]$

This concludes the phase.

Remarks: Boruvka (1926) has the first MST algorithm; his algorithm was rediscovered by Sollin (1961). The algorithm attributed to Prim (1957) was discovered earlier by Jarník (1930). These algorithms have been rediscovered many times. See [9] for further references. Both Boruvka and Jarník's work are in Czech. The Prim-Jarník algorithm is very similar in structure to Dijkstra's algorithm which we will encounter in the chapter on minimum cost paths.

Exercises

Exercise 6.1: We consider minimum spanning trees (MST's) in an undirected graph $G = (V, E)$ where each vertex $v \in V$ is given a numerical value $C(v) \geq 0$. The **cost** $C(u, v)$ of an edge $(u{-}v) \in E$ is defined to be $C(u) + C(v)$.
(a) Let G be the graph in Figure 12. Compute an MST of G using Boruvka's algorithm. Please organize your computation so that we can verify intermediate results. Also state the cost of your minimum spanning tree.
(b) Can you design an MST algorithm that takes advantage of the fact that edge costs has the special form $C(u, v) = C(u) + C(v)$? ♢

Exercise 6.2: Redo the previous problem with a different cost function, where $C(u{-}v) = C(u)C(v)$ (the product instead of the sum). Is the result the same? ♢

Exercise 6.3: Suppose G is the complete bipartite graph $G_{m,n}$. That is, the vertices V are partitioned into two subsets V_0 and V_1 where $|V_0| = m$ and $V_1| = n$ and $E = V_0 \times V_1$. Give a simple description of an MST of $G_{m,n}$. Argue that your description is indeed an MST. HINT: transform an arbitrary MST into your description by modifying one edge at a time. ♢

Exercise 6.4: Let G_n be the bigraph whose vertices are $V = \{1, 2, \ldots, n\}$. The edges are defined as follows: for each $i \in V$, if i is prime, then $(1, i) \in E$ with weight i. [Recall that 1 is not considered prime, so 2 is the smallest prime.] For $1 < i < j$, if i divides j then we add (i, j) to E with weight j/i.
(a) Draw the graph G_{19}.
(b) Compute the MST of G_{18} using Prim's algorithm, using node 1 as the source vertex. Please use the organization described in the appendix below.
(c) Are there special properties of the graphs G_n that can be exploited? ♢

Exercise 6.5: Let $G = (V, E; W)$ be a connected bigraph with edge weight function W. Fix a constant M and define the weight function W' where $W'(e) = M - W(e)$ for each $e \in E$. Let $G' = (V, E; W')$. Show that T is a maximum spanning tree of G iff T is a minimum spanning tree of G'. NOTE: Thus we say that the concepts of maximum spanning tree and minimum spanning tree are "cryptomorphic versions" of each other. ♢

Exercise 6.6: Describe the rule for reconstructing the MST from the matrix M using in our hand-simulation of Prim's Algorithm. ♢

Exercise 6.7: Hand Simulation of Kruskal's Algorithm on the graph of Figure 12. This exercise suggests a method for carry out the steps of this algorithm. We consider each edge in their sorted order, maintaining a partition of $V = \{1, \ldots, 12\}$ into disjoint sets. Let $L(i)$ denote the set containing vertex i. Initially, each node is in its own set, i.e., $L(i) = \{i\}$. Whenever an edge $i{-}j$ is added to the MST, we merge the corresponding sets $L(i) \cup L(j)$. E.g., in the first step, we add edge $1{-}3$. Thus the lists $L(1) = \{1\}$ and $L(3) = \{1\}$ are merged, and we get $L(1) = L(3) = \{1, 3\}$. To show the computation of Kruskal's algorithm, for each edge, if the edge is "rejected", we mark it with an "X". Otherwise, we indicate the merged list resulting from the union of $L(i)$ and $L(j)$: Please fill in the last two columns of the table (we have filled in the first 4 rows for you).

Sorting Order	Edge	Weight	Merged List	Cumulative Weight
1	1–3:	1	$\{1, 3\}$	1
2	6–11:	1	$\{6, 11\}$	2
3	10–11:	1	$\{6, 10, 11\}$	3
4	6–10:	2	X	3
5	7–11:	2		
6	11–12:	2		
7	1–2:	3		
8	3–8:	3		
9	6–7:	3		
10	7–10:	3		
11	2–5:	6		
12	3–4:	6		
13	5–7:	6		
14	5–12:	6		
15	9–10:	6		
16	1–4:	7		
17	4–6:	7		
18	8–9:	8		
19	4–5:	10		
20	4–9:	11		

♢

Exercise 6.8: This question considers two concrete ways to implement Kruskal's algorithm. Let $V = \{1, 2, \ldots, n\}$ and $D[1..n]$ be an array of size n that represents a **forest** $G(D)$ with vertex set V and edge set $E = \{(i, D[i]) : i \in V\}$. More precisely, $G(D)$ is an directed graph that has no cycles except for self-loops (i.e., edges of the form (i, i)). A vertex i such that $D[i] = i$ is called a **root**. The set V is thereby partitioned into disjoint subsets $V = V_1 \cup V_2 \cup \cdots \cup V_k$ (for some $k \geq 1$) such that each V_i has a unique root r_i, and from every $j \in V_i$ there is a path from j to r_i. For example, with $n = 7$, $D[1] = D[2] = D[3] = 3$, $D[4] = 4$, $D[5] = D[6] = 5$ and $D[7] = 6$ (see Figure 13). We call V_i a **component** of the graph $G(D)$ (this terminology is justified because V_i is a component in the usual sense if we view $G(D)$ as an *undirected* graph).

(i) Consider two restrictions on our data structure: Say D is **list type** if each component is a linear list. Say D is **star type** if each component is a star (i.e., each vertex in the component points to the root). E.g., in Figure 13, V_2 and V_3 are linear lists, while V_1 and V_2 are stars. Let ROOT(i) denote the root r of the component containing i. Give a pseudo-code for computing ROOT(i), and give its complexity in the 2 cases: (1) D is list type, (2) D is star type.

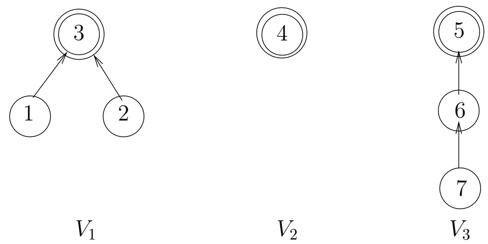


Figure 13: Directed graph $G(D)$ with three components (V_1, V_2, V_3)

(ii) Let COMP$(i) \subseteq V$ denote the component that contains i. Define the operation $MERGE(i, j)$ that transforms D so that COMP(i) and COMP(j) are combined into a new component (but all the other components are unchanged). E.g., the components in Figure 13 are $\{1, 2, 3\}$, $\{4\}$ and $\{5, 6, 7\}$. After $MERGE(1, 4)$, we have two components, $\{1, 2, 3, 4\}$ and $\{5, 6, 7\}$. Give a pseudo-code that implements $MERGE(i, j)$ under the assumption that i, j are roots and D is list type which you must preserve. Your algorithm *must* have complexity $O(1)$. To achieve this complexity, you need to maintain some additional information (perhaps by a simple modification of D).
(iii) Similarly to part (ii), implement $MERGE(i, j)$ when D is star type. Give the complexity of your algorithm.
(iv) Describe how to use ROOT(i) and $MERGE(i, j)$ to implement Kruskal's algorithm for computing the minimum spanning tree (MST) of a weighted connected undirected graph H.
(v) What is the complexity of Kruskal's in part (iv) if (1) D is list type, and if (2) D is star type. Assume H has n vertices and m edges. $\diamondsuit$

Exercise 6.9: Give two alternative proofs that the suggested algorithm for computing minimum base is correct:
(a) By verifying the analogue of the Correctness Lemma.
(b) By replacing the cost $C(e)$ (for each $e \in E$) by the cost $c_0 - C(e)$. Choose c_0 large enough so that $c_0 - C(e) > 0$. $\diamondsuit$

Exercise 6.10: Let G be a bigraph G with distinct weights. Give a direct argument for the (a) and (b).
(a) Prove that the MST of G must contain that the edge of smallest weight.
(b) Prove that the MST of G must contain that the edge of second smallest weight.
(c) Must it contain the edge of third smallest weight? $\diamondsuit$

Exercise 6.11: Show that every MST can be obtained from Kruskal's algorithm by a suitable re-ordering of the edges which have identical weights. Conclude that when the edge weights are unique, then the MST is unique. $\diamondsuit$

Exercise 6.12: Student Joe wants to reduce the minimum base problem for a costed matroid $(S, I; C)$ to the MIS problem for $(S, I; C')$ where C' is a suitable transformation of C. See next section for matroid definitions.
(a) Student Joe considers the modified cost function $C'(e) = 1/C(e)$ for each e. Construct an example to show that the MIS solution for C' need not be the same as the minimum base solution for C.

(b) Next, student Joe considers another variation: he now defines $C'(e) = -C(e)$ for each e. Again, provide a counter example. ♢

Exercise 6.13: Extend the algorithm to finding MIS in contracted matroids. ♢

Exercise 6.14: If $S \subseteq E$ is Prim-good, then clearly $G' = (V(S), S)$ is clearly a tree. Prove that S is actually an MST of the restricted graph $G|V(S)$. ♢

Exercise 6.15:
(a) Enumerate the X-good sets of vertices in Figure 9. Here, X is 'simply', 'Kruskal', 'Boruvka' or 'Prim'.
(b) Characterize the good singletons (relative to any of the three notions of goodness). ♢

Exercise 6.16: This question will develop Boruvka's approach to MST: for each vertex v, pick the edge $(v{-}u)$ that has the least cost among all the nodes u that are adjacent to v. Let P be the set of edges so picked.
(a) Show that $n/2 \le P \le n-1$. Give general examples to show that these two extreme bounds are achieved for each n.
(b) Show that if the costs are unique, P cannot contain a cycle. What kinds of cycles can form if weights are not unique?
(c) Assume edges in P are picked with the tie breaking rule: among the edges $v{-}u_i$ ($i = 1, 2, \ldots$) adjacent to v that have minimum cost, pick the u_i that is the smallest numbered vertex (assume vertices are numbered from 1 to n). Prove that P is acyclic and has the following property: if adding an edge e to P creates a cycle Z in $P + e$, then e has the maximum cost among the edges in Z.
(d) For any costed bigraph $G = (V, E; C)$, and $P \subseteq E$, define a new costed bigraph denoted G/P as follows. First, two vertices of V are said to be equivalent modulo P if they are connected by a sequence of edges in P. For $v \in V$, let $[v]$ denote the equivalence class of v. The vertex set of G/P is $\{[v] : v \in V\}$. The edge set of G/P comprises those $([u]{-}[v])$ such that there exists an edge $(u'{-}v') \in E$ where $u' \in [u]$ and $v' \in [v]$. The cost of $([u]{-}[v])$ is defined as $\min\{C(u', v') : u' \in [u], v' \in [v], (u'{-}v') \in E\}$. Note that G/P has at most $n/2$ vertices. Moreover, we can pick another set P' of edges in G/P using the same rules as before. This gives us another graph $(G/P)/P'$ with at most $n/4$ vertices. We can continue this until V has 1 vertex. Please convert this informal description into an algorithm to compute the cost of the MST. (You need not show how to compute the MST.)
(e) Determine the complexity of your algorithm. You will need to specify suitable data structures for carrying out the operations of the algorithm. (Please use data structures that you know up to this point.) ♢

Exercise 6.17: (Tarjan) Consider the following **generic accept/reject algorithm** for MST. This consists of steps that either **accept** or **reject** edges. In our generic MST algorithm, we only explicitly accept edges. However, we may be implicitly rejecting edges as well, as in the case of Kruskal's algorithm. Let S, R be the sets of accepted and rejected edges (so far). We say that (S, R) is **simply-good** if there is an MST that contains S but not containing any edge of R. Note that this extends our original definition of "simply good". Prove that the following extensions of S and R will maintain minimal goodness:
(a) Let $U \subseteq V$ be any subset of vertices. The set of edges of the form (u, v) where $u \in U$

and $v \notin U$ is called a U-**cut**. If e is the minimum cost edge of a U-cut and there are no accepted edges in the U-cut, then we may extend S by e.
(b) If e is the maximum cost edge in a cycle C and there are no rejected edges in C then we may extend R by e. $\diamondsuit$

Exercise 6.18: With respect to the generic accept/reject version of MST:
(a) Give a counter example to the following rejection rule: let e and e' be two edges in a U-cut. If $C(e) \ge C(e')$ then we may reject e'.
(b) Can the rule in part (a) be fixed by some additional properties that we can maintain?
(c) Can you make the criterion for rejection in the previous exercise (part (b)) computationally effective? Try to invent the "inverses" of Prim's and Boruvka's algorithm in which we solely reject edges.
(d) Is it always a bad idea to *only* reject edges? Suppose that we alternatively accept and reject edges. Is there some situation where this can be a win? $\diamondsuit$

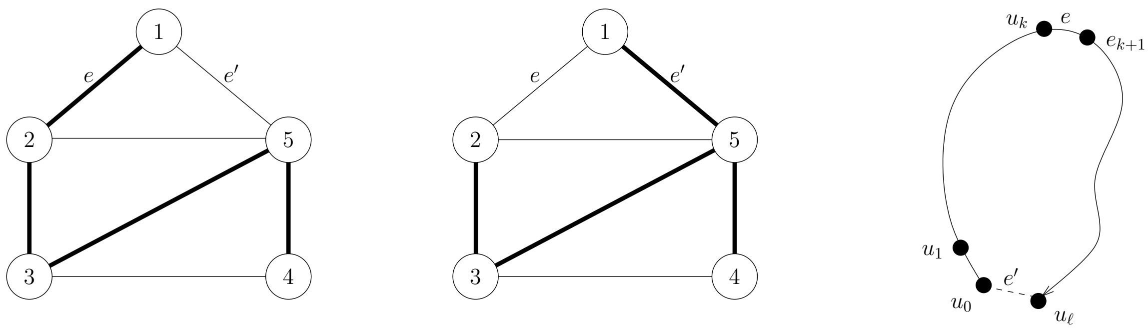
Exercise 6.19: Consider the following recursive "MST algorithm" on input $G = (V, E; C)$:
(I) Subdivide $V = V_1 \uplus V_2$.
(II) Recursive find a "MST" T_i of $G|V_i$ $(i = 1, 2)$.
(III) Find e in the V_1-cut of minimum cost. Return $T_1 + e + T_2$.
Give a small counter example to this algorithm. Can you fix this algorithm? $\diamondsuit$

Exercise 6.20: Is there an analogue of Prim and Boruvka's algorithm for the MIS problem for matroids? $\diamondsuit$

Exercise 6.21: Let $G = (V, E; C)$ be the complete graph in which each vertex $v \in V$ is a point in the Euclidean plane and $C(u, v)$ is just the Euclidean distance between the points u and v. Give efficient methods to compute the MST for G. $\diamondsuit$

Exercise 6.22: Fix a connected undirected graph $G = (V, E)$. Let $T \subseteq E$ be any spanning tree of G. A pair (e, e') of edges is called a **swappable pair for** T if
(i) $e \in T$ and $e' \in E \setminus T$ (Notation: for sets A, B, their difference is denoted $A \setminus B = \{a \in A : a \notin B\}$)
(ii) The set $(T \setminus \{e\}) \cup \{e'\}$ is a spanning tree.
Let $T(e, e')$ denote the spanning tree $(T \setminus \{e\}) \cup \{e'\}$ obtained from T **by swapping** e **and** e' (see illustration in Figure 14(a), (b)).

(a) Suppose (e, e') is a swappable pair for T and $e' = (u, v)$. Prove that e lies on the unique path, denoted by $P(u, v)$, of T from u to v. In Figure 14(a), $e' = (1–5) = (5–1)$. So the path is either $P(1, 5) = (1–2–3–5)$ or $P(5, 1) = (5–3–2–1)$.
(b) Let $n = |V|$. Relative to T, we define a $n \times n$ matrix $First$ indexed by pairs of vertices u, v, where $First[u, v] = w$ means that the first edge in the unique path $P(u, v)$ is (u, w). (In the special case of $u = v$, let $First[u, u] = u$.) In Figure 14(a), $First[1, 5] = 2$ and $First[5, 1] = 3$. Show the matrix $First$ for the tree T in Figure 14(a). Similarly, give the matrix $First$ for the tree $T(e, e')$ in Figure 14(b).
(c) Describe an $O(n^2)$ algorithm called $Update(First, e, e')$ which updates the matrix $First$ after we transform T to $T(e, e')$. HINT: For which pair of vertices (x, y) does the value of $First[x, y]$ have to change? Suppose $e' = (u', v')$ and $P(u', v') = (u_0, u_1, \ldots, u_\ell)$ is as illustrated in Figure 14(c). Then $u' = u_0, v' = u_\ell$, and also $e = (u_k, u_{k+1})$ for some $0 \le k < \ell$. Then, originally $First[u_0, u_\ell] = u_1$ but after the swap, $First[u_0, u_\ell] = u_\ell$. What else must change?



(a) T (b) $T(e, e')$ (c) Path $P(u_0, u_\ell)$.

Figure 14: (a) A swappable pair (e, e') for spanning tree T. (b) The new spanning tree $T(e, e')$ [NOTE: tree edges are indicated by thick lines]

(d) Analyze your algorithm to show that that it is $O(n^2)$. Be sure that your description in (c) is clear enough to support this analysis. ♢

End Exercises

§7. Matroids

An abstract structure that supports greedy algorithms is matroids. Indeed, we will see that Kruskal's algorithm for MST is an instance of a general greedy method to solve a matroid problem. We first illustrate the idea of matroids.

¶31. Graphic matroids. Let $G = (V, S)$ be a bigraph. A subset $A \subseteq S$ is **acyclic** if it does not contain any cycle. Let I be the set of all acyclic subsets of S. The empty set is a acyclic and hence belongs to I. We note two properties of I:

Hereditary property: If $A \subseteq B$ and $B \in I$ then $A \in I$.

Exchange property: If $A, B \in I$ and $|A| < |B|$ then there is an edge $e \in B - A$ such that $A \cup \{e\} \in I$.

The hereditary property is obvious. To prove the exchange property, note that the subgraph $G_A := (V, A)$ has $|V| - |A|$ (connected) components; similarly the subgraph $G_B := (V, B)$ has $|V| - |B|$ components. If every component $U \subseteq V$ of G_B is contained in some component of U' of G_A, then $|V| - |B| < |V| - |A|$ implies that some component of G_A contains no vertices, contradiction. Hence assume $U \subseteq V$ is a component of G_B that is not contained in any component of G_A. Let $T := B \cap \binom{U}{2}$. Thus (U, T) is a tree and there must exist an edge $e = (u{-}v) \in T$ such that u and v belongs to different components of G_A. This e will serve for the exchange property.

For example, in Figure 9 the sets $A = \{a{-}b, a{-}c, a{-}d\}$ and $B = \{b{-}c, c{-}a, a{-}d, d{-}e\}$ are acyclic. Then the exchange property between A and B is witnessed by the edge $d{-}e \in B \setminus A$, since adding $d{-}e$ to A will result in an acyclic set.

¶32. Matroids. The above system (S, I) is called the **graphic matroid** corresponding to graph $G = (V, S)$. In general, a **matroid** is a hypergraph

$$M = (S, I)$$

with special properties: S and $I \subseteq 2^S$ are both non-empty sets such that I has both the hereditary and exchange properties. The set S is called the **ground set**. Elements of I are called **independent sets**; other subsets of S are called **dependent sets**. Note that the empty set $\emptyset$ is always a member of I.

Another example of matroids arise with numerical matrices: for any matrix M, let S be its set of columns, and I be the family of linearly independent subsets of columns. Call this the **matrix matroid** of M. The terminology of independence comes from this setting. This was the motivation of Whitney, who coined the term 'matroid'.

The explicit enumeration of the set I is usually out of the question. So, in computational problems whose input is a matroid (S, I), the matroid is usually implicitly represented. The above examples illustrate this: a graphic matroid is represented by a graph G, and the matrix matroid is represented by a matrix M. The size of the input is then taken to be the size of G or M, not of $|I|$ which can exponentially larger.

¶33. Submatroids. Given matroids $M = (S, I)$ and $M' = (S', I')$, we call M' a **submatroid** of M if $S' \subseteq S$ and $I' \subseteq I$. There are two general methods to obtain submatroids, starting from a non-empty subset $R \subseteq S$:
(i) Induced submatroids. The **R-induced submatroid of** M is

$$M|R := (R, I \cap 2^R).$$

(ii) Contracted[7] submatroids. The **R-contracted submatroid of** M is

$$M \wedge R := (R, I \wedge R)$$

where $I \wedge R := \{A \cap R : A \in I, S - R \subseteq A\}$. Thus, there is a bijective correspondence between the independent sets A' of $M \wedge R$ and those independent sets A of M which contain $S - R$. Indeed, $A' = A \cap R$. Of course, if $S - R$ is dependent, then $I \wedge R$ is empty.

We leave it to an exercise to show that $M|R$ and $M \wedge R$ are matroids. Special cases of induced and contracted submatroids arise when $R = S - \{e\}$ for some $e \in S$. In this case, we say that $M|R$ is obtained by **deleting** e and $M \wedge R$ is obtained by **contracting** e.

¶34. Bases. Let $M = (S, I)$ be a matroid. If $A \subseteq B$ and $B \in I$ then we call B an **extension** of A; if $A = B$, the extension is **improper** and otherwise it is **proper**. A **base** of M (alternatively: a **maximal independent set**) is an independent set with no proper extensions. If $A \cup \{e\}$ is independent and $e \notin A$, we call $A \cup \{e\}$ a **simple extension** of A and say that e **extends** A. If $R \subseteq S$, we may relativize these concepts to R: we may speak of "$A \subseteq R$ being a base of R", "e extends A in R", etc. This is the same as viewing A as a set of the induced submatroid $M|R$.

[7] Contracted submatroids are introduced here for completeness. They are not used in the subsequent development (but the exercises refer to them).

¶35. Ranks. We note a simple property: *all bases of a matroid have the same size.* If A, B are bases and $|A| > |B|$ then there is an $e \in A - B$ such that $B \cup \{e\}$ is a simple extension of B. This is a contradiction. Note that this property is true even if S has infinite cardinality. Thus we may define the **rank** of a matroid M to be the size of its bases. More generally, we may define the rank of any $R \subseteq S$ to be the size of the bases of R (this size is just the rank of $M|R$). The **rank function**

$$r_M : 2^S \to \mathbb{N}$$

simply assigns the rank of $R \subseteq S$ to $r_M(R)$.

¶36. Problems on Matroids. A **costed matroid** is given by $M = (S, I; C)$ where (S, I) is a matroid and $C : S \to \mathbb{R}$. is a cost[8] function. The cost of a set $A \subseteq S$ is just the sum $\sum_{x \in A} C(x)$. The **maximum independent set problem** (abbreviated, MIS) is this: given a costed matroid $(S, I; C)$, find an independent set $A \subseteq S$ with maximum cost. A closely related problem is the **maximum base problem** where, given $(S, I; C)$, we want to find a base $B \subseteq S$ of maximum cost. If the costs are non-negative, then it is easy to see the MIS problem and the maximum base problem are identical. The following algorithm solves the maximum base problem:

```
GREEDY ALGORITHM FOR MAXIMUM BASE:
    Input: matroid M = (S, I; C) with cost function C.
    Output: a base A ∈ I with maximum cost.
    1.   Sort S = {x_1, ..., x_n} by cost.
         Suppose C(x_1) ≥ C(x_2) ≥ ··· ≥ C(x_n).
    2.   Initialize A ← ∅.
    3.   For i = 1 to n,
              put x_i into A provided this does not make A dependent.
    4.   Return A.
```

The steps in this abstract algorithm needs to be instantiated for particular representations of matroids. In particular, testing if a set A is independent is usually non-trivial (recall that matroids are usually given implicitly in terms of other combinatorial structures). We discuss this issue for graphic matroids below. It is interesting to note that the usual Gaussian algorithm for computing the rank of a matrix is an instance of this algorithm where the cost $C(x)$ of each element x is unit.

Let us see why the greedy algorithm is correct.

Lemma 10 (Correctness). *Suppose the elements of A are put into A in this order:*

$$z_1, z_2, \ldots, z_m,$$

where $m = |A|$. Let $A_i = \{z_1, z_2, \ldots, z_i\}$, $i = 1, \ldots, m$. Then:
1. A is a base.
2. If $x \in S$ extends A_i then $i < m$ and $C(x) \leq C(z_{i+1})$.
3. Let $B = \{u_1, \ldots, u_k\}$ be an independent set where $C(u_1) \geq C(u_2) \geq \cdots \geq C(u_k)$. Then $k \leq m$ and $C(u_i) \leq C(z_i)$ for all i.

[8] Recall our convention that costs may be negative. If the costs are non-negative, we call C a a "weight function".

*Proof.*1. By way of contradiction, suppose $x \in S$ extends A. Then $x \notin A$ and we must have decided not to place x into the set A at some point in the algorithm. That is, for some $j \le m$, $A_j \cup \{x\}$ is dependent. This contradicts the hereditary property because $A_j \cup \{x\}$ is a subset of the independent set $A \cup \{x\}$.
2. Suppose x extends A_i. By part 1, $i < m$. If $C(x) > C(z_{i+1})$ then for some $j \le i$, we must have decided not to place x into A_j. This means $A_j \cup \{x\}$ is dependent, which contradicts the hereditary property since $A_j \cup \{x\} \subseteq A_i \cup \{x\}$ and $A_i \cup \{x\}$ is independent.
3. Since all bases are independent sets with the maximum cardinality, we have $k \le m$. The result is clearly true for $k = 1$ and assume the result holds inductively for $k-1$. So $C(u_j) \le C(z_j)$ for $j \le k-1$. We only need to show $C(u_k) \le C(z_k)$. Since $|B| > |A_{k-1}|$, the exchange property says that there is an $x \in B - A_{k-1}$ that extends A_{k-1}. By part 2, $C(z_k) \ge C(x)$. But $C(x) \ge C(u_k)$, since u_k is the lightest element in B by assumption. Thus $C(u_k) \le C(z_k)$, as desired. **Q.E.D.**

From this lemma, it is not hard to see that an algorithm for the MIS problem is obtained by replacing the for-loop ("for $i = 1$ to n") in the above Greedy algorithm by "for $i = 1$ to m" where x_m is the last positive element in the list $(x_1, \ldots, x_m, \ldots, x_n)$.

¶37. Greedoids. While the matroid structure allows the Greedy Algorithm to work, it turns out that a more general abstract structure called **greedoids** is tailor-fitted to the greedy approach. To see what this structure looks like, consider the set system (S, F) where S is a non-empty finite set, and $F \subseteq 2^S$. In this context, each $A \in F$ is called a **feasible set**. We call (S, F) a **greedoid** if

Accessibility property If A is a non-empty feasible set, then there is some $e \in A$ such that $A \setminus \{e\}$ is feasible.

Exchange property: If A, B are feasible and $|A| < |B|$ then there is some $e \in B \setminus A$ such that $A \cup \{e\}$ is feasible.

Exercises

Exercise 7.1: Consider the graphic matroid in Figure 9. Determine its rank function. ♢

Exercise 7.2: The text described a modification of the Greedy Maximum Base Algorithm so that it will solve the MIS problem. Verify its correctness. ♢

Exercise 7.3:
(a) Interpret the induced and contracted submatroids $M|R$ and $M \wedge R$ in the bigraph of Figure 9, for various choices of the edge set R. When is $M|R = M \wedge R$?
(b) Show that $M|R$ and $M \wedge R$ are matroids in general. ♢

Exercise 7.4: Show that $r_M(A \cup B) + r_M(A \cap B) \le r_M(A) + r_M(B)$. This is called the **submodularity property** of the rank function. It is the basis of further generalizations of matroid theory. ♢

Exercise 7.5: In Gavril's activities selection problem, we have a set A of intervals of the form $[s, f)$. Recall that a subset $S \subseteq A$ is said to be compatible if S is pairwise disjoint. Does the set of compatible subsets of A form a matroid? If yes, prove it. If no, give a counter example. $\diamondsuit$

End Exercises

§8. Generating Permutations

In §1, we saw how the general bin packing problem can be reduced to linear bin packing. This reduction depends on the ability to generate all permutations of n elements efficiently. Since there are many uses for such permutation generators, we will take a small detour from greedy methods in order to address this interesting topic. A survey of this classic problem is given by Sedgewick [8]. Perhaps the oldest incarnation of this problem is the "change ringing problem" of bell-ringers in early 17th Century English churches [7]. This calls for ringing a sequence of n bells in all $n!$ permutations.

The problem of generating all permutations efficiently is representative of an important class of problems called **combinatorial enumeration**. For instance, we might want to general all size k subsets of a set, all graphs of size n, all convex polytopes based some given set of n vertices, etc. Such an enumerations would be considered optimal if the algorithm takes $O(1)$ time to generate each member.

It is good to fix some terminology. A n**-permutation** of a finite set X is a surjective function $p : \{1, \ldots, n\} \to X$. Surjectivity of p implies $n \geq |X|$. The function p may be represented by a sequence $(p(1), p(2), \ldots, p(n))$. Here we are interested in the case $n = |X|$, *i.e.*, permutation of *distinct* elements. We use a path-like notation for permutations, writing "$(p(1)–p(2)–\cdots–p(n))$" for the permutation $(p(1), p(2), \ldots, p(n))$.

Let S_n denote the set of all permutations of $X = \{1, 2, \ldots, n\}$; each element of S_n is called an n**-permutation**. Note that $|S_n| = n!$. E.g., the following is a listing of S_3:

$$(1–2–3), \quad (1–3–2), \quad (3–1–2); \quad (3–2–1), \quad (2–3–1), \quad (2–1–3). \tag{32}$$

Two n-permutations $\pi = (x_1– \cdots –x_n)$ and $\pi' = (x'_1– \cdots –x'_n)$ are **adjacent** (to each other) if there is some $i = 2, \ldots, n$ such that

$$x_i = \begin{cases} x'_{i-1} & \text{if } i = j \\ x'_{i+1} & \text{if } i = j - 1 \\ x'_i & \text{else.} \end{cases}$$

We may write $\pi' = Exch_i(\pi)$ in this case. E.g., $\pi = (1–2–4–3)$ and $\pi' = (1–4–2–3)$ are adjacent since $\pi' = Exch_3(\pi)$. An **adjacency ordering** of a set S of permutations is a listing of the elements of S such that every two consecutive permutations in this listing are adjacent. For instance, the listing of S_3 in (32) is an adjacency ordering.

[Figure: Adjacency Graph for 3-permutations]

We need another concept: if $\pi = (x_1– \cdots –x_{n-1})$ is an $(n-1)$-permutation, and π' is obtained from π by inserting the letter n into π, then we call π' an **extension** of π. Indeed, if n is inserted just before the ith letter in π, then we write $\pi' = Ext_i(\pi)$ for $i = 1, \ldots, n$. The meaning of "$Ext_n(\pi)$" should be clear: it is obtained by appending 'n' to the end of the sequence π. Note that there are n extensions of an $(n-1)$-permutation. E.g., if $\pi = (1{-}2)$ then the three extensions of π are $(3{-}1{-}2), (1{-}3{-}2), (1{-}2{-}3)$.

¶38. The Johnson-Trotter Ordering. Among the several known methods to generate all n-permutations, we will describe one that is independently discovered by S.M.Johnson and H.F.Trotter (1962), and apparently known to 17th Century English bell-ringers [7]. Hugo Steinhaus (1958) describes the problem of generating permutations by n particles moving along a line moving at variable speeds. The two main ideas in the Johnson-Trotter algorithm are (1) the n-permutations are generated as an adjacency ordering, and (2) the n-permutations are generated recursively. Suppose π is an $(n-1)$-permutation that has been recursively generated. Then we note that the n extensions of π can given one of two adjacency orderings. It is either

$$UP(\pi) : Ext_1(\pi), Ext_2(\pi), \ldots, Ext_n(\pi)$$

or the reverse sequence

$$DOWN(\pi) : Ext_n(\pi), Ext_{n-1}(\pi), \ldots, Ext_1(\pi).$$

E.g., $UP(1{-}2{-}3)$ is equal to

$$(4{-}1{-}2{-}3), \quad (1{-}4{-}2{-}3), \quad (1{-}2{-}4{-}3), \quad (1{-}2{-}3{-}4).$$

Note that if π' is another $(n-1)$-permutation that is adjacent to π, then the concatenated sequences

$$UP(\pi); DOWN(\pi')$$

and

$$DOWN(\pi); UP(\pi')$$

are both adjacency orderings. We have thus shown:

Lemma 11 (Johnson-Trotter ordering). *If $\pi_1, \ldots, \pi_{(n-1)!}$ is an adjacency ordering of S_{n-1}, then the concatenation of alternating DOWN/UP sequences*

$$DOWN(\pi_1); UP(\pi_2); DOWN(\pi_3); \cdots; DOWN(\pi_{(n-1)!})$$

is an adjacency ordering of S_n.

For example, starting from the adjacency ordering of 2-permutations $(\pi_1 = (1{-}2), \pi_2 = (2{-}1))$, our above lemma says that $DOWN(\pi_1), UP(\pi_2)$ is an adjacency ordering. Indeed, this is the ordering shown in (32).

Let us define the **permutation graph** G_n to be the bigraph whose vertex set is S_n and whose edges comprise those pairs of vertices that are adjacent in the sense defined for permutations. We note that the adjacency ordering produced by Lemma 11 is actually a cycle in the graph G_n. In other words, the adjacency ordering has the additional property that the first and the last permutations of the ordering are themselves adjacent. A cycle that goes through every vertex of a graph is said to be **Hamiltonian**. If $(\pi_1{-}\pi_2{-}\cdots{-}\pi_m)$ (for $m = (n-1)!$) is a Hamiltonian cycle for G_{n-1}, then it is easy to see that

$$(DOWN(\pi_1); UP(\pi_2); \cdots; UP(\pi_m))$$

is a Hamiltonian cycle for G_n.

¶39. The Permutation Generator. We proceed to derive an efficient means to generate successive permutations in the Johnson-Trotter ordering. We need an appropriate high level view of this generator. The generated permutations are to be used by some "permutation consumer" such as our greedy linear bin packing algorithm. There are two alternative views of the relation between the "permutation generator" and the "permutation consumer". We may view the consumer as calling[9] the generator repeatedly, where each call to the generator returns the next permutation. Alternatively, we view the generator as a skeleton program with the consumer program as a (shell) subroutine. We prefer the latter view, since this fits the established paradigm of BFS and DFS as skeleton programs (see Chapter 4). Indeed, we may view the permutation generator as a bigraph traversal: the implicit bigraph here is the permutation graph G_n.

In the following, an n-permutation is represented by the array $per[1..n]$. We will transform per by exchange of two adjacent values, indicated by

$$per[i] \Leftrightarrow per[i-1] \tag{33}$$

for some $i = 2, \ldots, n$, or

$$per[i] \Leftrightarrow per[i+1]$$

where $i = 1, \ldots, n-1$.

¶40. A Counter for n factorial. To keep track of the successive exchanges in Johnson-Trotter generator, we introduce an array of n counters

$$C[1..n]$$

where each $C[i]$ is initialized to 1 but always satisfying the relation $1 \le C[i] \le i$. Of course, $C[1]$ may be omitted since its value cannot change under our restrictions. The array counter C has $n!$ distinct values. We say the i-th counter is **full** iff $C[i] = i$. The **level** of the C is the largest index ℓ such that the ℓ-th counter is not full. If all the counters are full, the level of C is defined to be 1. E.g., $C[1..5] = [1, 2, 2, 1, 5]$ has level 4. We define the **increment** of this counter array as follows: if the level of the counter is ℓ, then (1) we increment $C[\ell]$ provided $\ell > 1$, and (2) we set $C[i] = 1$ for all $i > \ell$. E.g., the increment of $C[1..5] = [1, 2, 2, 1, 5]$ gives $[1, 2, 2, 2, 1]$. In code:

```
Inc(C)
    ℓ ← n.
    While (C[ℓ] = ℓ) ∧ (ℓ > 1)
        C[ℓ--] ← 1.
    If (ℓ > 1)
        C[ℓ]++.
    Return(ℓ)
```

Note that Inc returns the level of the original counter value. This macro is a generalization of the usual increment of binary counters (Chapter 6.1). For instance, for $n = 4$, starting with the initial value of $[1, 1, 1]$, successive increments of this array produce the following cyclic sequence:

$$\begin{aligned} C[2,3,4] &= [1,1,1] \to [1,1,2] \to [1,1,3] \to [1,1,4] \to [1,2,1] \\ &\to [1,2,2] \to [1,2,3] \to [1,2,4] \to [1,3,1] \to \cdots \\ &\to [2,3,3] \to [2,3,4] \to [1,1,1] \to \cdots . \end{aligned} \tag{34}$$

[9] The generator in this viewpoint is a **co-routine**. It has to remember its state from the previous call.

Let the cost of incrementing the counter array be equal to $n+1-\ell$ where ℓ is the level. CLAIM: the cost to increment the counter array from $[1,1,\ldots,1]$ to $[2,3,\ldots,n]$ is $< 2(n!)$. In proof, note that $C[\ell]$ is updated after every $n!/\ell!$ steps, so that the overall, $C[\ell]$ is updated $\ell!$ times. Hence the total number of updates for the $n-1$ counters is

$$n! + (n-1)! + \cdots + 2! < 2(n!),$$

which proves our Claim.

This gives us the top level structure for our permutation generator:

Johnson-Trotter Generator (Sketch)
Input: natural number $n \geq 2$
▷ *Initialization*
$per[1..n] \leftarrow [1,2,\ldots,n]$. ◁ *Initial permutation*
$C[2..n] \leftarrow [1,1,\ldots,1]$. ◁ *Initial counter value*
▷ *Main Loop*
do
$\ell \leftarrow Inc(C)$
UPDATE(ℓ) ◁ *The permutation is updates*
CONSUME(per) ◁ *Permutation is consumed*
While ($\ell > 1$)

The shell macro CONSUME is application-dependent. As default, we simply use it to print the current permutation.

¶41. How to update the permutation. We now describe the UPDATE macro. It uses the previous counter level ℓ to transform the current permutation to the next permutation. For example, the successive counter values in (34) correspond to the following sequence of permutations:

$$\begin{aligned} &\xrightarrow{[1,1,1]} (1{-}2{-}3{-}\underline{4}) \xrightarrow{[1,1,2]} (1{-}2{-}\underline{4}{-}3) \xrightarrow{[1,1,3]} (1{-}\underline{4}{-}2{-}3) \xrightarrow{[1,1,4]} (4{-}1{-}2{-}\underline{3}) \xrightarrow{[1,2,1]} (\underline{4}{-}1{-}3{-}2) \\ &\xrightarrow{[1,2,2]} (1{-}\underline{4}{-}3{-}2) \xrightarrow{[1,2,3]} (1{-}3{-}\underline{4}{-}2) \xrightarrow{[1,2,4]} (1{-}\underline{3}{-}2{-}4) \xrightarrow{[1,3,1]} (3{-}1{-}2{-}\underline{4}) \xrightarrow{[1,3,2]} \cdots \\ &\xrightarrow{[2,3,3]} (1{-}\underline{4}{-}2{-}3) \xrightarrow{[2,3,4]} (1{-}2{-}\underline{4}{-}3) \xrightarrow{[1,1,1]} (1{-}2{-}3{-}\underline{4}) \longrightarrow \cdots . \end{aligned} \tag{35}$$

To interpret the above, consider a general step of the form

$$\cdots \xrightarrow{[c_2,c_3,c_4]} (x_1{-}x_2{-}x_3{-}x_4) \xrightarrow{[c_2',c_3',c_4']} (x_1'{-}x_2'{-}x_3'{-}x_4') \cdots$$

We start with the counter value $[c_2, c_3, c_4]$ and permutation $(x_1{-}x_2{-}x_3{-}x_4)$. After calling `Inc`, the counter is updated to $[c_2', c_3', c_4']$, and it returns the level ℓ of $[c_2, c_3, c_4]$. If $\ell = 1$, we may[10] terminate; otherwise, $\ell \in \{2, 3, 4\}$. We find the index i such that $x_i = \ell$ (for some $i = 1, 2, 3, 4$). UPDATE will then exchange x_i with its neighbor x_{i+1} or x_{i-1}. The resulting permutation is $(x_1'{-}x_2'{-}x_3'{-}x_4')$.

In (35), we indicate x_i by an underscore, "$\underline{x_i}$". The choice of which neighbor (x_{i-1} or x_{i+1}) depends on whether we are in the "UP" phase or "DOWN" phase of level ℓ. Let $UP[1..n]$ be a Boolean array where $UP[\ell]$ is true in the UP phase, and false in the DOWN phase when we

[10] In case we want to continue, the case $\ell = 1$ is treated as if $\ell = n$. E.g., in (35), the case $\ell = 1$ is treated as $\ell = 4$.

are incrementing a counter at level ℓ. Moreover, the value of $UP[\ell]$ is changed (flipped) each time $C[\ell]$ is reinitialized to 1. For instance, in the first row of (35), $UP[4] =$ `false` and so the entry 4 is moving down with each swap involving 4. In the next row, $UP[4] =$ `true` and so the entry 4 is moving up with each swap.

Hence we modify our previous Inc macro to include this update:

```
INCREMENT(C)
Output: Increments C, updates UP, and returns the previous level of C
    ℓ ← n.
    While (C[ℓ] = ℓ) ∧ (ℓ > 1)    ◁ Loop to find the counter level
        C[ℓ] ← 1;
        UP[ℓ] ← ¬UP[ℓ];    ◁ Flips the boolean value UP[ℓ]
        ℓ--.
    If (ℓ > 1)
        C[ℓ]++.
    Return(ℓ).
```

For a given level ℓ, the UPDATE macro need to find the "position" i where $per[i] = \ell$ ($i = 1, \ldots, n$). We could search for this position in $O(n)$ time, but it is more efficient to maintain this information directly: let $pos[\ell]$ denote the current position of ℓ. Thus the $pos[1..n]$ is just the inverse of the array $per[1..n]$ in the sense that

$$per[pos[\ell]] = \ell \qquad (\ell = 1, \ldots, n).$$

We can now specify the UPDATE macro to update both pos and per:

```
UPDATE(ℓ)
    if (UP[ℓ])
        per[pos[ℓ]] ⇔ per[pos[ℓ] + 1];    ◁ modify permutation
        pos[per[pos[ℓ]]] ← pos[ℓ];    ◁ update position array
        pos[ℓ]++;    ◁ update position array
    else
        per[pos[ℓ]] ⇔ per[pos[ℓ] − 1];
        pos[per[pos[ℓ]]] ← pos[ℓ];
        pos[ℓ]--;
```

Thus, the final algorithm is:

```
JOHNSON-TROTTER GENERATOR
Input: natural number n ≥ 2
    ▷ Initialization
        per[1..n] ← [1, 2, . . . , n].    ◁ Initial permutation
        pos[1..n] ← [1, 2, . . . , n].    ◁ Initial positions
        C[2..n] ← [1, 1, . . . , 1].    ◁ Initial counter value
    ▷ Main Loop
        do
            ℓ ← Increment(C);
            UPDATE(ℓ);    ◁ The permutation is updated
            CONSUME(per);    ◁ Permutation is consumed
        While(ℓ > 1)
```

The Java code for the Johnson-Trotter Algorithm is presented in an appendix of this chapter.

Remarks:

1. We can introduce early termination criteria into our permutation generator. For instance, in the bin packing application, there is a trivial lower bound on the number of bins, namely $b_0 = \lceil (\sum_{i=1}^{n} w_i)/M \rceil$. We can stop when we found a solution with b_0 bins. If we want only an approximate optimal, say within a factor of 2, we may exit when the we achieve $\le 2b_0$ bins.
2. We have focused on permutations of distinct objects. In case the objects may be identical, more efficient techniques may be devised. For more information about permutation generation, see the book of Paige and Wilson [6]. Knuth's much anticipated 4th volume will treat permutations; this will no doubt become a principle reference for the subject.
3. The Johnson-Trotter enumeration of permutations can be extended to the enumeration of all strings of length n over a finite alphabet Σ. Now, we define two strings to be **adjacent** if they differ at only one position. For $\Sigma = \{0, 1\}$, the most famous such enumerations is the Gray code (or reflected binary code): if $G(n)$ is the Gray code for strings of length n, then $G(n+1) = 0 \cdot G(n); 1 \cdot G^R(n)$ where $G^R(n)$ is the reverse (reflected) enumeration. E.g., $G(2) = (00, 01, 11, 10)$ and $G(3) = 0 \cdot (00, 01, 11, 10); 1 \cdot (10, 11, 01, 00)(000, 001, 011, 010; 110, 111, 101, 100)$. An application is physical implementation of counters. To count to 2^n, we want to cycle through all the binary strings of length n. Using the standard binary representation, a counter increment may cause more than one bit to change. If each bit is controlled by some physical switch, we might produce intermediate states due to the switches not changing simultaneously. This problem does not arise if we use an enumerations like the Gray code.

Exercises

Exercise 8.1:
(a) Draw the adjacency bigraph corresponding to 4-permutations. HINT: first draw the adjacency graph for 3-permutations and view 4-permutations as extension of 3-permutations.
(b) How many edges are there in the adjacency bigraph of n-permutations?
(c) What is the radius and diameter of the bigraph in part (b)? [See definition of radius and diameter in Exercise 4.8 (Chapter 4).] ♢

Exercise 8.2: Another way to list all the n-permutations in S_n is lexicographic ordering: $(x_1 - \cdots - x_n) < (x'_1 - \cdots - x'_n)$ if the first index i such that $x_i \neq x'_i$ satisfies $x_i < x'_i$. Thus the lexicographic smallest n-permutation is $(1-2-\cdots-n)$. Give an algorithm to generate

n-permutations in lexicographic ordering. Compare this algorithm to the Johnson-Trotter algorithm. $\diamondsuit$

Exercise 8.3: All adjacency orderings of 2- and 3-permutations are cyclic. Is it true of 4-permutations? $\diamondsuit$

Exercise 8.4: Two n-permutations π, π' are **cyclic equivalent** if $\pi = (x_1 - x_2 - \cdots - x_n)$ and $\pi' = (x_i - x_{i+1} - \cdots - x_n - x_1 - x_2 - \cdots - x_{i-1})$ for some $i = 1, \ldots, n$. A **cyclic n-permutation** is an equivalence class of the cyclic equivalence relation. Note that there are exactly n permutations in each cyclic n-permutation. Let S'_n denote the set of cyclic n-permutations. So $|S'_n| = (n-1)!$. Again, we can define the cyclic permutation graph G'_n whose vertex set is S'_n, and edges determined by adjacent pairs of cyclic permutations. Give an efficient algorithm to generate a Hamiltonian cycle of G'_n. $\diamondsuit$

Exercise 8.5: Suppose you are given a set S of n points in the plane. Give an efficient method to generate all the convex polygons whose vertices are from S. Give the complexity of your algorithm as a function of n. $\diamondsuit$

End Exercises

§A. APPENDIX: Java Code for Permutations

```
/****************************************************
 * Per(mutations)
 *      This generates the Johnson-Trotter permutation order.
 *      By n-permutation, we mean a permutation of the symbols {1,2,...,n}.
 *
 * Usage:
 *      % javac Per.java
 *      % java Per [n=3] [m=0]
 *
 *      will print all n-permutations.  Default values n=3 and m=0.
 *      If m=1, output in verbose mode.
 *      Thus "java Per" will print
 *            (1,2,3), (1,3,2), (3,1,2), (3,2,1), (2,3,1), (2,1,3).
 *      See Lecture Notes for details of this algorithm.
 *
 ****************************************************/

public class Per {

 // Global variables
 ////////////////////////////////////////////////////
 static int n;                    // n-permutations are being considered
                    // Quirk: Following arrays are indexed from 1 to n
 static int[] per;                // represents the current n-permutation
 static int[] pos;                // inverse of per:  per[pos[i]]=i (for i=1..n)
 static int[] C;                  // Counter array: 1 <= C[i] <= i (for i=1..n)
 static boolean[] UP;             // UP[i]=true iff pos[i] is increasing
                                  //     (going up) in the current phase

 // Display permutation or position arrays
 ////////////////////////////////////////////////////
 static void showArray(int[] myArray, String message){
   System.out.print(message);
   System.out.print("(" + myArray[1]);
   for (int i=2; i<=n; i++)
     System.out.print("," + myArray[i]);
   System.out.println(")");
 }

 // Print counter
 ////////////////////////////////////////////////////
 static void showC(String m){
   System.out.print(m);
   System.out.print("(" + C[2]);
   for (int i=3; i<=n; i++)
     System.out.print("," + C[i]);
   System.out.println(")");
 }

 // Increment counter
 ////////////////////////////////////////////////////
 static int inc(){
   int ell=n;
   while ((C[ell]==ell) && (ell>1)){
        UP[ell] = !(UP[ell]);      // flip Boolean flag
```

```
      C[ell--]=1;
  }
  if (ell>1)
      C[ell]++;
  return ell;                     // level of previous counter value
}

// Update per and pos arrays
////////////////////////////////////////////////////
static void update(int ell){
  int tmpSymbol;            // this is not necessary, but for clarity
  if (UP[ell]) {
    tmpSymbol = per[pos[ell]+1];  // Assert: pos[ell]+1 makes sense!
    per[pos[ell]] = tmpSymbol;
    per[pos[ell]+1] = ell;
    pos[ell]++;
    pos[tmpSymbol]--;
  } else {
    tmpSymbol = per[pos[ell]-1];  // Assert: pos[ell]-1 makes sense!
    per[pos[ell]]= tmpSymbol;
    per[pos[ell]-1] = ell;
    pos[ell]--;
    pos[tmpSymbol]++;
  }
}

// Main program
////////////////////////////////////////////////////
public static void main (String[] args)
       throws java.io.IOException
{
  //Command line Processing
  n=3;                    // default value of n
  boolean verbose=false;  // default is false (corresponds to second argument = 0)
  if (args.length>0)
      n = Integer.parseInt(args[0]);
  if ((args.length>1) && (Integer.parseInt(args[1]) != 0))
      verbose = true;

  //Initialize
  per = new int[n+1];
  pos = new int[n+1];
  C   = new int[n+1];
  UP  = new boolean[n+1];
  for (int i=0; i<=n; i++) {
    per[i]=i;
    pos[i]=i;
    C[i]=1;
    UP[i]=false;
  }

  //Setup For Loop
  int count=0;             // only used in verbose mode
  int ell=1;
  System.out.println("Johnson-Trotter ordering of "+ n + "-permutations");
  if (verbose)
      showArray(per, count + ", level="+ ell + " :\t" );
  else
      showArray(per, "");
```

```
  //Main Loop
  do {
    ell = inc();
    update(ell);
    if (verbose)
       count++;
       showArray(per, count + ", level="+ ell + " :\t" );
    else
       showArray(per, "");
  } while (ell>1);

 }//main
}//class Per
```

References

[1] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Comm. of the ACM*, 29(4):320–330, 1986.

[2] X. Cai and Y. Zheng. Canonical coin systems for change-making problems. *arXiv:0809.0400v1 [cs.DS]*, 2009. 14 pages.

[3] S. K. Chang and A. Gill. Algorithmic solution of the change-making problem. *J. ACM*, 17(1):113–122, 1970.

[4] J.Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, IT-24:530–536, 1978.

[5] D. Kozen and S.Zaks. Optimal bounds for the change-making problem. *Theor. Computer Science*, 123(2):377–388, 1994.

[6] E. Page and L. Wilson. *An Introduction to Computational Combinatorics*. Cambridge Computer Science Texts, No. 9. Cambridge University Press, 1979.

[7] T. W. Parsons. Letter: A forgotten generation of permutations, 1977.

[8] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9(2):137–164, 1977.

[9] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA, 1974.

[10] J. S. Vitter. The design and analysis of dynamic huffman codes. *J. ACM*, 34(4):825–845, 1987.