"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

¶1. 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 $G(x)$.

Even when our simple greedy method is not optimal, we may be able to prove that it is only some factor off from optimal, say, a factor of 2. In this case, we say it has "approximation ratio" of 2. This leads to another theme in this chapter: the design of greedy methods with good approximation ratios.

The greedy method is supposed to exemplify the idea of "local search". But a closer examination of many greedy algorithms will reveal some global information being used. The global information is usually minimal or easily obtained, such as doing a global sort. Indeed, the preferred data structure for delivering this sorting information is the priority queue. The ith step in the above decision sequence follows this sorting order.

We begin with three classes of greedy problems: the bin packing problems, the coin changing problems and interval selection problems. 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 extremely simple problem called **linear bin packing**. But it will open the door to a major topic in combinatorial algorithms called bin packing. In fact, the general bin packing problem is an example of the NP-complete problems which are not known to have polynomial-time solutions.

¶2. 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. There is a weight limit $M > 0$ for all cars. The number of riders in a car is immaterial, as long as their total weight is $\le M$ pounds. We may assume that no rider has weight $> M$. The way that riders are put into cars is controlled by two policies:

(1) **Online policy**: this means we must make a decision for each rider at the moment that he or she arrives at the head of the queue. This decision may depend on earlier riders in the queue, but not depend on subsequent riders. In the current model, this decision is binary: "take the current car" or "take the next car".

(2) **First come, first ride policy** (FCFR). In other words, there is no "jumping the queue" in getting into a car.

Therefore, whenever we make the decision "take the next car", we must immediately dispatch the current car, and call for the next empty car in order to satisfy the FCRC policy. That is because our joy ride model assumes that there is only one car for loading riders. In Exercises, we explore policies where if you have a "two loading cars" model or are allowed to peek ahead into the queue.

These two policies are independent. (1) Suppose we have the online policy without the FCRC policy: after we ask a rider to "take the next car", we need not call for the new car! We can continue loading riders into the current car. In the meantime, we have a secondary queue of riders waiting to take the next car. Of course, this secondary queue must never exceed M in total weight. (2) We can also have the FCRC policy *without* the online policy. This setting becomes necessary with negative weights. See Exercise below.

Example. Let the weight limit be $M = 400$ (in pounds) and the weights of the riders in the queue be

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

where 30 represents the front of the queue. We can load the riders into 3 cars as follows:

$$Solution_1 : (30, 190, 80;\ 210, 100, 80;\ 50)$$

where the semi-colons separate successive car loads. For easy read, we henceforth drop the last digit of 0 from these weights, and simply write

$$Solution_1 : (3, 19, 8;\ 21, 10, 8;\ 5). \tag{2}$$

What is our goal in this problem? *It is to minimize the number of cars used.* $Solution_1$ uses three cars. We may verify that $Solution_1$ conforms to the Online and FCFR policies. We call $Solution_1$ the **greedy solution** for the instance (1). In the greedy solution, we always make the decision "take the current car" if it does not violate the weight limitation. It is easy to see that the greedy solution is always exist and is unique. But $Solution_1$ is not the only one to satisfy our policies. Here are two others:

$$Solution_2 : (3, 19;\ 8, 21;\ 10, 8, 5).$$

$$Solution_3 : (3, 19;\ 8, 21;\ 10, 8;\ 5).$$

They do not improve upon the greedy solution in terms of the number of cars. In fact, $Solution_3$ is worse because it uses 4 cars. Nevertheless we are prompted to ask if there exists instances where a non-greedy solution is better than the greedy one?

Leaving this question aside for now, let us illustrate our introductory remark about global information in greedy algorithms. Suppose we place riders into the cars in two phases. In phase one, we sort the weights w in (1), giving us a new queue:

$$Sort(w) = (3, 5, 8, 8, 10, 19, 21). \tag{3}$$

In phase two, we apply the greedy algorithm to $Sort(w)$, giving us the solution

$$Solution_4 : (3, 5, 8, 8, 10;\ 19, 21).$$

This uses only two cars, improving our greedy algorithm. Decisions exploiting sorting is using global information about the queue w. Thus $Solution_4$ has violated both our policies.

¶3. Linear Bin Packing. The solutions compliant with the FCFR Policy can be formalized as "linear solutions". Given an integer M and a sequence $w = (w_1, \ldots, w_n)$ of weights satisfying $0 < w_i \le M$ $(i = 1, \ldots, n)$, a **linear solution** of (M, w) is given by a sequence

$$0 < t(1) < t(2) < \cdots < t(k) = n. \tag{4}$$

of k **breakpoints** (for some $k \ge 1$). These k breakpoints define k bins $B_1, \ldots, B_k$ where $B_i := (w_{t(i-1)+1}, \ldots, w_{t(i)})$. If $i = 1$, let $t(0) = 0$ by definition. We say the linear solution is **feasible** if each B_i contains a total weight at most M. E.g., the greedy solution (2) is a feasible linear solution with three breakpoints: $t(1) = 3, t(2) = 6, t(3) = 7$. A feasible linear solution is **optimal** if k is minimum over all feasible solutions. The **linear bin packing problem** is to compute an optimal linear solution for any input M, w.

¶4. Greedy Algorithm. It is instructive to write out the Greedy Algorithm for linear bin packing problem (a.k.a. joy ride problem) in **pseudo-code**. An important criterion for pseudo-code that it exposes the control-loop structures of the algorithm. Critical variables used in these control-loops should also be exposed. We are happy with English descriptions of variables, etc.

Let $w = (w_1, w_2, \ldots, w_n)$ be the input sequence of weights and C denote a container (or bin or car) that is being filled with elements of w. Let the variable W keep track of the sum of the weights in C.

Greedy Algorithm for Linear Bin Packing:
Input: (M, w) where $w = (w_1, \ldots, w_n)$, each $w_i \le M$.
Output: Container sequence $(C_1; C_2; \cdots; C_m)$ representing a 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 $M < W + w_i)$
      Append C as C_i to the output sequence
      $C \leftarrow \emptyset$, $W \leftarrow 0$
    $W \leftarrow W + w_i$; $C \leftarrow C \cup \{w_i\}$.

Pseudo-code is for human understanding. It is deliberately short of any actual programming language because programming languages are meant for computers (to be compiled). Our pseudo-code exploits the power of mathematical notations *and* the linguistic structures of English which humans understand best. Of course, English can be replaced by any other natural language. Also there are many possible levels of detail, depending on the goals of the pseudo-code. The above pseudo-code achieves two informal goals:

(P1) It can be "directly" transcribed into common programming languages, assuming you know how to use common data types like arrays or linked lists.

(P2) It exposes enough details for its complexity to be analyzed up to Θ-order. The Θ-order complexity of common data types are assumed known.

We urge the student to check out (G1). To illustrate (G2), we prove that the complexity is $O(n)$: There is a loop with n iterations. Inside the loop, each step is $O(1)$. For instance, we may assume that the container C is represented by a linked list, and so the step $C \leftarrow C \cup \{w_i\}$ amounts to appending to a linked list.

We need an important assumption in (G2) which is often taken for granted: we assume the real RAM computational model of Chapter 1. In this model, we can compare and perform arithmetic operations on any two real numbers in constant time.

¶5. Optimality of Greedy Algorithm It may not be obvious why the greedy algorithm produces an optimal linear solution. The proof is instructive.

Theorem 1 *The greedy solution is optimal for linear bin packing.*

Proof. Suppose the greedy algorithm outputs k bins as defined by the sequence of breakpoints

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

Let us compare the greedy solution with some optimal solution with these breakpoints

$$0 = s(0) < s(1) < s(2) < \cdots < s(\ell) = n.$$

By way of contradiction, assume the greedy algorithm is not optimal, i.e.,

$$\ell < k. \tag{5}$$

Now we compare $s(i)$ with $t(i)$ for $i = 1, \ldots, \ell$. Note that

(a) We have $s(1) \leq t(1)$ because $t(1)$ is obtained by the greedy method.

(b) We have $s(\ell) > t(\ell)$ because
$$t(\ell) < t(k) = n = s(\ell).$$

It follows that $\ell > 1$. So there is a smallest index i^* ($2 \leq i^* \leq \ell$) such that $s(i^*) > t(i^*)$. Then

$$s(i^* - 1) \leq t(i^* - 1) < t(i^*) < s(i^*). \tag{6}$$

The weights in the i^*-th bin of the optimal solution is given by the subsequence $w(s(i^* - 1) .. s(i^*)]$. But according to (6), this subsequence contains

$$w(t(i^* - 1) .. t(i^*) + 1]. \tag{7}$$

This expression is well-defined since $t(i^*) + 1 \le s(i^*) \le n$. By definition of the greedy algorithm, the total weight in (7) exceeds M (otherwise the greedy algorithm would have added $w_{t(i^*)+1}$ to its i^*th car). This is our desired contradiction. **Q.E.D.**

¶6. Global Bin Packing. The joy ride problem is a restricted version of the following **(Global) Bin Packing Problem**:

Given a multiset $S = \{w_1, \dots, w_n\}$ of weights where each $w_i \in (0, 1]$, find a partition[1]

$$S = S_1 \uplus S_2 \uplus \cdots \uplus S_m \tag{8}$$

of S into the minimum number $m \ge 1$ of multisets where each S_i has a total weight at most 1.

Call (8) an **optimal solution** to the Bin Packing instance S and also denote the minimum m by $Opt(S)$. Note that we have departed from the Linear Bin Packing problem in two ways: (1) The bin capacity is now $M = 1$. This is not an significant departure since we can divide the original input weights by the capacity M to obtain $w_i \in (0, 1]$. This process is called **normalization**, and w_i are the **normalized weights**. In fact, we will often write

$$S = \tfrac{1}{M}\{w_1, \dots, w_n\}$$

where w_i are the original weights.

(2) The input weights are not ordered. This is a more profound change because we are no longer dealing with a "linear" problem. Alternatively, if the input is given as a list $(w_1, \dots, w_n)$, we are free to reorder them anyway we like.

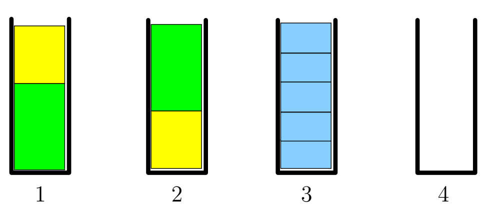


Figure 1: Bin packing solution.

E.g., if $S = \frac{1}{5}\{1, 1, 1, 3, 2, 2, 1, 3, 1\}$ then one global solution (before normalization) 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.

Suppose that instead of computing the global solution (8), we only want to compute the value of $Opt(S)$. That is, you only want to know the optimal number of bins, not how the weights are put into bins. We may call this the **#Bin Packing Problem**, and it is a simple but useful surrogate for Bin Packing. Of course, if you can compute (8), you can get $Opt(S)$. The converse is less clear.

[1] Recall that $\uplus$ is a way of annotating set union to say that the sets are disjoint.

As far as we know, one cannot do much better than the brute force method for computing $Opt(S)$. What do we mean by "brute force"? It is to try all possibilities. But what are these possibilities? First, we must give concrete representations of S and its solutions. First, we represent the set S by any listing of its elements in an arbitrary order:

$$w = (w_1, \ldots, w_n). \tag{9}$$

If S_n denote the set of all permutations of $[1..n]$, and $\sigma \in S_n$, then $\sigma(w) := (w_{\sigma(1)}, \ldots, w_{\sigma(n)})$. Likewise, we represent any solution $B_1, \ldots, B_m$ (8) to S by any permutation $\pi \in S_n$ in which

$$\pi(w) = (w_{\pi(1)}, \ldots, w_{\pi(n)}) \tag{10}$$

lists the weights in B_1 first (in any order), followed by the weights in B_2, etc. It is clear that there are many representation of any solution; that is OK since S_n is a very large set!

OBSERVATION: if (10) represents a global solution to the bin packing instance (9), then

$$\boxed{\mathtt{Grd}(\pi(w)) = Opt(w)}$$

where $\mathtt{Grd}(w')$ is the number of bins used by greedy algorithm on any input w'.

It follows that

$$Opt(w) = \min_{\sigma \in S_n} \mathtt{Grd}(\sigma(w)).$$

The "brute force" algorithm for $Opt(w)$ will use the greedy algorithm of linear bin packing solution as a subroutine: it cycles through each of the $n!$ permutations of S_n, and for each σ, computes $\mathtt{Grd}(\sigma(w))$ in $O(n)$ time. The minimum of these values is $Opt(w)$. We conclude that the brute force method has a complexity of

$$\Theta(n \cdot n!) = \Theta((n/e)^{n+(3/2)}).$$

Use the Θ-form of Stirling's approximation

Here, we assume that we can generate all n-permutations in $O(n!)$ time. This is a nontrivial assumption; see §8 for details on how to do this.

Karp and Held noted that 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. Since there are $(n-1)!$ such permutations, we obtain:

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

See Exercise where we further exploit this idea to improve the brute force algorithm by any polynomial factor. The relation between global bin packing and linear bin packing is worth noting: by imposing restrictions on the space of possible solutions, we have turned a difficult problem like global bin packing into a feasible one like linear bin packing. The latter problem is in turn useful as a subroutine for the original problem.

¶7. How good is the Greedy Solution? How good is the greedy algorithm `Grd` when viewed as an algorithm for global bin packing? We are not interested in goodness in terms of computational complexity. Instead we are interested in the quality of the output of `Grd`, namely the number of bins, $\mathtt{Grd}(w)$. We shall compare $\mathtt{Grd}(w)$ to $\mathtt{Opt}(w)$, the optimal solution. Since $\mathtt{Grd}(x)/\mathtt{Opt}(w) \geq 1$, we want to upper bound the ratio $\mathtt{Grd}(w)/\mathtt{Opt}(w)$.

Theorem 3 *For any unit weight sequence w,*

$$\mathtt{Opt}(w) \geq 1 + \lfloor \mathtt{Grd}(w)/2 \rfloor \tag{11}$$

Moreover, for each n, there are weight sequences with $\mathtt{Opt}(w) = n$ for which (11) is an equality.

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

$$W_i + W_{i+1} > 1. \tag{12}$$

To see this, note that the first weight v to be put into the $i+1$st bin by the greedy algorithm must satisfy $W_i + v > 1$. This implies (12) since $W_{i+1} \geq v$. Summing up (12) for $i = 1, 3, 5, \ldots, 2\lfloor k/2 \rfloor - 1$, we obtain $\sum_{i=1}^k W_i > \lfloor k/2 \rfloor$. The last inequality implies that the optimal solution needs more than $\lfloor k/2 \rfloor$ bins, i.e., $\mathtt{Opt}(w) \geq 1 + \lfloor k/2 \rfloor$. This proves (11). To see that the inequality (11) is sharp, consider the following[2] unit weight sequence of length n:

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

The greedy solution uses n bins, but clearly $\mathtt{Opt}(w) = 1 + \lfloor n/2 \rfloor$.

Q.E.D.

¶8. Absolute and Asymptotic Approximation Ratios. We said the global bin packing is an important problem for which there are no polynomial-time algorithms. So it is essential to seek good approximations to global bin packing. Let A be any bin packing algorithm. To evaluate the performance of A, consider the simplest definition that comes to mind: define the **absolute approximation ratio** of A as

$$\alpha_0(A) := \sup_w A(w)/\mathtt{Opt}(w) \tag{13}$$

where w range over all non-empty weight sequences. By definition $\alpha_0(A) \geq 1$. Our goal is to design algorithms A with small $\alpha_0(A)$.

What is not to like about α_0? Let us consider the absolute approximation ratio of `Grd`. The problem is that $\alpha_0(A)$ may determined by a single input w rather by those w with arbitrarily large $\mathtt{Opt}(w)$. For instance, suppose A_1 has the property that

$$A_1(w) = 2 \cdot \mathtt{Opt}(w) + 1 \tag{14}$$

for all w. For each $n \in \mathbb{N}$, there is some weight sequences w_n such $\mathtt{Opt}(w_n) = n$. We would conclude that $\alpha_1(A_1) = 3$ since

$$\alpha_0(A_1) = \sup_w \frac{A_1(w)}{\mathtt{Opt}(w)} = \sup_w \frac{2 \cdot \mathtt{Opt}(w) + 1}{\mathtt{Opt}(w)} = 2 + \sup_w \tfrac{1}{\mathtt{Opt}(w)} = 3,$$

[2]Thanks to Jason Y. Lee (2008) for this simplification.

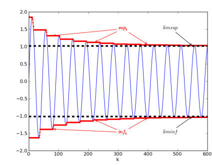
since $\sup_w \frac{1}{\mathtt{Opt}(w)} = \frac{1}{\mathtt{Opt}(w_1)} = 1$. The worst case is controlled by the "trivial" input w_1. This conclusion seems artificial. Intuitively, we think that a correct definition of approximation ratio ought to yield the conclusion that "$\alpha(A_1) = 2$". Here is the definition to achieve this: Define the **(asymptotic) approximation ratio** of algorithm A as

$$\alpha(A) := \limsup_{n\to\infty} a_n$$

where $a_n = \sup\left\{\frac{A(w)}{\mathtt{Opt}(w)} : \mathtt{Opt}(w) = n\right\}$. Recall from mathematical analysis that the **limit superior** of an infinite sequence of numbers $(x_1, x_2, \ldots)$ is given by

$$\limsup_{n\to\infty} x_n := \lim_{n\to\infty} \{\sup\{x_i : i \ge n\}\}.$$

source Wikipedia



For the algorithm A_1, we define $a_n = \sup\left\{\frac{A_1(w)}{\mathtt{Opt}(w)} : \mathtt{Opt}(w) = n\right\} = \frac{A_1(w_n)}{\mathtt{Opt}(w_n)} = (2n+1)/n$. Then $\alpha(A_1) = \limsup_{n\to\infty} a_n = 2$. The behavior of our hypothetical A_1 is rather like Grd, as seen in Theorem 3.

Lemma 4 *The greedy algorithm* Grd *for Linear Bin Packing has (asymptotic) approximation ratio*

$$\alpha(\mathtt{Grd}) = 2. \tag{15}$$

Proof. Note that Theorem 3 implies that $\mathtt{Opt}(w) \ge 1 + \lfloor \mathtt{Grd}(w)/2 \rfloor \ge \frac{\mathtt{Grd}(w)}{2} + \frac{1}{2}$. This implies

$$\frac{\mathtt{Grd}(w)}{\mathtt{Opt}(w)} \le 2 - \frac{1}{\mathtt{Opt}(w)}. \tag{16}$$

This implies $\alpha(\mathtt{Grd}) \le 2$ (Careful: we are not entitled to say "$\alpha(\mathtt{Grd}) < 2$"). Moreover, Theorem 3 also shows that for each odd integer $n \ge 2$, there are inputs with $\mathtt{Opt}(w) = n$ for which (16) is an equality. This implies $\alpha(\mathtt{Grd}) \ge 2 - \varepsilon$ for any $\varepsilon > 0$. By definition of lim sup, $\alpha(\mathtt{Grd}) \ge 2$. This proves (15). **Q.E.D.**

¶9. First Fit Algorithm. In order to beat the approximation factor of 2 in (15), we must give up the "linearity constraint" of linear bin packing. We consider the following "non-linear" bin packing algorithm:

> First Fit Algorithm
> Given $w = (w_1, \ldots, w_n)$:
> Initialize n empty bins: $B_1, B_2, \ldots, B_n$.
> For $i = 1, \ldots, n$,
> *place w_i into the first bin B_j that it fits into.*
> Return the sequence of non-empty bins $(B_1, \ldots, B_m)$.

Let $\mathtt{FF}(w)$ denote the number of bins used by the **First Fit Algorithm**. We leave it as an Exercise to give a more detailed pseudo-code for this algorithm, keeping in mind our goals for pseudo-code ¶4.

Example: Let $M = 7$ and $w = (3,5,4,1,3,2,3)$. It is easy to check that $\texttt{Opt}(M,w) = 3$ with the bins $(5,2),(4,3),(3,3,1)$. The greedy achieves $\texttt{Grd}(M,w) = 5$ with $(3;5;4,1;3,2;3)$. The following simulation of the First Fit Algorithm shows that $\texttt{FF}(M,w) = 4$:

i	w_i	B_1	B_2	B_3	B_4
1	3	3			
2	5	3	5		
3	4	3 4	5		
4	1	3 4	5 1		
5	3	3 4	5 1	3	
6	2	3 4	5 1	3 2	
7	3	3 4	5 1	3 2	3

FF was one of the earliest bin packing algorithms to be studied. Johnson, Ullman, Garey and others began its analysis in the 1970s.

Theorem 5 $\alpha(\texttt{FF}) = 17/10$.

In fact, it is now known that even the absolute approximation ratio of FF is 1.7:

$$\alpha_0(\texttt{FF}) = 17/10.$$

This is a much more intricate result that is only proved in 2013 by Gy. Dósa and J. Sgall [4].

Let us first prove a lower bound on $\alpha(\texttt{FF})$, somewhat less than the optimal bound of 1.7:

$$\alpha(\texttt{FF}) \ge 5/3 = 1.6\overline{6}. \tag{17}$$

Let w be a sequence of $18n$ weights given in 3 groups:

$$w = (\underbrace{0.15,\ldots,0.15}_{6n}, \underbrace{0.34,\ldots,0.34}_{6n}, \underbrace{0.51,\ldots,0.51}_{6n})$$

Clearly, $\texttt{Opt}(w) = 6n$ since the total weight is $(0.15+0.34+0.51)6n = 6n$ and so $6n$ bins are necessary. It is also clearly sufficient. We next claim that $FF(w) = 10n$: FF puts the first group into n bins (each bin having weight $0.15 \times 6 = 0.9$), second group in $3n$ bins (each bin having weight $0.34 \times 2 = 0.68$), and the last group in $6n$ bins (each bin of weight 0.51). So $FF(w)/\texttt{Opt}(w) = (n+3n+6n)/6n = 5/3$. as claimed in (17).

The upper bound in Theorem 5 is from Ullman (1971) who showed $\texttt{FF}(w) \le 1.7Opt(w) + 3$. We give a compact proof from Gy. Dósa and J. Sgall (see [4]). The structure of this highly stylized proof should be first understood:

The **value** of each item $w_i \in w$ is $v(w_i) := \frac{6}{5}w_i + b(w_i)$ where $b(x)$ is the **bonus** defined by

$$b(x) := \begin{cases} 0 & \text{if } x \le \frac{1}{6}, \\ \frac{3}{5}(x - \frac{1}{6}) & \text{if } \frac{1}{6} < x < \frac{1}{3}, \\ 0.1 & \text{if } \frac{1}{3} \le x \le \frac{1}{2}, \\ 0.4 & \text{if } x > \frac{1}{2}. \end{cases}$$

The bonus function $b(x)$ is continuous except at $x = 1/2$. We extend these definitions to any set B of items: let $v(B) := \sum_{x\in B} v(x)$ and $b(B) := \sum_{x\in B} b(x)$. Also, let the **size** of the bin be given by $s(B) := \sum_{x\in B} x$. Thus $v(B) = b(B) + s(B)$. If B are the items in a bin from bin packing, then clearly $s(B) \le 1$. GRAPH of $b(x)$:

For any set B of items, if $s(B) \le 1$ then $v(B) \le 1.7$: because $\frac{6}{5}s(B) \le 1.2$ and it is sufficient to show that $v(B) \le 0.5$. (see paper)

Heuristics for Bin Packing. The First Fit algorithm, also known as **Greedy First Fit**, is based on the **First Fit Heuristic** (FF) in which each weight w_i is placed into the first (smallest j) bin B_j in which it fits. This heuristic assumes a sequence of bins $B_1, \ldots, B_n$ that remain open for packing additional weights, if they are not full. What we have called the Greedy algorithm for Linear Bin Packing, $\mathtt{Grd}(w)$, is based on what is known as the **Next Fit Heuristic** (NF). Hence another name for `Grd` is **Greedy Next Fit**. Finally, the **Best Fit Heuristic** (BF) is one that packs w_i into the bin whose residual capacity exceeds w_i by the least amount. You can easily guess how the **Greedy Best Fit** Algorithm should behave.

¶10. Bin Packing on Weight Decreasing Sequences. The next idea is to introduce sorting. Heuristically, it seems a good idea to pack the larger weights before the smaller ones. Let $\mathtt{Sort}(w)$ return the sequence w in decreasing sorting order (breaking ties arbitrarily) E.g., $\mathtt{Sort}(2, 3, 1, 2) = (3, 2, 2, 1)$. Adding this heuristic to `Grd` and to `FF`, we obtain two new bin packing algorithms:

- **Next Fit Decreasing**: $\mathtt{NFD}(w) := \mathtt{Grd}(\mathtt{Sort}(w))$
- **First Fit Decreasing**: $\mathtt{FFD}(w) := \mathtt{FF}(\mathtt{Sort}(w))$

Our previous bound that $\alpha(\mathtt{Grd}) = 2$ is no longer valid for $\alpha(\mathtt{NFD})$, but we have this[3] lower bound:

Lemma 6 *(Next Fit Decreasing)* $\alpha(\mathtt{NFD}) \ge 5/3 = 1.6\overline{6}$.

Proof. Consider this sorted sequence of $3n$ weights in three equal size groups:

$$w = \left(\tfrac{1}{2}+e, \ldots, \tfrac{1}{2}+e; \tfrac{1}{3}+e, \ldots, \tfrac{1}{3}+e; \tfrac{1}{7}+e, \ldots, \tfrac{1}{7}+e\right).$$

where $\frac{1}{2} + \frac{1}{3} + \frac{1}{7} + 3e = 1$. So $e = 1/126$. Thus $\mathtt{Opt}(w) = n$. The greedy algorithm uses n bins for the first group. The first weight of $\frac{1}{3} + e$ fits into the previous bin. The next $n-1$ weights fit into $(n-1)/2$ bins (assume n is odd). The first weight of $\frac{1}{7} + e$ fits into the previous bin. The remaining $n-1$ weights fit into $(n-1)/6$ bits (assume $n-1$ is divisible by 6). Thus $\mathtt{NFD}(w) = n + (n-1)/2 + (n-1)/6 = (5n-2)/3$, or $\mathtt{NFD}(w)/\mathtt{Opt}(w) = 5/3 - (2/3n)$. We conclude that $\alpha(\mathtt{NFD}) \ge 5/3$. **Q.E.D.**

[3] From notes of Peter Sanders.

Theorem 7 *(First Fit Decreasing)*

$$1.2\overline{2} = 11/9 \le \alpha(\texttt{FFD}) \le 1.5.$$

Remark: the tight upper bound of

$$\alpha(\texttt{FFD}) \le 11/9 = 1.2\overline{2}$$

is known; but we state a weaker result here because its proof is relatively simple.

Proof. The lower bound on $\alpha(\texttt{FFD})$ comes from the following weight sequence $w = (w_1, \ldots, w_{30n})$ where

$$w_i = \begin{cases} \frac{1}{2} + \varepsilon & (i = 1, \ldots, 6n), \\ \frac{1}{4} + 2\varepsilon & (i = 6n+1, \ldots, 12n), \\ \frac{1}{4} + \varepsilon & (i = 12n+1, \ldots, 18n), \\ \frac{1}{4} - 2\varepsilon & (i = 18n+1, \ldots, 30n). \end{cases}$$

The optimal packing uses $9n$ bins: we can pack $6n$ bins with weights $(\frac{1}{2}+\varepsilon, \frac{1}{4}+\varepsilon, \frac{1}{4}-2\varepsilon)$, and $3n$ bins with weights $(\frac{1}{4}+2\varepsilon, \frac{1}{4}+2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon)$. However, we see that FFD uses $11n$ bins: it fills the first $6n$ bins with $(\frac{1}{2}+\varepsilon, \frac{1}{4}+2\varepsilon)$, the next $2n$ bins with $(\frac{1}{4}+\varepsilon, \frac{1}{4}+\varepsilon, \frac{1}{4}+\varepsilon)$, and the last $3n$ bins with $(\frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon)$.

To prove the upper bound[4] on $\alpha(\texttt{FFD})$, it suffices to prove that for any unit weight sequence w,

$$\texttt{FFD}(w) \le 1.5 \cdot \texttt{Opt}(w) + 1. \tag{18}$$

Suppose $\texttt{FFD}(w) = k$. Consider the first $m = \lfloor 2k/3 \rfloor$ bins, say $B_1, \ldots, B_m$. We claim that $\texttt{Opt}(w) \ge m$. There two cases. (1) Suppose bin B_m contains a weight $> 1/2$ (for $i = 1, \ldots, m$). Thus $\texttt{Opt}(w) \ge m$ as claimed. (2) Suppose bin B_m contains only weights that are $\le 1/2$. Therefore, the remaining $k-m$ bins contains only weights that are $\le 1/2$. This implies each of these $k-m$ bins contains at least two weights. So there are $2(k-m) = 2\lceil k/3 \rceil \ge 2k/3 \ge m$ weights in these bins. Take any m of these weights $v_1, v_2, \ldots, v_m$. None of these weights fits into the first m bins. Therefore $v_i + |B_i| > 1$ for $i = 1, \ldots, m$ (where $|B_i|$ is the total weight contained in B_i), i.e., $\texttt{Opt}(w) \ge m$. So we have proved that $\texttt{Opt}(w) \ge m$. It follows that $\texttt{Opt}(w) \ge m = \lfloor 2k/3 \rfloor \ge (2k-2)/3$, or $k = \texttt{FFD}(w) \le 1.5 \cdot \texttt{Opt}(w) + 1$. This concludes our demonstration of (18) **Q.E.D.**

The efficient implementation of Step 4 in the First Fit algorithm is interesting: 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 this 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 $u.\texttt{key}$ which is the maximum residual capacity of the bins in the subtree at u. Initially, each W_i is zero, and so $u.\texttt{key} = M$ 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:

[4] Thanks to Chien-Chin Huang (Fall 2013) for a simplified argument.

```
    INSERT(w_i):
1.  Initialize u to the root of T.
2.  While (u is not a leaf)
3.      If (u.left.key ≥ w_i)
            u ← u.left
4.      else
            u ← u.right
5.  u.key ← u.key − w_i     ◁ u is a leaf
6.  While (u is not the root)
7.      u ← u.p    ◁ move to the parent
8.      u.key ← max {u.left.key, u.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.

¶11. Two-Car Loading. Consider the extension of linear bin packing where we simultaneously load two cars. Call these two cars the **front** and **rear cars**. This is a realistic joy ride scenario. It will 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 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 using 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.

What is a good design $\texttt{Grd}_2$ for 2-car loading? The goal, as usual, is to minimize the number of cars used for any given input sequence w. we want to ensure $\texttt{Grd}_2$ is at least as good as $\texttt{Grd}$, the loading policy of the original greedy algorithm (¶4). More precisely, for all $w = (w_1, \ldots, w_n)$, we require

$$\texttt{Grd}_2(w) \le \texttt{Grd}(w). \tag{19}$$

A trivial way to ensure (19) is to just just imitate $\texttt{Grd}$! This means (19) is actually an equality, but it is not very interesting. We want a policy $\texttt{Grd}_2$ where, in addition to (19), there are many inputs w where $\texttt{Grd}_2(w) < \texttt{Grd}(w)$, with quantifiable advantages.

Consider the following definition of $\texttt{Grd}_2$: *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.* We leave as an exercise to show (19). But $\texttt{Grd}_2$ can be strictly better than $\texttt{Grd}$: For instance, if $w = (30, 190, 80, 210, 90, 80, 50)$ and $M = 400$ in our example in (1), then the first fit policy gives an improvement: $\texttt{Grd}_2(w) = 2 < 3 = \texttt{Grd}(w)$.

We can extend the 2-car loading framework: these extensions):

- 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.

- 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.

¶12. Extensions. There are many ways to extend the bin packing problem. Let us consider the higher dimensional version of this problem. In particular, the packing of rectangles into rectangular bins. One interesting simplified 2-dimension problem is to consider one bin of unit width but infinite height: we want to pack the rectangles so as to minimize the height of the packing. A popular computer game is based on this problem.

What kind of bin packing is Tetris? Is it an online problem?

Let us consider another simplified 2-dimensional problem: the bins are unit squares, and the input is a sequence of squares. Let $w = (w_1, w_2, \ldots, w_n)$ where $0 < w_i \le 1$ represents an input square of width w_i. Things are complicated enough in 2D that we will begin with assuming that the w_i are already sorted by non-increasing weights.

We start with the simplest greedy packing heuristic that we can analyze. Consider the[5] following **cell packing** idea. Suppose we subdivide each bin into **cells** using the usual quadtree method: a bin is a cell, and given any cell, we can recursively split it into 4 congruent subcells. We want to put the w_i's into cells, at most one per cell! Moreover, each cell is guaranteed to be at least 1/4 full. The latter condition is easy to fulfill: if a cell is not 1/4 full, we can split it first, and use one of its child. It is not hard to prove that this gives an approximation ratio $\alpha(A) = 1/4$.

Exercises

Exercise 1.1: We consider linear bin packing problem in which the weights w_i's can be negative.
(a) Show that $\texttt{Grd}(w)$ is no longer optimal for linear bin packing.
(b) How bad can $\texttt{Grd}(w)$ compared to the optimal linear bin packing solution? Please quantify "badness" in some reasonable way. ♢

Exercise 1.2: There are two places where our optimality proof for the greedy algorithm breaks down when there are negative weights. What are they? ♢

Exercise 1.3: Consider the linear bin packing problem when the weight w_i's can be negative. A solution is determined by its 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: for $i \ge 1$, assuming n_{i-1} is defined, let n_i to be the largest index (but at most n) such that $\sum_{j=n_{i-1}+1}^{n_i} w_j \le M$. Either prove that this solution is optimal (in the sense of linear bin packing), or give a counter example. NOTE: this algorithm is no longer "online". ♢

[5] Thanks to Shravas Rao (Fall'2013) for this suggestion.

Exercise 1.4: Give an $O(n^2)$ algorithm for linear bin packing when there are negative weights.
HINT: When solving the problem for (M, w), assume that you already know the solution for each (M, w') where w' is a suffix of w. $\diamondsuit$

Exercise 1.5: 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 1.6: Above we extended the Karp-Held idea of improving brute-force by another factor of n. Can this be further extended? $\diamondsuit$

Exercise 1.7: 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 $\texttt{Grd}'_2(w)$ be the number of cars used by this strategy on input w (say capacity is 1). This is open ended: compare $\texttt{Grd}'_2(w)$ to $\texttt{Grd}$ and $\texttt{Grd}_2$. $\diamondsuit$

Exercise 1.8: Let $\texttt{Grd}_2(w)$ 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 $\texttt{Grd}_2(w) > G'_2(w)$.
(b) Show an example where $\texttt{Grd}_2(w) < G'_2(w)$. $\diamondsuit$

Exercise 1.9: 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. $\diamondsuit$

Exercise 1.10: (Open ended) Explore the revoking of decisions in 1- or 2-car loading. $\diamondsuit$

Exercise 1.11: (Open ended) Quantify the improvements possible when loading 2 cars in parallel tracks instead of loading 2 cars in a single track. $\diamondsuit$

Exercise 1.12: 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 $i \leq k \leq i'$ and $j \leq \ell \leq j'$. 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. $\diamondsuit$

Exercise 1.13: 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 1.14: 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. HINT: There is an example with 7 vertices.
(b) Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ where each variable 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
$$x_i + x_j \ge 1.$$

Let $\boldsymbol{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$. If the assignment $\boldsymbol{x} \leftarrow \boldsymbol{a}$ 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 $\boldsymbol{x}$,

$$|\boldsymbol{x}^*| \le |\boldsymbol{x}|. \tag{20}$$

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) There exists an optimal $\boldsymbol{x}^*$ that is half-integer valued, i.e., each component is $0, \frac{1}{2}$ or 1.
(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 $|\lfloor \boldsymbol{x}^* \rceil| \le 2|C^*|$. ♢

Exercise 1.15: Consider the following trivial vertex cover algorithm:

Trivial Vertex Cover($G(V, E)$):
1. Initialize $C \subseteq V, M \subseteq E$ to be empty sets
2. While $E \neq \emptyset$
3. Pick any edge $u - v \in E$
4. $M \leftarrow M \cup \{u - v\}$
5. $C \leftarrow C \cup \{u, v\}$
6. Remove from E any edge that is incident on either u or v
7. Return C

(a) Show that the output C is at most twice the size of the minimum vertex cover. HINT: the set $M \subseteq E$ is useful for this proof. For two sets A and B, you can show that $|A| \leq |B|$ by showing that there exists an injective function $f : A \to B$.

(b) Given that the trivial algorithm is also a 2-approximation algorithm for Vertex Cover, are there reasons to use the rounding algorithm $\lfloor \boldsymbol{x}^* \rceil$ of the previous algorithm? ♢

Exercise 1.16: Suppose you have an algorithm to solve the #Bin Packing Problem. Can you use this algorithm as a subroutine to find an optimal bin packing? How often do you need to call the subroutine? ♢

Exercise 1.17: For $k \geq 1$, 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 $C(u) \neq C(v)$. 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 $i \geq 2$, $C(v_i)$ is the smallest integer $k \geq 1$ 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 paid your grocer bill of \$5.75 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 get, say, three singles and 5 quarters even though this is the right amount. Your expectation suggests that there is a deterministic algorithm that cashiers use under normal circumstances.

greed and money, yes, they go together

In fact, it is a greedy algorithm. What problem does this greedy algorithm solve? It is popularly known as the **coin changing problem** although paper bills also count as coins here.

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,

$$D = [1, 5, 10, 25, \$1, \$5, \$10, \$20, \$50, \$100]. \tag{21}$$

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

In (21), 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 (21), 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.

¶13. 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" is 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 $\texttt{Opt}_D(x)$ (resp., $\texttt{Grd}_D(x)$) denote any optimal (resp., the greedy) solution, assuming x is a multiple of d_1. Clearly, we have $|\texttt{Opt}_D(x)| \le |\texttt{Grd}_D(x)|$. We say D is **canonical** if $|\texttt{Grd}_D(x)| = |\texttt{Opt}_D(x)|$ for all $x \in \mathbb{N}$ that is a multiple of d_1. For instance, $D = [1, 2, 4, 5]$ is not canonical. To see this, notice that greedy solution for $x = 8$ is $(1, 1, 0, 1)$ of size 3. The optimal solution is $(0, 0, 2, 0)$ with size 2. On the other hand $D = [1, 2, 4, 8]$ is canonical because for any x, the greedy solution (s_1, s_2, s_3, s_4) has $s_4 = \lfloor x/8 \rfloor$ which is necessary for optimality, and the remainder $x \mod 8$ has a unique optimal solution in the system $[1, 2, 4]$. This argument shows that any binary currency system $[1, 2, 4, \ldots, 2^{m-1}, 2^m]$ is canonical. The key open problem in coin changing is *to characterize all canonical systems.* Surprisingly, this problem remain unsolved.

the main problem!

Besides canonicity, another desirable property is uniqueness: D has **uniqueness** if $\texttt{Opt}_D(x)$ 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 $x = 4$ has two solutions, $(1, 0, 1) \equiv (0, 2, 0)$.

If $s = (s_1, \ldots, s_m)$ and $s' = (s'_1, \ldots, s'_m)$ then we write $s \le s'$ to mean that $s_i \le s'_i$ for all i. Call s a **subsolution** of s'. The following is easy to see: *Optimal solutions are closed under the* *subsolutions*. In other words, subsolutions of an optimal solution are optimal. This is a form of the Dynamic Programming Principle which we will address in Chapter 7.

To show non-canonicity, we need counter examples. If $|\mathtt{Grd}_D(x)| > |\mathtt{Opt}_D(x)|$, 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 $\mathtt{Opt}(x) = (s_1^*, \ldots, s_m^*)$ is an optimum solution for such an x then

$$s_i \cdot s_i^* = 0 \qquad \text{(for all } i = 1, \ldots, m) \tag{22}$$

where $\mathtt{Grd}_D(x) = (s_1, \ldots, s_m)$. 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, $\mathtt{Grd}(x')$ is obtained from $\mathtt{Grd}(x)$ by subtracting 1 from the i-th component. Also, $|\mathtt{Opt}(x')| \le |\mathtt{Opt}(x)| - 1$. We know that $|\mathtt{Opt}(x)| < |\mathtt{Grd}(x)|$. Thus $|\mathtt{Opt}(x')| \le |\mathtt{Opt}(x)| - 1 < |\mathtt{Grd}(x)| - 1 = |\mathtt{Grd}(x')|$. 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 for any x, we claim

$$\mathtt{Grd}(x) \le q(D) \tag{23}$$

in a componentwise manner: if $\mathtt{Grd}(x) = (s_1, \ldots, s_m)$ then $s_i \le q_i$ for all i. If (23) is violated at some i, say $s_i > q_i$, then we can construct a solution s' that is lexicographically larger than s (in particular, we can make $s'_{i+1} > s_i$). This contradicts the definition of s as the greedy solution.

¶14. Is the US Currency System canonical? Not all currency system is canonical – the pre-1971 British currency system is non-canonical (Exercise). Experience suggests that the US Currency system is canonical. But how do we prove this?

have you ever met a counter example?

Towards this end, let us say one currency systems D' is an **extension** of another D when D is a prefix of D'. Consider the following sequence of extensions:

$$D_1 = [1, 2, 4], \quad D_2 = [1, 2, 4, 5], \quad D_3 = [1, 2, 4, 5, 8].$$

We already noted that D_1 is canonical and D_2 is not. Thus, there are non-canonical extensions of canonical ones. Moreover, it can be shown that D_3 is canonical (Cai-Zheng). Thus, there are also canonical extensions of non-canonical ones. These examples hint at the difficulty of canonicity.

Consider two kinds of extensions of $D = [d_1, \ldots, d_m]$:

- **Type A extension** is $D' = [d_1, \ldots, d_m, d_{m+1}]$ where $d_{m+1} = d_m \cdot q$ for some $q \ge 2$. E.g., $[1, 5, 10]$ is a type A extension of $[1, 5]$.
- **Type B extension** is $D' = [d_1, \ldots, d_m, d_{m+1}, d_{m+2}]$ where

$$\left.\begin{array}{rcll} d_{m+1} & = & d_m \cdot q & \text{(for some } q \ge 2), \\ d_{m+2} & = & d_{m+1} \cdot q' + d_m \cdot r & \text{(for some } q' \ge 1, r \ge 1. \end{array}\right\} \tag{24}$$

Call (q, q', r) a set of **parameters** of the Type B extension. E.g., $[1, 5, 10, 20, 50]$ is a type B extension of $[1, 5, 10]$ with parameters $(q, q', r) = (2, 2, 1)$.

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 it a **reduced set of parameters** for the Type B extension.

Examples: Trivially, $D = [d_1]$ and $D = [1, d_2]$ is canonical for any $d_1 \ge 1$ and $d_2 > 1$. The system $D = [1, 2, 3]$ is a Type B extension with parameters $(q, q', r) = (2, 1, 1)$. 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 (21) 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 8 *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 $D = [d_1]$ (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 $\mathtt{Grd}_{D'}(x) = (s'_1, s'_2)$ then $s'_2 \ge 1$. If $\mathtt{Opt}_{D'}(x) = (s^*_1, s^*_2)$ then by property (22), $s^*_2 = 0$. Thus, we can view $\mathtt{Opt}_{D'}(x)$ as a D-solution for x. Since D is canonical, this proves that $|\mathtt{Opt}_{D'}(x)| \ge |\mathtt{Grd}_D(x)|$. Thus,

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

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

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

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 $\mathtt{Grd}_{D''}(x) = (a, 0, 1)$ 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 (22)). Since (s^*_1, s^*_2) is equal to $\mathtt{Grd}_{D'}(x)$ (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 $a + r < q$, $|s^*| = a + r + q' \ge a + 1 = |\mathtt{Grd}_{D''}(x)|$. If $a + r \le q$, $|s^*| = a + (r - q) + 1 + q' \ge a + 1 = |\mathtt{Grd}_{D''}(x)|$. In either case, we have shown that $|s^*| \ge |\mathtt{Grd}_{D''}(x)|$, 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 $|\mathtt{Opt}_{D''}(x)| \le |\mathtt{Grd}_{D''}(x)|$. The same argument as before will lead to the contradiction that $|\mathtt{Opt}_{D''}(x)| > |\mathtt{Grd}_{D''}(x)|$.

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 (21) is uniquely canonical.

¶15. 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 $m=3$ [6], and our Type A and Type B extensions have roots there. 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 $D=[1,5,10,25,50,100,200]$. 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: 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 2.3: In 1971, the British denomination converted to a decimal system. The old system[6] has these denominations:

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

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

Yogi Berra was referring to this in the introductory quote!

Exercise 2.4: Show by a direct argument that the binary system $D=(1,2,4,\ldots,2^m)$ is a canonical system that is also unique. Direct argument means to avoid quoting our theorem on extensions. ♢

[6]There was an obsolete coin, the Guinea which is equal to 21 shillings.

Exercise 2.5: (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 2.6: (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 (4))

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

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 $Min(w, \epsilon)$. ♢

Exercise 2.7: 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? ♢

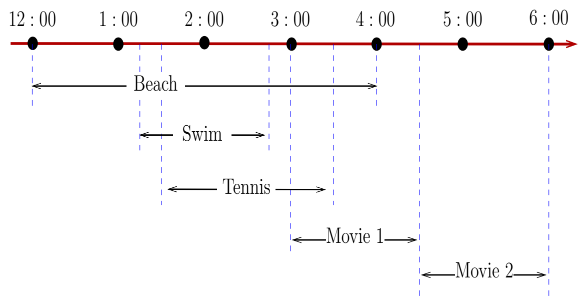
End Exercises

§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.

¶16. Intervals as Activities. 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 our activity 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:



$$\begin{array}{ll} \text{beach} & 12:00-4:00, \\ \text{swim} & 1:15-2:45, \\ \text{tennis} & 1:30-3:20, \\ \text{movie} & 3:00-4:30, \\ \text{movie} & 4:30-6:00. \end{array}$$

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*

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

of intervals, 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 S of intervals
Output: $A \subseteq S$, a set of compatible intervals
▷ *Initialization*
  Sort S according to some numerical criterion.
    Let $(I_1, \ldots, I_n)$ be the sorted sequence.
  Let $A = \emptyset$.
▷ *Main Loop*
  For $i = 1$ to n
    If $A \cup \{I_i\}$ is compatible, add I_i to A
  Return(S)

Thus, A 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 A.

But what greedy criteria should we use for sorting? As usual, ties are broken arbitrarily. Here are four possibilities. Notice that we say "sort in increasing order", instead of the more accurate "sort in non-decreasing' order'. I prefer this direct formulation, *always assuming that ties are broken 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, tennis, beach

(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 criteria yield an optimal solution. Tally: (a) 11, (b) 0, (c) 16, (d) 20. Wisdom of the crowd?

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 and "weak" ones that depend on how ties are broken.

Optimality of sorting by increasing finish times: We use a proof by contradiction, reminiscent of the joy ride proof. Let $A = (I_1, I_2, \ldots, I_k)$ 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 $A^* = (I_1^*, I_2^*, \ldots, I_\ell^*)$ is an optimal solution where $I_i^* = [s_i^*, f_i^*)$ and again $f_1^* < f_2^* < \cdots < f_\ell^*$. By optimality of A^*, 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: non-optimality of the greedy solution means $k < \ell$. Therefore the interval I_{k+1}^* is defined. Then

$$\begin{aligned} f_k &\le f_k^* && \text{(by CLAIM)} \\ &\le s_{k+1}^* && (I_{k+1}^* \text{ is defined and } I_k^*, I_{k+1}^* \text{ have no conflict}) \end{aligned}$$

and so I_{k+1}^* is compatible with $\{I_1, \ldots, I_k\}$. This is a contradiction since the greedy algorithm halted after choosing I_k – it could have continued with I_{k+1}^*.

What is the running time of this algorithm? In deciding if interval I_i is compatible with the current set A, 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.

¶17. 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: The text gave four different greedy criteria (a)-(d) for the activities selection problem.
(i) Show that (b), (c), (d) are suboptimal using "strong" counter examples (we prefer visualized intervals). Extra credit if (b), (c), (d) are shown with the *same* set of activities. How small can this set of activities be?
(i) Each of the criteria (a')-(d') have an inverted version in which we sort in decreasing order. Again, one of these inverted criteria is optimal, and the other three suboptimal. Prove the optimality of one, and provide counter examples for the other three. ♢

Draw your counter examples!

Exercise 3.2: Suppose the input $S = (I_1, \ldots, I_n)$ 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. Let $S = \{I_1, \ldots, I_n\}$ be a set of activities where each activity $I_i = [s_i, f_i)$ is a half-open interval. We want to find a compatible set $A \subseteq S$ which maximizes the **length** $|A|$ where

$$|A| := \sum_{I \in A} |I|$$

and $|I_i| := f_i - s_i$. Denote by $Opt(S)$ the maximum length of $A \subseteq S$.
Let $S_{i,j} = \{I_i, I_{i+1}, \ldots, I_j\}$ for $i \le j$ and $Opt_{i,j} := Opt(S_{i,j})$.

(a) Show by a counter example that the following algorithm does not work:

$$Opt_{i,j} = \max\{Opt_{i,k} + Opt_{k+1,j} : i \le k \le j-1\} \tag{25}$$

HINT: May assume $|S| \le 4$ in the counter example.

(b) Give an $O(n \log n)$ algorithm for computing $Opt_{1,n}$.
HINT: order the activities in the set S according to their finish times.

♢

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!) $\diamondsuit$

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. $\diamondsuit$

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.

¶18. An Encoding Problem 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 a principal application for such a problem. A computer file may be regarded as a string s, so problem (P) can be called the **file compression problem**. Very often, the characters in computer files are encoded [7] in ASCII characters. In modern computers, these files are called **text files**. Each ASCII character is represented by an 8-bit binary string, and so the alphabet Σ is often identified with the set $\{0,1\}^8$ of size $2^8 = 256$. This is a false identification since no one really thinks of the letter $a \in \Sigma$ as a particular sequence of 8 bits. It is more accurate to view Σ as some *a priori set*[8] of symbols, and the ASCII code as a bijective map

$$C_{asc} : \Sigma \to \{0,1\}^8 . \tag{26}$$

This code is a *fixed-length binary code*: $|C_{asc}(x)| = 8$ for all $x \in \Sigma$. So the ASCII encode of a file of m characters is a binary string of length $8m$. Can we do better, i.e., can we choose a different code that uses less than $8m$ bits? Basically, this is the problem of data compression.

Gauss appears to be referring to a kind of data compression or removing redundancy in the epigraph of this Chapter

[7] We simply say "ASCII characters" to cover both the original 128 ASCII characters, as well as its extension to the 256 extended ASCII characters.

[8] By an "a priori set" we mean an abstract set that the human mind has already comprehended, prior to any formalization or representation. Thus the map C_{asc} is therefore only an mental association of such a set Σ with a concrete set $\{0,1\}^8$.

Problem (P) suggests the use of **variable length code** to take exploit the relative frequency of characters in Σ. For instance, in typical English texts, the letters 'e' and 't' are most frequent and it is a good idea to use shorter codes for them. On the other hand, infrequent letters like 'q' or 'z' could have longer codes. According to some statistics (see §5, Exercises), the relative frequencies of 'e', 't', 'q', 'z' are $12.02, 9.10, 0.11, 0.07$, respectively. 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 $s = x_1 x_2 \cdots x_m \in \Sigma^*$ is then encoded as

$$C(s) := C(x_1)C(x_2)\cdots C(x_m) \in \{0,1\}^*. \tag{27}$$

This raises the problem of **decoding** $C(s)$, *i.e.*, recovering s from $C(s)$. For a general code C and s, we cannot expect unique decoding. A basic requirement for decodability is the ability to detect the boundary between code words in $C(s)$. One solution is to introduce a new symbol '\$' and insert it between successive $C(x_i)$'s. If we insist on the binary alphabet for the code, this may force us to convert, say, '0' to 00, '1' to 01 and '\$' to 11. This doubles the number of bits, probably wasteful.

¶19. Prefix-free codes. Our preferred solution for unique decoding is that C be **prefix-free**. This means that if $a, b \in \Sigma$ are distinct letters then $C(a)$ is not a prefix of $C(b)$. Clearly, this ensures unique decoding. With suitable preprocessing (basically to construct the "code tree" for C, defined next) decoding can be done quite simply, in an on-line fashion.

We represent a prefix-free code C by an external 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.

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

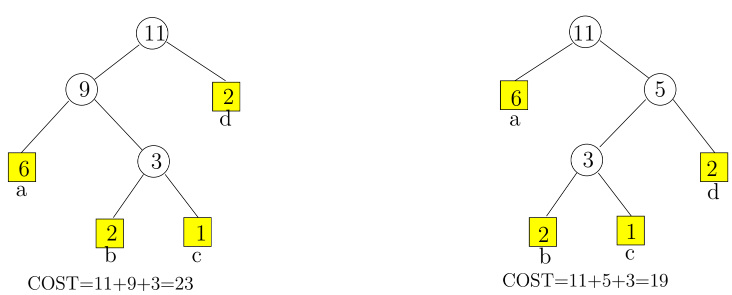


Figure 2: Code trees for two prefix-free codes: assume $f(\mathtt{a}) = 6$, $f(\mathtt{b}) = 2$, $f(\mathtt{c}) = 1$ and $f(\mathtt{d}) = 2$.

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. *Henceforth, we assume that $|\Sigma| \ge 2$ and all code trees are full binary trees.* The case $|\Sigma| = 1$ is clearly very special, and we may ignore it. Returning to the informal problem

$|\Sigma| \ge 2$!

(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 $C(s)$. Clearly, the only statistics important about s is the frequency $f_s(x)$ of each letter x in s, i.e., the number of occurrences of x in s. In general, call a function[9] of the form

$$f : \Sigma \to \mathbb{N} \tag{28}$$

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

$$COST(f, C) := \sum_{a \in \Sigma} |C(a)| \cdot f(a). \tag{29}$$

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).

Consider a simple example[10] where s is "`abadacadaba`" (So $\Sigma = \{$`a`, `b`, `c`, `d`$\}$) with frequencies of the characters a, b, c, d equal to $6, 2, 1, 2$ (respectively). For the codes in Figure 2, the cost of the first code is $6 \cdot 2 + 2 \cdot 3 + 1 \cdot 3 + 2 \cdot 1 = 23$. The second code is better, with cost 19.

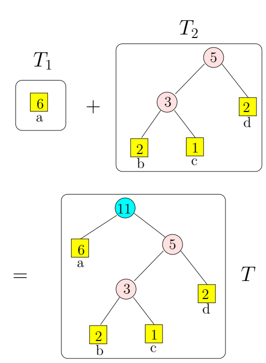
¶20. Huffman Code Algorithm. We can now restate the informal problem (P) as the precise **Huffman coding problem**:

> (H) 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 $3, 9, 11$ and so the $COST(T_{f,C}) = 3 + 9 + 11 = 23$. In general, the reader may verify that

$$COST(f, C) = COST(T_{f,C}). \tag{30}$$

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 subtree of T. If T_1, T_2 are weighted code trees, the result T is also a weighted code tree.



[9] Sometimes, frequency is regarded as a fraction between zero and one. But we view it as an counting value: perhaps "census function" is a better term here.

[10] Regard it as toddler's version of the incantation "`abracadabra`" or its more palindromic form "`abradacadabra`" The palindromic nature on this incantation is supposed to be part of the magic.

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 Q be a set of weighted code trees. Initially, Q has $n = |\Sigma|$ trivial trees, each with only one node representing a single character in Σ.
> 2. While Q has more than one tree,
>   2.1. Extract $T, T' \in Q$ of minimum and next-to-minimum weights.
>   2.2. Merge T, T' and insert the result $T + T'$ into Q.
> 3. Now Q 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 the Huffman code algorithm is nondeterministic – when two trees have the same weight, the algorithm may pick either one. For instance, the first tree in Figure 2 is non-Huffman because its node of weight 9 cannot be a result of the Huffman algorithm; however, the second tree in Figure 2(b) is Huffman.

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 most 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.

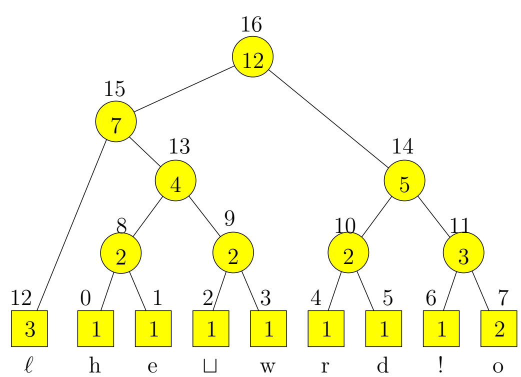


Figure 3: Huffman Tree for "`hello world!`": weights are inside each node, but ranks $(0, 1, \ldots, 16)$ are beside the nodes.

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 the order in which nodes were extracted from the priority queue. For instance, the leaves `h` (rank 0) and `e` (rank 1) 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.

¶21. 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 Q containing a set 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).

But the code tree T will be used for determining the code word $C(a)$ for each $a \in \Sigma$. The weights are no longer relevant. Instead T is viewed as an external BST in the sense of ¶III.33. Why? Recall that we want to convert our input string $s = x_1 \cdots x_m$ into the binary string $C(s) = C(x_1) \cdots C(x_m)$ as in (27). For each x_i, we use T to compute $C(x_i)$ as follows: do a binary search for x_i in T, and output a 0 (resp., 1 for each comparison that makes us turn left (resp., right). Hence we must modify the above Huffman tree algorithm to construct such an external BST. Recall (§III.5) that if v is an external node of such a BST, then $v.\texttt{left}.\texttt{key} = v.\texttt{key}$ and $v.\texttt{left}$ is a pointer to the predecessor of v (if it exists). Suppose T_L and T_R are two external BST's to be merged. We take these steps to create the desired merger:

- Create a new node u.
- Make T_L and T_R the left and right subtrees of u.
- If v is the minimum external node of T_R, then $v.\texttt{left} \leftarrow u$, and $u.\texttt{key} \leftarrow v.\texttt{key}$.

¶22. 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 9 (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. The proof follows from a simple observation: suppose a and b are two leaves in T such that the depth of a is at most the depth of b: $D(a) \le D(b)$. If their frequencies satisfy $f(a) \le f(b)$, then we can exchange $a \leftrightarrow b$ in the tree without increasing its cost, $COST(f,T)$. If T is optimal for f, the exchange will preserve optimality. By making two such exchanges in an initially optimal T, we produce an optimal tree with the desired "deepest pair" property. **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':

$$COST(f') = COST(f', C'). \tag{31}$$

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

$$COST(f, C) = COST(f', C') + f(b) + f(b'). \tag{32}$$

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

$$COST(f) = COST(f') + f(b) + f(b'). \tag{33}$$

[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 (31)–(33), we conclude $COST(f) = COST(f, C)$, *i.e.*, C is optimal. ■

¶23. Representation of the Code Tree for Transmission. Suppose I want to send you the string s. It is not enough to send you its code $C(s)$. I must also send you some representation of the code tree T_C. Let this representation be a binary string α_C. We will now provide one description of α_C that is essentially optimal in its length. We split the binary α_C into two parts:

$$\alpha_C = \beta_C \gamma_C \tag{34}$$

where β_C encodes the "shape" of the code tree T_C, and γ_C is just a list of the characters in Σ, in the order they appear as leaves of T_C.

We first explain γ_C. Let the code be $C : \Sigma \to \{0,1\}^*$. The alphabet Σ occurs in some larger context. This context is provided by a **standard character set** which we denote by Σ_0. For instance, Σ_0 may be the ASCII set or Unicode (see Figure 8 and notes below). *We assume that* Σ *is a subset of* Σ_0. For the present purposes, assume $\Sigma_0 \subseteq \{0,1\}^N$ for some fixed N. This N is the common knowledge used by both the transmitter and receiver. As a practical matter, it is important 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 transmitting.

E.g., Let C be Huffman code in Figure 2(b), Σ_0 be the extended ASCII set (so $N = 8$). If $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$, then the codes for these letters in hexadecimal are `a=0x61`, `b=0x62`, `c=0x63`, `d=0x64` (see Figure 8 in the next section). Therefore γ_C = `0x61626364`. In full glory, γ_C = `0110'0001'0110'0010'0110'0011'0110'0100`.

It remains to explain the string β_C in (34). We give a progression of ideas that lead to the final form. 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` (35)

$4n - 4 = 12$ bits

The '-marks here are visual decorations, 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?

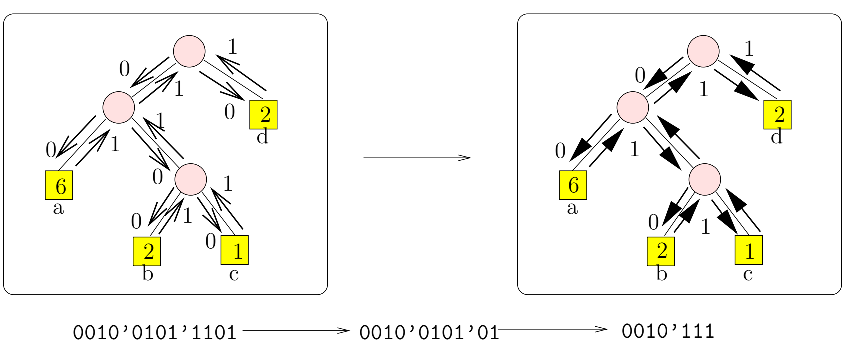


Figure 4: Compressed bit representation for the Huffman tree Figure 2a

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 have reached the end of the encoding. This property is useful for many applications; in particular, we need this property in (34): we must know when β_C ends, and γ_C begins.

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 (35) is now represented by

$$\texttt{0010'0101'01}. \tag{36}$$

$3n-2=10$ 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 (36) is now shortened to:

How about $01 \to 1$?

$$\texttt{0010'111}. \tag{37}$$

$2n-1=7$ bits

See the final illustration in Figure 4. The final scheme (37) will be known as the **compressed bit representation** β_T of a full binary tree T. In case T is the code tree for a Huffman code C, β_T is the desired β_C of (34).

If T has more than one leaf, β_T will begin with a 0 and end with a 1. Since we assume $|\Sigma| \ge 2$, the shortest bit string is 011, represent a full binary tree with two leaves. For completeness, we will define $\beta_T = 1$ for the tree T with only one leaf. The following lemma summarizes several simple properties:

Lemma 10 *Let T be a full binary tree T with $n \ge 1$ leaves, and β_T is its compressed bit representation.*
(i) The length of β_T is $2n-1$, with $n-1$ zeros and n ones.
(ii) In any proper prefix of β_T, the number of zeros is at least the number of ones.
(iii) The set $S \subseteq \{0,1\}^*$ of all such compressed bit representations β_T forms a prefix-free set.
(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 β_T for some T.

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

Theorem 11 *There is a protocol to transmit a binary string α_C representing any Huffman code $C : \Sigma \to \{0,1\}^*$ on $|\Sigma| = n$ letters such that*
(i) The length of α_C is $(2n-1) + Nn = n(N+2) - 1$.
(ii) A receiver can recover the code C from α_C in linear time, without prior knowledge of Σ except that $\Sigma \subseteq \Sigma_0 \subseteq \{0,1\}^N$.

Proof. Using the notation of (34), $\alpha_C = \beta_C \gamma_C$ where $|\beta_C| = 2n-1$ when the code tree T_C has n leaves, and $|\gamma_C| = nN$ under our assumption that $\Sigma \subseteq \Sigma_0 \subseteq \{0,1\}^N$. 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 reconstruct the shape of T_C and thus knows n. Since the receiver knows N, he can also parse each symbol of Σ in the rest of β_C. **Q.E.D.**

¶24. Canonical Huffman Trees. full binary tree is said to be **canonical** if in each level, all the leaves appear to the left of all the internal nodes. Thus the first tree in Figure 2 is

non-canonical, but the second is canonical. If a full binary tree is non-canonical, we can try to make it canonical by **swapping**: pick any two nodes u, v at the same level. We totally order the nodes in a given level using "$u < v$" if u is to the left of v. If $u < v$ where u is internal and v is external, we call (u, v) an **inverted pair**. Hence T is canonical iff there are no inverted pairs. We define the $Swap(u, v)$ operation in which the the subtrees T_u, T_v rooted at u and v are swapped. This can be implemented with two pointer assignments. If u, v are in level $\ell \geq 1$, then:
(i) The number of inverted pairs at levels $< \ell$ is unchanged.
(ii) The number of inverted pairs at level ℓ strictly decreases.
(iii) $COST(T)$ is invariant under such swaps.
By repeated swaps, we can obtain a canonical tree (Exercise). For example, the tree in Figure 2(a) is non-canonical: at level 1, the leaf `d` appears after the internal node `9`. After swapping them, the tree become canonical. Likewise, the tree in Figure 2(b) can be made canonical with one swap. The tree in Figure 3 is a canonical (despite the way we draw all leaves at the "same" level).

Canonical trees has a self-limiting encoding that uses only $n+2$ bits where n is the number of leaves (Exercise). We want to exploit this in the preceding lemma on α_C. Suppose T is Huffman tree. We note that swaps does not change the cost of the code tree (it is less clear whether swaps preserves Huffman-ness of trees). By repeated swaps, we finally reach a "canonical Huffman tree". In the exercises, we show that the encoding of canonical trees can be further improved.

TO BE COMPLETED:

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. According to Adam Gashlin[11] the encodings of canonical Huffman trees are used in Audio-Video compression applications.

¶25. Notes on Morse Code. In the Morse[12] 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.

[11] Private Communication, 2012.

[12] 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.

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

Exercises

Exercise 4.1: Let s be the following string: "`hello!␣this␣is␣my␣little␣world!`". Show the Huffman code C for s, and what is $|C(s)|$? ♢

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. Try to minimize $|\Sigma|$. ♢

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: The text proved 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. $\diamondsuit$

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 β_T where T is the full binary tree with 6 leaves and every right child is a leaf.

(c) What is β_T where T is the full binary tree with 6 leaves and every left child is a leaf.

(d) What is β_T where T is the complete binary tree with 8 leaves.

$\diamondsuit$

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? $\diamondsuit$

Exercise 4.10: We define a **ternary tree** by analogy with a binary tree: it is either the empty tree with no nodes, or it has a unique node called the root. Each non-leaf has at most three children, each with a unique label, taken from the set {L,M,R} (corresponding to Left, Middle or Right child). The tree is **full** if each non-leaf has 3 children. In the following, let T_n denote a full ternary tree with exactly $n \ge 1$ internal nodes. By definition, T_0 is the tree with only one node. Be sure to justify your answers!

(a) How many leaves does T_n have?

(b) Describe the "compressed bit representation" of T_n by analogy with the case of full binary trees. We will write $\alpha(T_n)$ for this representation.

(c) Please draw the full ternary tree represented by

`‘0011’0111’1001’1111’`

(d) How many bits are used in $\alpha(T_n)$?

(e) Let $A \subseteq \{0,1\}^*$ be the set of all compressed bit representation of full ternary trees. Characterize A. This means give a complete list of properties that determines A.

$\diamondsuit$

Exercise 4.11: Let T be a full binary tree on n leaves. Give an algorithm to convert its compressed bit representation $\beta_T[1..2n-1]$ to a $4n-4$ array $B[1..4n-4]$ representing the traversal of T. $\diamondsuit$

Exercise 4.12: 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. $\diamondsuit$

Exercise 4.13: (a) Prove (30).

(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?

$\diamondsuit$

Exercise 4.14: (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

$$1 \ge \sum_{i=1}^{n} 2^{-d_i}. \tag{38}$$

Moreover, if the binary tree is a full binary tree, then the inequality is an equality. $\diamondsuit$

Exercise 4.15: (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 $n \ge 1$, $rep_1(n) = 0^{len(n)}{}'bin(n)$. 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 $n \ge 0$. 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 $n \ge n_0$ (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 4.16: (Gashlin 2012) We give a further improvement on the encoding of canonical Huffman trees, using a radically different approach: instead of focusing on the leaves, we focus on the internal nodes. An integer sequence of the form

$$\boldsymbol{n} = (n_0, n_1, \ldots, n_{d-1}) \tag{39}$$

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 (39) that satisfies $N = \sum_{i=0}^{d-1} n_i$, $n_0 = 1$ and $1 \le n_i \le 2n_{i-1}$ $(i = 1, \ldots, d-1)$.
(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 (39) 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 $T(N)$. ♢

Exercise 4.17: 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.
```

♢

Exercise 4.18: 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

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

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

Exercise 4.19: Suppose you are given the frequencies f_i in sorted order. Show that you can construct the Huffman tree in linear time. ♢

Exercise 4.20: (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 4.21: (Properties of Optimal Ternary Huffman Code)
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 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, then both its children are leaves.

(b) Let T be an optimal 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.

♢

Exercise 4.22: (Algorithm for Optimal Ternary Huffman Code)
Ternary means that our code uses "trits" instead of bits. Prove the correctness of the following algorithm for computing the optimal ternary Huffman code algorithm. You may assume the properties proved in the previous question.

> Ternary Huffman Code Algorithm:
> Input: Frequency function $f : \Sigma \to \mathbb{N}$.
> Output: Optimal ternary code tree T^* for f.
> 1. Let Q be a priority queue containing weighted code trees.
>   Priority is determined by the weight of the root.
>   Initially, Q is the set of $n = |\Sigma|$ trivial trees,
>   each tree with one node representing a single character in Σ.
> 2. If n is even,
> $T \leftarrow Q.\texttt{deleteMin}()$, $T' \leftarrow Q.\texttt{deleteMin}()$.
> $Q.enqueue(Merge(T, T'))$
> 3. While Q has more than one tree,
>   3.1. $T \leftarrow Q.\texttt{deleteMin}()$, $T' \leftarrow Q.\texttt{deleteMin}()$ $T'' \leftarrow Q.\texttt{deleteMin}()$
>   3.2. $Q.enqueue(Merge(T, T', T''))$.
> 4. Now Q has only one tree T^*. Output T^*.

◇

Exercise 4.23: Given the optimal ternary Huffman code C for strings s below. In each case, determine $|C(s)|$.
(a) $s =$ `hello␣world!`
(b) $s =$ `hi!␣my␣little␣world!`. ◇

Exercise 4.24: We want to compare the relative efficiency of bits versus trits in Huffman coding. Let $f : \Sigma \to \mathbb{N}$ be a frequency function. Suppose $H_2(f)$ is cost of f under the standard (binary) Huffman code; let $H_3(f)$ be the cost using a ternary Huffman code. We say "bits are better than trits" on f if $H_2(f) < H_3(f)\lg(3)$ (and conversely if the inequality goes the other way). Using the algorithm in the previous question to compute $H_3(f)$, answer the question whether "bits are better than trits" for the following f's:
(a) Let $f(a) = f(b) = f(c) = 1$ (and f is zero on other characters).
(b) Let f be the frequency function for `compress␣this␣please`. ◇

Exercise 4.25: 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 $n \ge 1$ 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 $3N_2 + 2N_3 \le 4$, 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 $N'_2 + N'_3 \le 1$. 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 4.26: Further generalize the 3-ary Huffman tree construction to arbitrary k-ary codes for $k \geq 4$. ◇

Exercise 4.27: 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 $f(c) = 6$.

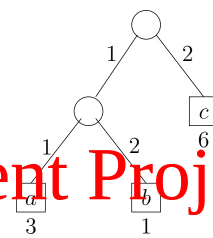


Figure 5: A skew Huffman tree with skew cost of 21.

(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.

◇

Exercise 4.28: (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. ◇

Exercise 4.29: (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. ◇

Exercise 4.30: 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? ◇

Exercise 4.31: (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. ♢

End Exercises

§5. Dynamic Huffman Code

The original Huffman coding formulation does not take into account the full context of its applications, which we now clarify.

¶26. The Larger Context of Huffman Coding. Here is the typical sequence of steps for compressing and transmitting a string s using the Huffman code algorithm. Imagine two 2 players in this process, called the Sender and Receiver.

Sender Steps:

(S1) To transmit string s, first compute its frequency function f_s.

(S2) Using f_s, construct a Huffman code tree T_C corresponding to the code C.

(S3) Using T_C, compute the compressed string $C(s)$.

(S4) Finally, transmit the string $\alpha_C; C(s)$ that is the concatenation of α_C (representing T_C as in Theorem 11), with the compressing string $C(s)$.

Receiver Steps:

(R1) On receipt of α_C, reconstruct T_C. Since α_C is self-limiting, the Receiver knows the break between α_C and $C(s)$.

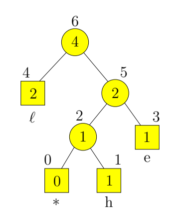
(R2) Using T_C, the Receiver can now reconstruct s from $C(s)$

Since the Sender must make two passes over the string s (in steps (S1) and (S3)), the original Huffman tree approach is called "2-pass Huffman encoding". There are weaknesses in this 2-pass process: (a) Multiple passes over the string s makes the algorithm unsuitable for realtime data transmissions. If s represents a large file, this require extra buffer space. It is unsuitable for strings that are (potentially) infinite. Examples of the latter include Ticker Tape data from a Stock Exchange, or sensor data that is continuously transmitted from satellites to earth stations. (b) The Huffman code tree must be explicitly transmitted before the decoding of $C(s)$ can begin. The sender needs an algorithm to convert T_C to α_C, and the receiver needs another algorithm to do the reverse conversion.

An approach called **Dynamic Huffman coding** (or adaptive Huffman coding) can overcome these weaknesses: the Sender does not need to explicitly transmit the code tree, and makes only one pass over the string s. In fact, the Sender does not even have to pass over the entire string even once before starting transmission; this allows the Receiver to begin decoding some prefix of s while the transmission is ongoing. This property is vital for the transmission of (potentially) infinite strings as noted above. There are two algorithms for dynamic Huffman coding: one is the **FGK Algorithm** (Faller 1973, Gallager 1978, Knuth 1985) and the **Lambda Algorithm** (Vitter 1987). See [11]. The dynamic Huffman code algorithm has been used [13] for data compression in the Unix utility called `compress/uncompress`.

¶27. 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 non-decreasing with rank) If w_i is the weight of node with rank i, then $w_{i-1} \leq w_i$ for $i = 1, \ldots, 2k$.

(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 12 *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.**

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

¶28. 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 $k+1 \geq 1$ 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

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

of length $2k+1$ where $\texttt{Wt}[i]$ is the *weight* of node i, and $\texttt{Lc}[i]$ is an even integer indicating the *left child* of node i. In case node i is a leaf, we may let $\texttt{Lc}[i] = -1$. Alternatively, as will be done in these notes, we let $\texttt{Lc}[i]$ 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 $\texttt{Lc}[i]+1$, and $\texttt{Lc}[i]$ 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 Lc is not essential, but it makes the examples more compact. 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. Because of the dynamic nature of Σ, we may encounter $x \notin \Sigma$. We assume that all letters will belong to a larger set Σ_0 that contains Σ. Hence define

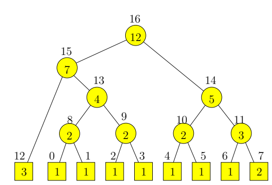
$$\texttt{Cm} : \Sigma_0 \to \{-1, 0, 1, 2, \ldots, 2k\} \tag{40}$$

such that $\texttt{Cm}[x] = i$ iff $\texttt{Lc}[i] = x \in \Sigma$, and $\texttt{Cm}[x] = -1$ if $x \notin \Sigma$. Call Cm the **character map array**. Initially, $\texttt{Cm}[x] = -1$ 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 entry in Cm updated.

In summary, our Huffman tree is represented by three arrays Lc, Wt, Cm. For example, the Huffman tree in Figure 3 is[14] illustrated by the arrays in Table 2: Two of these arrays, Lc and

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



Wt, are explicitly shown. But the array $\texttt{Cm}[x \in \Sigma_0]$ is easily inferred from the leaf entries of Lc. E.g., $\texttt{Cm}[h] = 0$, $\texttt{Cm}[e] = 1$ and $\texttt{Cm}[a] = -1$. There is no "Rank" array in this representation because, trivially, $Rank[v] = v$ for all $v \in \{0, \ldots, 2k\}$.

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 $u = \texttt{Cm}[x]$. 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

[14] 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. Check: all left-child $\texttt{Lc}[u]$ have even ranks like 14.

$C(x)$:
    $u \leftarrow \texttt{Cm}[x]$
    Output(parity(u))
    While $u < 2k$
        $u \leftarrow parent(u)$
        Output(parity(u))

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

$parent(u)$:
    $\ell \leftarrow 2\lfloor u/2 \rfloor$
    for $p \leftarrow u+1$ to $2k$
        if $(\texttt{Lc}[p] = \ell)$, 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).

¶29. 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'.*

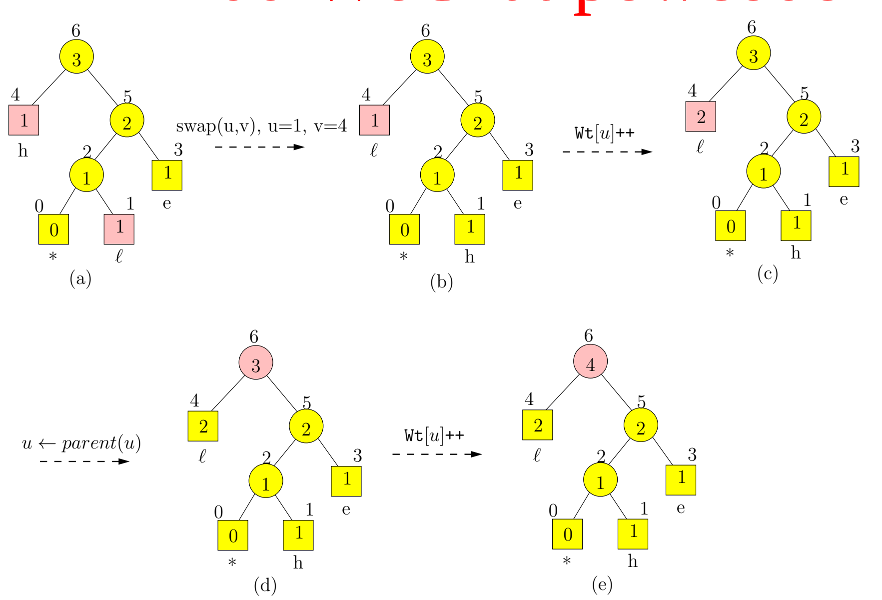


Figure 6: Restoring Huffmanness after incrementing the frequency of letter ℓ

Let us first give some intuition of what has to be done, using our example of **he$\ell\ell$o worℓd!**. Begin with the Huffman tree after having transmitted the prefix **heℓ**. 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 represents ℓ) to $2 = 1+1$, we would violate the ranking property (S1) of ¶27. 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 node v of largest rank $R(v)$ subject*
  ▷ *to $\mathtt{Wt}[v] = \mathtt{Wt}[u]$. 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 $\mathtt{Lc}$ array, we only have to exchange the values in the array entries $\mathtt{Lc}[u]$ and $\mathtt{Lc}[v]$. But note that this swap may involve leaves, in which case we have to update the character map $\mathtt{Cm}$:

$Swap(u, v)$
1. $tmp \leftarrow \mathtt{Lc}[u]$
2. $\mathtt{Lc}[u] \leftarrow \mathtt{Lc}[v]$
3. $\mathtt{Lc}[v] \leftarrow tmp$
4. If $(\mathtt{Lc}[u] \in \Sigma_0)$ then $\mathtt{Cm}[\mathtt{Lc}[u]] \leftarrow u$
5. If $(\mathtt{Lc}[v] \in \Sigma_0)$ then $\mathtt{Cm}[\mathtt{Lc}[v]] \leftarrow v$

The first 3 assignments in $Swap(u, v)$ is the standard meaning of swapping two values: $\texttt{Lc}[u] \leftrightarrow \texttt{Lc}[v]$. The last two assignments are to update our character map `Cm` for the purposes of We do not need to exchange $\texttt{Wt}[u]$ and $\texttt{Wt}[v]$ since they have the same weights! A swap is done only if $v > u$ (Line 2 of Restore). Thus the rank of the current node u is strictly increased by such swaps. After swapping u and v, their siblings are automatically swapped (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 `Lc`, `Wt` arrays. Suppose we have just completely processed our famous string "$\texttt{he}\ell\ell\texttt{o world!}$", and the resulting Huffman tree T is shown in 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 so we call Restore(u). The state of the `Lc`, `Wt` arrays just before this restore is taken from Table 2 above:

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
Restore (u)			u														

Note that u has rank 2. For clarity, let x_i denote the node whose rank is i. Thus $u = v_2$. Moreover, $\texttt{Wt}[x_2] = 1$ and so we must find the largest ranked node with the same weight of 1. As seen from the table, this is the node x_6. In the restore algorithm, after the while loop, the variable v is equal to x_6, and we call $Swap(u, v) = Swap(x_2, x_6)$. After the swap, $v = x_2$ and $u = x_6$, and we incrementing the weight of x_6:

`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	14
`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	<u>11</u>	<u>12</u>	13	14	<u>15</u>	<u>16</u>
`Lc`	h	e	!	w	r	d	⊔	o	0	2	4	ℓ	6	8	10	12	14
`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	⊔	o	0	2	4	ℓ	6	8	10	12	14
`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	⊔	o	0	2	4	ℓ	6	8	10	12	14
`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`																	$u = v$

¶30. How to add a new letter: the 0-Node. Our dynamic Huffman code tree T must be capable of expanding its alphabet Σ. This is illustrated in Figure 7 which shows how T evolves as we process successive letters in the string $s =$ `hello`.

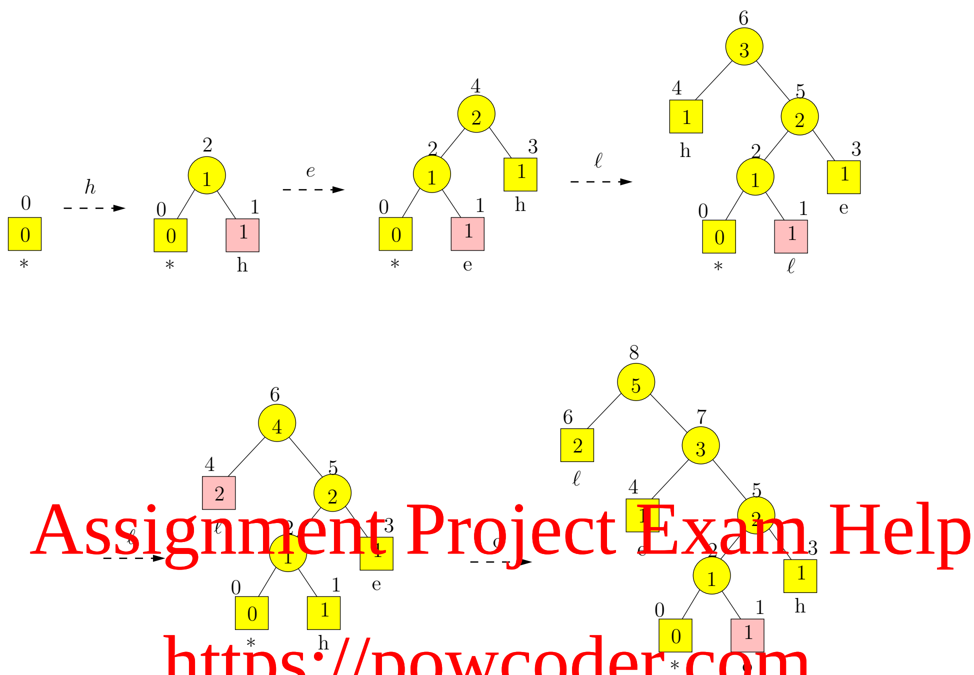


Figure 7: Evolving Huffman tree on inserting the string `hello`

Initially, $\Sigma = \{*\}$ where $*$ is a special letter of weight 0. The leaf of T representing $*$ is called the 0**-node** (or, the $*$-node). It is justified by the fact that no other node has weight 0, and $*$ does not occur in the input string s. Although this node does not represent any input letter, yet in another sense, it represents all the yet unseen letters! We slightly revise our definition of `Cm` in (40) to

$$\texttt{Cm} : \Sigma_0 \to \{0, 1, 2, \ldots, 2k\} \tag{41}$$

such that $\texttt{Cm}[x] = 0$ iff $x \notin \Sigma$, and otherwise $\texttt{Lc}[\texttt{Cm}[x]] = x$.

By the Sibling Property, the 0-node has rank 0. If T has more than 1 leaf, then by the Sibling Property, the 0-node has a sibling u of rank 1. Note that u is necessarily a leaf. Let p be the parent of u and the 0-node. Then the weight of p and u are equal. This is an anomaly: in no other situation can a parent and child have the same weight. Although the Sibling Property does not require it, it is easy to see that we may assume that p has rank 2. Summarizing:

1. Our Huffman tree T has a special 0-node.
2. If T has other nodes, then the parent of the 0-node has rank 2.

(42)

Let us now see how T evolves as we process the next letter x in the string s. There are two cases:

(Case-1) The letter x is already in T: In this case, the Character Map $\texttt{Cm}[x]$ returns the leaf that represents x. We can output the code $C(x)$ of x starting from $\texttt{Cm}[x]$, as described above. Moreover, we update T by increment the weight of $\texttt{Cm}[x]$, and call Restore(u) where u is the parent of $\texttt{Cm}[x]$.

(Case-2) The letter x is new: Now, we assume that $\mathtt{Cm}[x]$ returns the 0-node. Again we can output $C(*)$, and send the standard code for x (e.g., ASCII code of x). Let u be the 0-node. We now turn u into an internal node with children u_L and u_R. Make u_L the new 0-node, and u_R represent x with weight 1. That means[15] updating the Character Map with $\mathtt{Cm}[*] \leftarrow u_L$ and $\mathtt{Cm}[x] \leftarrow u_R$. We update the weights as follows: $\mathtt{Wt}[u_L] \leftarrow 0, \mathtt{Wt}[u_R] \leftarrow 1$. Note that $\mathtt{Wt}[u] = 0$, but must be incremented: this is done by calling Restore(u), which ensures that the update is propagated all the way to the root while preserving the Sibling Property.

The transformation of the arrays `Lc`/`Wt` in Case-2 seems formidable as described above. We will eventually see that the mechanics is quite simple. But to understand what needs to be done, we study an example. A Case-2 transition in Figure 7 happens when we process the letter `o` in the string `hello`. The `Lc`/`Wt` arrays before and after this transition is shown here:

Before	Rank	–	–	0	1	2	3	4	5	6
	`Lc`	–	–	`*`	`h`	0	`e`	ℓ	2	4
	`Wt`	–	–	0	1	1	1	2	2	4
After	Rank	0	1	2	3	4	5	6	7	8
	`Lc`	`*`	`o`	0	`h`	2	`e`	ℓ	4	6
	`Wt`	0	1	0	1	1	1	2	2	4
Restored	Rank	0	1	2	3	4	5	6	7	8
	`Lc`					`e`	2			
	`Wt`			1			2		3	5

Conceptually, we must add two new array entries *in front of the original arrays!* Under the assumption (42), the former 0-node becomes the rank 2 node. Indeed, all the nodes of the original tree have their ranks increased by 2. That meant for any internal node of rank i, if $\mathtt{Lc}[i] = j$ before the transition, then $\mathtt{Lc}[i+2] = j+2$ after. E.g., if $\mathtt{Lc}[6] = 4$ before, then $\mathtt{Lc}[8] = 6$ after. This appears to be a non-trivial transformation, but we will solve it by an encoding trick. We use a blue font to identify those entries of `Lc`/`Wt` array that have been changed. These transformations (Before $\rightarrow$ After) are easily achieved using extendible arrays, to be described.

In the After arrays, there is a single entry in red font, namely $\mathtt{Wt}[2] = 0$. This is the entry that needs to be incremented to $\mathtt{Wt}[2] = 1$. We call Restore(2) to do this increment since it must be propagated along the root path. The final result is shown in the arrays for Restored where we only indicate entries that have changed. Note that we have effected one swap operation, Swap$(4,5)$=Swap$(4, \mathtt{Cm}[\mathtt{e}])$.

We address the final issue of how to increase the ranks of all nodes by 2 in the array `Lc` (i.e., transforming $\mathtt{Lc}[i] = j$ to $\mathtt{Lc}[i+2] = j+2$ above). There are two related issues:

(a) First, we need **extendible arrays**. Recall that an array A in conventional programming languages has a fixed length n, say $A = A[0..n-1]$. But in most applications, we may assume that the length of A can be extended by any needed amount $m \geq 1$, transforming

[15] As we shall see, $\mathtt{Cm}[*]$ is always 0, so this never needed updating. Also $\mathtt{Cm}[x]$ can always be set to 1. Identifying nodes with their ranks, we may write $u = 2, u_R = 1, u_L = 0$.

$A[0..n-1]$ to $A[0..n+m-1]$ in which $A[n..n+m-1]$ *are new slots while* $A[0..n-1]$ *are unchanged.* Such an extendible array A^* can be simulated by a fixed length array by pre-allocating a sufficiently large array, provided we introduce a member variable n to remember the *current length* of the array. Extending the array by m slots just amounts to increasing n by m. Thus we will simulate $\texttt{Lc}/\texttt{Wt}$ by the extendible arrays $\texttt{Lc}^*/\texttt{Wt}^*$.

(b) The second issue is that the array A^* is extendible at the *high end* (i.e., the new m slots are $A[n..n+m-1]$) while our application needs to extend at the *low end* (i.e., new slots in $A[0..m-1]$). We partially resolve this by defining this relationship between A and A^*:

$$A[i] = j \text{ iff } A^*[\texttt{Idx}(i)] = j \tag{43}$$

where $\texttt{Idx}(i) = n-1-i$ is the **index function**. But for the array $\texttt{Lc}$, the value of $\texttt{Lc}[i]$ (when i is an internal node) is actually, an index into the array $\texttt{Lc}$. Therefore the relationship between $\texttt{Lc}$ and $\texttt{Lc}^*$ is modified to

$$\texttt{Lc}[i] = j \text{ iff } \texttt{Lc}^*[\texttt{Idx}(i)] = \texttt{Idx}(j). \tag{44}$$

Since the size of $\texttt{Lc}/\texttt{Wt}$ is $n = 2k+1$ ($k = |\Sigma|$), it follows that $\texttt{Idx}(i) = 2k - i$ in our application. E.g., if i is (the rank of) an internal node, then (the ranks of) its left and right children are

$$\texttt{Lc}[i] = \texttt{Lc}^*[\texttt{Idx}(i)], \quad \texttt{Lc}[i]+1 = \texttt{Lc}^*[\texttt{Idx}(i)] - 1.$$

Note that the root always has rank $2k$ in $\texttt{Lc}/\texttt{Wt}$, it follows that $\texttt{Lc}^*[\texttt{Idx}(2k)] = \texttt{Lc}^*[0]$ is always the left child of the root, independent of the size $n = 2k+1$. Similar remarks apply to the array $\texttt{Wt}^*$: e.g., $\texttt{Wt}[i] = \texttt{Wt}^*[\texttt{Idx}(i)]$, and therefore $\texttt{Wt}^*[0]$ is the weight of the root.

The implementation of (a) and (b) results in the following Before→After transformation of the array $\texttt{Lc}^*$ of our running example. We also show $\texttt{Lc}$ for comparison:

Before

Rank i	–	–	0	1	2	3	4	5	6
$\texttt{Lc}[i]$	–	–	*	h	0	e	ℓ	2	4
$\texttt{Idx}(i)$	0	1	2	3	4	5	6	–	–
$\texttt{Lc}^*[\texttt{Idx}(i)]$	2	4	ℓ	e	6	h	*	–	–

After

Rank i	0	1	2	3	4	5	6	7	8
$\texttt{Lc}[i]$	*	o	0	h	2	e	ℓ	4	6
$\texttt{Idx}(i)$	0	1	2	3	4	5	6	7	8
$\texttt{Lc}^*[\texttt{Idx}(i)]$	2	4	ℓ	e	6	h	8	o	*

It is clear the transformation affects only the last 3 entries of $\texttt{Lc}^*$ (in red font).

In summary: all operations on the arrays $\texttt{Lc}/\texttt{Wt}$ are effected through operations on the extendible arrays $\texttt{Lc}^*/\texttt{Wt}^*$. In particular, subroutines such as Restore must be reinterpreted as operations on $\texttt{Lc}^*/\texttt{Wt}^*$. For Case-2, we extend these arrays simply by incrementing k, effectively extending $\texttt{Lc}^*/\texttt{Wt}^*$ by 2 slots. These new slots are at the high end of $\texttt{Lc}^*/\texttt{Wt}^*$ (corresponding to the low end of $\texttt{Lc}/\texttt{Wt}$). The transformation can be effected in $O(1)$ time.

¶31. Interface between Huffman Code and Standard 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 **standard encoding**. The main example of a standard encoding U is the set of ASCII characters with $N = 8$. A more complicated example is when U is some unicode set. We assume the transmitter and receiver both know this standard encoding (in particular the parameter N). In the encoding process, we assume that $s \in U^*$. Upon seeing a letter x in s, we must decide whether $x \in \Sigma$ (i.e., in our current Huffman tree), and if so, transmit its current Huffman code. To be specific, suppose $|\Sigma| = k$ and the current Huffman tree T is represented by the arrays $\mathtt{Lc}[0..2k], \mathtt{Wt}[0..2k]$. 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.

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 $\mathtt{Lc}, \mathtt{Wt}$ 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 the 0-node, and size $k \leftarrow 0$.
    (T is represented by the arrays $\mathtt{Lc}^*/\mathtt{Wt}^*/\mathtt{Cm}$)
▷ *Main Loop*

  While s is non-empty
1. Remove letter x from the front of string s.
2. Let $u = \mathtt{Cm}[x]$ be the leaf of T corresponding to x.
3. Using u, transmit the code word $C(x)$.
4. If u is the 0-node ◁ *x is a new character*
5. Transmit the standard code for x.
6. k++ and perform the Before → After transformation.
7. Call Restore(2).

  Signal termination, using some convention.

On the Receiver End, decoding is relatively straightforward. We are processing a continuous binary sequence, but we know where the implicit "breaks" are in this continuous sequence because the transmitted information is a sequence of self-limiting codes. 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 w, there is a corresponding action:

(R1) If $w = C(x)$ is the code word for some $x \in \Sigma$, we spit out the standard code of x. We also increment the weight of $\mathtt{Cm}[x]$ by calling Restore($\mathtt{Cm}[x]$).

(R2) If $w = C(*)$, this is is a signal that the next word is the standard encoding of a new letter.

(R3) If w is the anticipated standard encoding of a new letter x, we spit out w. Then we update $\mathtt{Cm}[x] = 1$ and perform the <u>Before</u> → <u>After</u> transformations of $\mathtt{Lc}^*/\mathtt{Wt}^*$. Finally, we call Restore(2).

What is spit out by the receiver is clearly a faithful reproduction of the original string s which is in standard encoding.

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].

¶32. 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 [5].

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	[illegible]
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		[illegible]	[illegible]	±	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

¶33. 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. There are 95 **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), 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 hexadecimal code for any character in Figure 8 is given as $0xCR$ where C is the column number and R is the row number of that character. E.g., $0x20$ is the space character (also denoted ␣), and the lower case alphabet `a, b, c,..., x, y, z` occupy the sequence $0x61$ to $0x7A$.

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(`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(`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.

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

Note that ␣ (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(§) = $0xA7$ and ASCII(¶) = $0xB6$. Actually, not all of the available codes in the extension has been assigned to characters. A recent assignment is $ASCII$(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.*.

¶34. 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 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 tree used for encoding in part(a)? Note: using static Huffman coding, you must first transmit part(b) before transmitting part(a).
(c) How many bits would be transmitted by the Dynamic Huffman code algorithm in sending the string "`hello, world!`"? Compare this number with parts(a)+(b). $\diamondsuit$

Exercise 5.2: Simplified[16] dynamic Huffman tree.
Let T be the external BST in Figure 9 whose leaves store the set $\Sigma = \{*, \mathtt{e}, \mathtt{h}, \mathtt{l}, \mathtt{o}, \mathtt{r}, \mathtt{w}\}$ of characters with the sorting order

$$* < \mathtt{e} < \mathtt{h} < \mathtt{l} < \mathtt{o} < \mathtt{r} < \mathtt{w}.$$

Given $x \in \Sigma$, we can compute its code $C(x)$ by doing Lookup(x) in T, and $C(x)$ is just the corresponding search path. E.g., $C(\mathtt{h}) = 0100$, and $C(\mathtt{w}) = 11$. Thus, T represents a prefix-free code for Σ.

Let $s \in \Sigma^*$ be a string to be transmitted. Assuming that receiver also has a copy of the tree T, the receiver would be able to decode our message. We look at two scenarios: a static and a dynamic way to use T for transmission.

1. We first assume that T is static (unchanging). How many bits does it take to transmit $C(s)$ where
$$s = \mathtt{hellloooo} \tag{45}$$
Please show the encoded string $C(s)$.

[16] This is simplified because we do not do insertion and deletion from T.

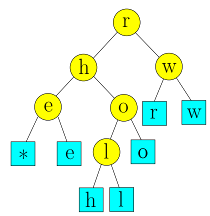


Figure 9: An external BST

2. Next, assume that T is dynamic. After each Lookup(x), you will immediately transmit the code $C(x)$. *But you further splay the parent of* x. Hence the next time you do Lookup(x), the code $C(x)$ may be different! Note that the receiver can do the same actions. How many bits does it take to transmit the string (45) of part(a)? Again, show us the encoded binary string $C(s)$.

$\diamondsuit$

Exercise 5.3: What is the message s?

(a) Consider the following compressed bit representation α_T of a full binary tree T:

$$\alpha_T = 0001'1011'0011'0011'011. \tag{46}$$

Please draw T.

(b) The leaves of T contain the following ASCII characters, listed in left-to-right order:

⊔ o	g e d	r i s n t

Parts(a) and (b) defines a Huffman code. We used it to encode a string s into $C(s)$:

$$C(s) = 0101'0101'1011'1000'0011'0011'0100'0111'0001'1111'0000'1000'1001'100 \tag{47}$$

What is our original string s?

(c) What is the total number of bits I need to transmit in order to send you this message, including the Huffman tree? Is this an improvement over just sending you the straight ASCII code for s? $\diamondsuit$

Exercise 5.4: 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. $\diamondsuit$

Exercise 5.5: Natural languages are highly redundant[17]. 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.

[17] For good reasons! Never complain about their ambiguity or redundancy – it is their secret power.

```
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 5.6:
(a) Please reconstruct the Huffman code tree T from the following representation:

$$r(T) = 0000'1111'0011'011d'mrit'yo$$

Besides the 0/1 symbols, the letters d, m, i, etc, stand 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 an ASCII string s encoded using this Huffman code:

$$C(s) = 0001'1110'1001'1001'0111'1011'10$$

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

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

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

Exercise 5.8: 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 5.9: 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 5.10: Consider 3-ary Huffman tree code. State and prove the Sibling property for this code. ♢

Exercise 5.11: 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 5.12: 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 5.13: 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? ♢

Exercise 5.14: Programming Project. Suppose we are given s_1, a string of (extended) ASCII characters. Let C_1 be the static Huffman code for s_1. Consider the binary string

$$\alpha_{C_1} C_1(s_1)$$

where the first part α_{C_1} is the encoding of C_1 described in ¶23 above, and the second part $C_1(s_1)$ is just the application of C_1 to s_1. Break this up into 8-bit blocks and so reinterpret it as an ASCII string denoted s_2. If necessary, we pad this string with 0's so that the last block still has 8 bits. This transformation $s_1 \to s_2$ can be repeated: $s_2 \to s_3$, etc. What is the limit of this process? ♢

Exercise 5.15: Programming Project. Compare the use of a fixed Huffman code of a large document versus a dynamic Huffman encoding. For the fixed Huffman code, use the following statistics[18] on the frequency English letters based on a sample of 40,000 words:

Letter:	E	T	A	O	I	N	S	R	H	D	L	U	C
Count:	21912	16587	14810	14003	13318	12666	11450	10977	10795	7874	7253	5246	4943
Frequency:	12.02	9.10	8.12	7.68	7.31	6.95	6.28	6.02	5.92	4.32	3.98	2.88	2.71

Letter:	M	F	Y	W	G	P	B	V	K	X	Q	J	Z
Count:	4761	4200	3853	3819	3693	3316	2715	2019	1257	315	205	188	128
Frequency:	2.61	2.30	2.11	2.09	2.03	1.82	1.49	1.11	0.69	0.17	0.11	0.10	0.07

♢

End Exercises

§6. Minimum Spanning Tree

[18] Source: Math Explorer's Club, Cornell Math Department.

¶35. Minimizing over Maximal Sets. In the **minimum spanning forest problem** we are given a costed bigraph

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

where $C : E \to \mathbb{R}$ is the cost function. 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 this standard simplification:

$$\boxed{\textit{The input bigraph } G \textit{ is connected.}}$$

With this assumption, a spanning forest 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.

An important point to note is that we are *minimizing* over certain *maximal sets* (spanning trees). In general, "min-max problems" can have very high complexity. But for MST, we are saved by the fact that these maximal sets have a great deal of structure. One formalization of this structure is the concept of matroid in the next section.

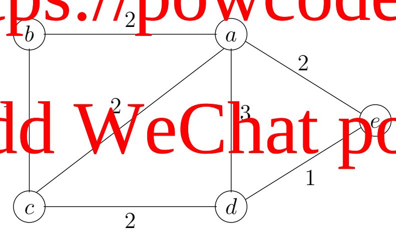


Figure 10: Bigraph G_5 with edge costs.

Consider the bigraph G_5 in Figure 10 with vertices $V = \{a, b, c, d, e\}$. One such MST is $\{a - b, b - c, c - d, d - e\}$, with cost 6 and shown in Figure 11(a).

¶36. Some Global Properties of MSTs. Fix a connected bigraph $G = (V, E; C)$ and let $MST(G)$ denote the set of all MST's of G. If all the edges of G have unit cost, then $MST(G)$ is just the set of all spanning trees of G.

For the bigraph G_5 of Figure 10, we may verify that $MST(G_5)$ consists of 5 MST's as shown in Figure 11.

Each $T \in MST(G)$ has size $n - 1$ where $n = |V|$. If $T, T' \in MST(G)$, notice that their symmetric difference $T \oplus T' = (T \setminus T) \cup (T' \setminus T)$ has an even cardinality since T and T' have the same size implies $|T \setminus T'| = |T' \setminus T|$. Thus $|T \oplus T'| = 2k$ for some $k = 0, \ldots, n - 1$. If $|T \oplus T'| = 2$, we call (T, T') an **exchange pair**. Exchange pairs have these properties:

- $|T \cap T'| = n - 2$

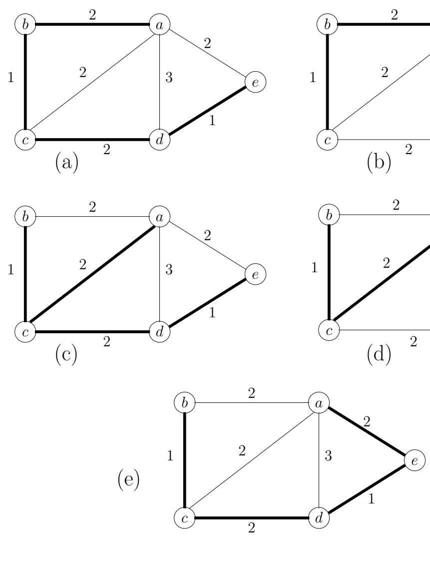


Figure 11: The 5 MST's of G_5.

- There are edges $e \in T \setminus T'$ and $e' \in T' \setminus T$ with $C(e) = C(e')$
- $T = T' - e' + e$ and $T' = T - e + e'$

In Figure 11, the pair $(T_{(a)}, T_{(b)})$ is an exchange pair, but $(T_{(a)}, T_{(d)})$ is not an exchange pair.

Define the **(MST) exchange graph** $Exch(G)$ to be the bigraph $(MST(G), H)$ whose nodes are MST's and edges $T - T' \in H$ are the exchange pairs. In Figure 12 we see $Exch(G_5)$ with 8 exchange pairs in blue.

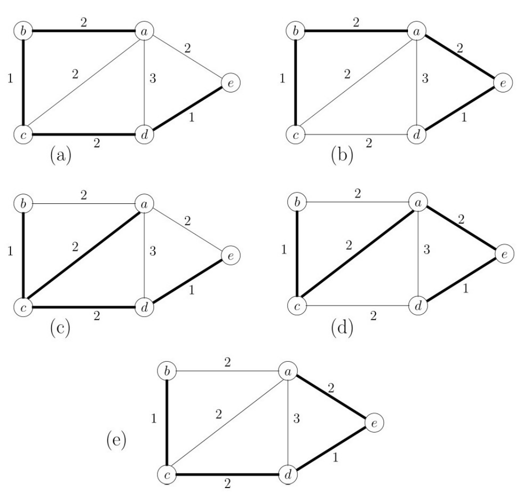


Figure 10: MST's of a bigraph.

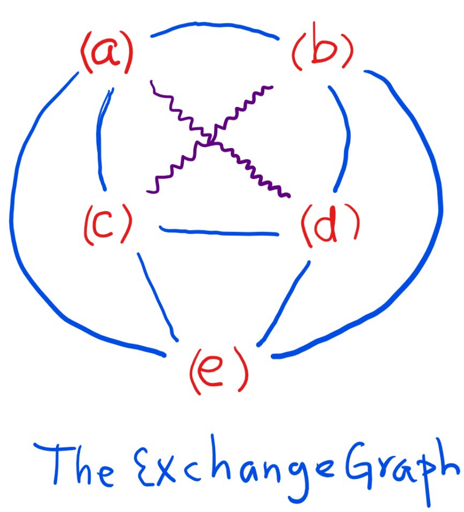


Figure 12: Exchange graph of G_5

Lemma 13 (Exchange) *If $T \neq T' \in MST(G)$, then there exists $e \in T \setminus T'$ and $e' \in T' \setminus T$*

such that $T + e' - e \in MST(G)$.

Proof. Pick $e' = \mathrm{argmin}_{e \in T' \setminus T} C(e)$. Then $T + e'$ has a unique cycle Z. In this cycle, there exists some $e \in T \setminus T'$ (otherwise $Z \subseteq T'$, contradiction). Now $T - e + e'$ is a tree. Also $C(e) \le C(e')$ since T is MST. If $C(e) = C(e')$ then $T - e + e'$ is our desired MST, proving our claim.

Otherwise, $C(e) < C(e')$ and we obtain a contradiction: by the same argument, $T' + e$ has a unique cycle Z' and there exists in this cycle some $e'' \in T' \setminus T$. Also $C(e'') \le C(e)$ since T' is a MST. This implies $C(e'') \le C(e) < C(e')$, contradicting our initial choice of e' as an argmin over $T' \setminus T$. **Q.E.D.**

We leave this as an exercise:

Theorem 14
(a) For $T, T' \in MST(G)$, *if* $|T \oplus T'| = 2k$, *then the link-distance between* T, T' *in* $Exch(G)$ *is equal* k.
(b) $Exch(G)$ *is a connected graph of diameter at most* $n - 1$ *where* $n = |V|$.

If T is a set of edges, let $C^*(T) = \{C(e) : e \in T\}$ viewed as a multiset. For instance, if T is an MST of G_5 (Figure 11) then $C^*(T) = \{1, 1, 2, 2\}$.

Theorem 15 *For all* $T, T' \in MST(G)$, $C^*(T) = C^*(T')$.

¶37. Prim's Algorithm. Let us see how one algorithm for computing the MST operates. The input is the costed bigraph $G = (V, E; C)$, and the final output is a set $T \subseteq E$ that forms a MST of G. The goal of this section is to understand the basic mechanics of Prim's algorithm, not to implement it on a computer. After this understanding you should be able to do hand-simulation.

Imagine that Prim's algorithm is trying to grow two sets:

- $S \subseteq V$ corresponding to the "settled nodes"
- $T \subseteq E$ corresponding to a minimum spanning tree for $G|S$.

We maintain an array $m[v \in V]$ with this property: for all $u \in V \setminus S$,

$$m[u] := \min\{C(v - u) : v \in S\} \tag{48}$$

with the usual convention that minimizing over an empty set of numbers is infinity: $\min \emptyset = \infty$. We proceed in **stages**: In Stage 0, we initialize $S \leftarrow \emptyset$ and

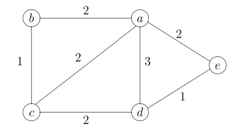
$$m[v] = \begin{cases} 0 & \text{if } v = s_0 \\ \infty & \text{else} \end{cases}$$

for some arbitrary $s_0 \in V$. At subsequent Stage i $(i = 1, \ldots, n)$ we choose any

$$u_i = \text{argmin}\,\{m[u] : u \in V \setminus S\}$$

and add u_i to S. Thus u_i is henceforth a "known" node. Since S is updated, we also need to update m following the definition (48). Here is a simulation of Prim's algorithm on our graph Figure 10:

Stage	a	b	c	d	e	$u : S \leftarrow S + u$	$e : T \leftarrow T + e$	Cost
0	0/–	∞	∞	∞	∞	$S \leftarrow \emptyset$	$T \leftarrow \emptyset$	0
1	0/–	2/a	2/a	3/a	2/a	$u \leftarrow a$	–	0
2		2/a	1/b			$u \leftarrow b$	$a - b$	2
3			1/b	2/c		$u \leftarrow c$	$b - c$	3
4				2/c	1/d	$u \leftarrow d$	$c - d$	5
5					1/d	$u \leftarrow e$	$d - e$	6



Graph G_5

Here is how to read the tabular simulation:

1. Each node $v \in V$ has a column in this table which we denote by *Column*(v).
2. Each stage i has a row in this table which we denote by *Row*(i).
3. Every entry in *Row*(i) and *Column*(v) has the form x/u where x is a number and u is a node. This tells us that "*current value of* $m[v]$ *is* x *where* $x = C(u - v)$".
4. A red entry x/u in *Row*(i) and *Column*(v) tells us "*node v is added to S in the ith stage, and the edge $u - v$ to the minimum spanning tree T* ".
5. An entry is blank if it is unchanged from the previous stage.

¶38. Generic MST Algorithm. Prim's algorithm is one of many known algorithms for computing the MST. We now want to describe a "Generic MST Algorithm" that captures many of these algorithms. Our generic algorithm is trying to grow a set T of edges. Initially, T is empty, and the algorithm stops when $|T| = |V| - 1$: this T is output as an MST. Any subset $T \subseteq E$ that is a subset of some MST is said to be **feasible**. It follows that our set T must be feasible at every step.

Let $S = V(T)$ be the vertices that appear in T. As T grows, so that S. Sometimes (e.g., in Prim's algorithm), we can also formulate the algorithm as trying grow the set S. The algorithm stops when $S = V$.

Generic Greedy MST Algorithm
Input: $G = (V, E; C)$ a connected bigraph with edge costs.
Output: $T \subseteq E$, a MST for G.
  $T \leftarrow \emptyset$.
  for $i = 1$ to $|V| - 1$ do
    1. Greedy Step: find an $e \in E \setminus T$ that is "good for T".
    2. $T \leftarrow T + e$.
  Output T as the minimum spanning tree.

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

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

¶39. Some Greedy MST Criteria. Let us say that e is a **candidate** for T if $T+e$ is acyclic. Consider the graph $G|T = (V, T)$, the restriction of G to T. The connected components of $G|T$ are called T**-components**. For instance if T is the empty set, then each vertex forms its own T-component (such components are **trivial**). For any candidate $e = u - v$, let the T-component that contains u and v (resp.) be denoted C_u and C_v. Clearly $C_u \cap C_v$ is empty (otherwise we get a cycle). Moreover, $C_u \cup C_v$ is a $(T+e)$-component. We say that e **extends** the component C_u (and also C_v by symmetry). Among the candidates for T, only some are "good". Here are 4 notions of what "good" means:

- (Minimal) $T+e$ is feasible. This is clearly the minimal condition expected of e.
- (Kruskal) Edge e has the least cost among all the candidates.
- (Boruvka) If $e = (u - v)$ then either (i) e has the least cost among all the candidates that extend the component C_u, or (ii) e has the least cost among all the candidates that extend the component C_v. In case (i), we call C_u a **Boruvka witness** for e; similarly for C_v in case (ii). It is possible that both C_u and C_v are witnesses for e.
- (Prim) This can be viewed adding an extra requirement to Boruvka's condition: assume we are given a vertex $s \in V$ called the source. The edge e must have least cost among candidates that extends the unique component $U_s \subseteq V$ containing s.

A set $T \subseteq E$ that may arise during the execution of the generic MST algorithm is said to be **X-good** where X $\in$ {minimal, Boruvka, Kruskal, Prim} depending on the criteria used. By assumption, the empty set T is X-good for any X. The correctness of these algorithms amounts to showing that "*X-good implies minimally-good*" where X = Kruskal, Boruvka or Prim. Of course, minimally-good is synonymous with feasibility. We now prove this:

Lemma 16 (Correctness of Algorithm X)
Let $T \subseteq E$.
(a) If T is Prim-good then T is Boruvka-good.
(b) If T is Kruskal-good then T is Boruvka-good.
(c) If T is Boruvka-good then T is minimally-good.

Proof. In each case, we want to show that if T is X-good, then it is Y-good, for appropriate X and Y. We use induction on $|T|$. When $|T| = 0$ and the lemma holds trivially. Suppose T and $T+e$ are X-good. By induction, T is Y-good. It remains to show that $T+e$ is Y-good.

(a) X=Prim, Y=Boruvka: We must show that $T+e$ is Boruvka-good. Since $T+e$ is assumed to be Prim-good, we know e has least cost among the edges that extend the T-component C_s containing the source s. So C_s is a Boruvka-witness for e. Thus $T+e$ is Boruvka-good.

(b) X=Kruskal, Y=Boruvka: We must show that $T+e$ is Boruvka-good. Since $T+e$ is assumed to be Kruskal-good, we know e has least cost among the edges that extend *any*

T-component. The T-component for which e has least cost serves as a Boruvka-witness for e. Thus $T + e$ is Boruvka-good.

(c) X=Boruvka, Y=minimally: We need to prove that $T + e$ is minimally-good. By the Boruvka-goodness of T, there is a T-component U which is the Boruvka-witness for e. By induction hypothesis, T is minimally-good. Hence there is a MST T' that contains T. If $e \in T'$, then we are done (as T' is witness that $T + e$ is minimally-good). So assume $e \notin T'$. This means that T'_e contains a closed path Z.

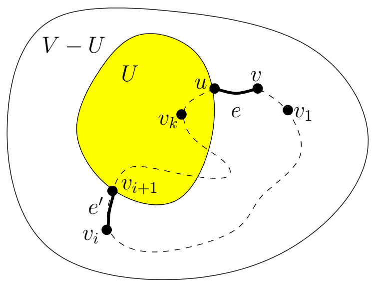


Figure 13: Extending a component U by $e = (u, v)$.

Let $e = u - v$ where $u \in U$ and $v \notin U$. The closed path Z has the form

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

Clearly, there exists some $i = 0, \ldots, k$ such that $v_i \notin U$ and $v_{i+1} \in U$ where $v = v_0$ and $u = v_{k+1}$ in this notation. Rewriting,

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

Let $e' := (v_i - v_{i+1})$. Note that $T'' := (T' - e') + e$ is acyclic (since we broke the unique cycle Z by omitting e'). It is also a spanning tree. Looking at costs, we have $C(e) \leq C(e')$, by our choice of e as least cost edge out of U. Hence $C(T'') \leq C(T')$. Since T' is MST, this means T'' is also an MST. Thus $T + e$ is minimally-good as it is contained in T''.

Q.E.D.

This minimally-good criterion is computational ineffective. The remaining three criteria are effective and are named after the inventors of three well-known MST algorithms. We next discuss the algorithmic techniques needed to make these criteria effective:

- (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 $T + e$ is acyclic? If e is $u - v$, this amounts to checking if u, v are in the same connected component of the graph $G|T = (V, T)$. A simple method is to do this is have a linked list for each connected component of $G|T$, with the nodes of the linked list representing vertices of the component. Given a vertex u, assume we have a pointer from u to the representative 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 $T + e$ 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) Suppose T is a Boruvka-good set. If T is not a spanning tree of G, the number of T-components is at least 2. Let these T-components be $U_1, \ldots, U_k$ ($k \geq 2$). For each U_i, there is at least one e_i that extends U_i with least cost. These e_i's need not be distinct (it is possible that $e_i = e_j$ with $i \neq j$). Nevertheless, there are at least $\lceil k/2 \rceil$ distinct edges in the set $\{e_1, \ldots, e_k\}$. Algorithmically, we want maintain the least cost edge that extends each S-component. We can exploit the union-find data structure (see Chapter 13), but at present, a simple direct solution can be found. The 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.
- (Prim) Because of its focus on one component, Prim's algorithm is somewhat easier to implement than Boruvka's. The ultimate version of Prim's algorithm can only be taken up in Chapter VI (amortization techniques).

¶40. Good sets of vertices. Let us extend the notion of "goodness" to sets of vertices. For any set $T \subseteq E$ of edges, let $V(T)$ denote the set of vertices that are incident on some edge of T. We say a set $S \subseteq V$ is **X-good** if there exists an X-good set $T \subseteq E$ such that $S = V(T)$. Here, X is equal to 'minimally', 'Prim', 'Kruskal' or 'Boruvka'. We also declare any singleton set with only one vertex to be X-good.

¶41. Boruvka's MST Algorithm. Let $T \subseteq E$ be Boruvka-good, and $G|T = (V, T)$. If C is a connected component of $G|T$, 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 $T + e$ is Boruvka-good. If $G|T$ has k connected components, we pick one such minimum outgoing edge for each component. Let e_i be the edge picked edge for component C_i ($i = 1, \ldots, k$). Our goal want to extend T to

$$T' \leftarrow T \cup \{e_1, \ldots, e_k\}. \tag{49}$$

There are two issues. First, it may happen that the same edge may be picked by two components: $e_i = e_j$ for $i \neq j$. Let there be k' distinct edges in the set $\{e_1, \ldots, e_k\}$. By re-indexing the edges if necessary, assume that the k' distinct edges are $\{e_1, \ldots, e_{k'}\}$. Since no more than two components may pick the same minimum outgoing edge, there are at least $k/2$ distinct edges, i.e.,

$$k' \geq k/2.$$

Next, we need to be sure that T' in (49) is acyclic. How might a cycle arise? Consider a cycle of $m \geq 2$ distinct components of the form

$$[C_1, C_2, \ldots, C_m]$$

such that C_j picks $e_j = (u_j - v_j)$ where $u_j \in C_j$ and $v_j \in C_{j+1}$ (assume $C_{m+1} = C_1$ in this notation). Let w_j be the cost of e_j. Since C_{j+1} picks e_{j+1} but not e_j. it implies $w_{j+1} \leq w_j$ for all $j = 1, \ldots, m$. Since these m inequalities creates a cycle, we conclude that $w_1 = w_2 = \cdots = w_m$.

Note that if $m = 2$, this does not constitute a cycle since $e_1 = e_2$. However, if $m \geq 3$, then we have a cycle. To prevent cycles, *we now assume some tie-breaking rule among edges with the same cost so that they are totally ordered.* We now claim that no cycle can arise for $m \geq 3$. By way of contradiction, suppose we have a cycle. Then the set $\{e_j : j = 1, \ldots, m\}$ has m distinct edges. Wlog, assume e_1 is the minimum edge of $\{e_j : j = 1, \ldots, m\}$. Since e_1 and e_2 are incident on C_2, our tie breaking rule implies that C_2 must pick e_1 and not e_2, contradiction.

Thus our tie breaking rule implies T' is acyclic. It follows that each of the edges $e_1, \ldots, e_{k'}$ reduces the number of components in T by 1. Thus T' has $k - k'$ components. Since T' has at least one component, $k - k' \geq 1$ or $k > k'$. In summary:

$$\left.\begin{array}{l}\text{the number } k' \text{ of distinct edge in the set } \{e_1, e_2, \ldots, e_k\} \\ \text{is between } k/2 \text{ and } k-1, \text{ and } T \cup \{e_1, \ldots, e_k\} \text{ is acyclic.}\end{array}\right\} \tag{50}$$

This extension of T by simultaneously adding $k' \geq k/2$ edges is called a "phase". 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 $T \subseteq E$ of G
$T \leftarrow \emptyset$ $\triangleleft$ *Initialize a Boruvka-good set of edges*
While $(|T| < n - 1)$
  for each connected component C of $G|T$, $\triangleleft$ *Do Phase*
    $e \leftarrow \mathrm{argmin}\,\{C(e) : e = (u - v), u \in C, v \notin C\}$
    $T \leftarrow T + \{e\}$

How many phases can there be? If $G|S$ has k components, and we form T' by adding k' distinct edges to T, then $G|T'$ has $k - k'$ connected components. From (50), we conclude that $1 \leq k - k' \leq k/2$. 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 $O((m + n) \lg n)$.

¶42. 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 ¶IV.23, we have provided such an algorithm based on BFS.

- The main data structure is an array $\mathtt{CC}[1..n]$ to keep track of the connected components of $G|T$. For each $i \in \{1, \ldots, n\} = V$, let $\mathtt{CC}[i]$ 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 $\mathtt{CC}[j] = j$. Equivalently, we have an "idempotent" property: for all i, $\mathtt{CC}[i] = \mathtt{CC}[\mathtt{CC}[i]]$. Initially, $T = \emptyset$ and hence we have $\mathtt{CC}[i] = i$ for all i. Assume inductively that the array $\mathtt{CC}[1..n]$ is available at the beginning of the phase. If $\mathtt{CC}[i] = j$, 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 $A[j] \leftarrow \mathtt{nil}$, 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 $(i - j) \in E$,
  If $\mathtt{CC}[i] \neq \mathtt{CC}[j]$ $\triangleleft$ *$(i - j)$ is an outgoing edge of component of i*
    If the cost of $A[\mathtt{CC}[i]]$ is "greater" than $C(i - j)$
      $A[\mathtt{CC}[i]] \leftarrow (i - j)$.

We write "greater" in quotes because in case of ties, we must break ties uniquely. One way is this: assume that the set E is totally sorted by $<_E$. we say the cost of $e \in E$ is "greater" than the cost of $e' \in E$ if $C(e) > C(e')$ or else $C(e) = C(e')$ *and* $e >_E e'$. This is essential to avoid cycles. At the end of this for-loop, it is clear that the array A has the desired information.

- We must now extend the set T to T': this amounts to updating the array $\mathtt{CC}[1..n]$ 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 $(j - k) \leftarrow A[\mathtt{CC}[i]]$. We add $\mathtt{CC}[i]$ to V^r and $(\mathtt{CC}[j] - \mathtt{CC}[k])$ to E^r. Then we use a BFS Driver to compute the connected components of G^r. Assume (see ¶IV.23) that the BFS Driver returns an array $\mathtt{CC}^r$ such that for each $j \in V^r$, we have $\mathtt{CC}^r[j]$ is a representative vertex in the connected component of j. Now we update $\mathtt{CC}$ with the help of $\mathtt{CC}^r$:

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

  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 [10] 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: Consider the bigraph G_5 with vertex set $V = \{a, b, c, d, e\}$ in Figure 10. We change its cost function C as follows: view G_5 as an Euclidean graph where each vertex $v \in V$ is a point $pos(v) \in \mathbb{R}^2$:

$$pos(a) = (5, 2), pos(b) = (0, 2), pos(c) = (0, 0), pos(d) = (5, 0), pos(d) = (8, 1).$$

The cost of an edge $u - v$ is the Euclidean distance between $pos(u)$ and $pos(v)$. E.g.,

$$C(a - c) = \|pos(a) - pos(c)\| = \|(5, 2) - (0, 0)\| = \sqrt{5^2 + 2^2} = \sqrt{29}.$$

Run Prim's algorithm and Kruskal's algorithm on this bigraph. ♢

Exercise 6.2: 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 14.

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 $C(u, v) = C(u) + C(v)$? ♢

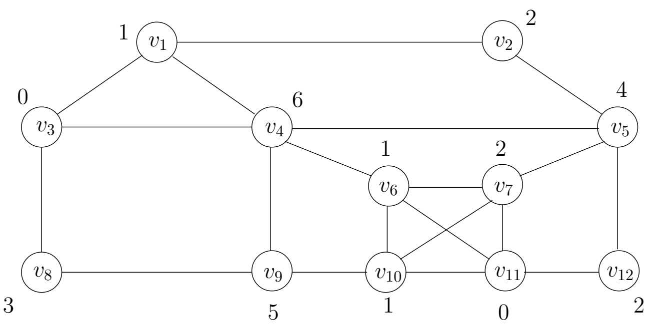


Figure 14: The house graph: The cost of edge $v_i - v_j$ is defined as $C(v_i) + C(v_j)$, where $C(v)$ is the value indicated next to v. E.g. $C(v_1 - v_4) = 1 + 6 = 7$.

Exercise 6.3: Redo the previous problem with a different cost function, where $C(u - v) = C(u)C(v)$ (the product instead of the sum). $\diamondsuit$

Exercise 6.4: Simulating Prim's and Kruskal's Algorithm
Consider the graph G_{ag} in Figure 15 with edge costs. In simulations, please do it "canonically" meaning that if there is a choice, pick the smallest index vertex or value first.

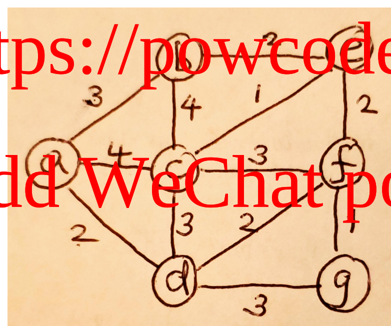


Figure 15: Graph G_{ag}

(a) Simulate Prim's algorithm using the tabular form shown in the text. At the end, list the edges and total cost of the MST.
(b) Simulate Kruskal's algorithm by listing the sorted edges and then accepting or rejecting each in increasing order of cost. Is your Kruskal MST the same as the Prim MST?
(c) Although the sorted sequence in Kruskals algorithm has length m, we know that Kruskal's algorithm can stop as soon as it has accepted $n - 1$ edges. Prove that in the worst case, Kruskal's algorithm needs to examine $\Omega(n^2)$ edges. How to show such a result? Show this: *there is an infinite family*

$$G_n = (V_n, E_n; C_n), \quad |V_n| = n$$

of inputs for the MST problem such that Kruskal's algorithm on G_n must examine $\Omega(n^2)$ edges in the worst case. $\diamondsuit$

Exercise 6.5: 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 6.6: Let G_n be the bigraph whose vertices are $V = \{1, 2, \ldots, n\}$. It has two kinds of edges:
(i) **Prime edges**: for each $i \in V$, if i is prime, then $(1, i) \in E$ with cost i. [Recall that 1 is not considered prime, so 2 is the smallest prime.]
(ii) **Divisibility edges**: For $1 < i < j$, if i divides j then we add (i, j) to E with cost j/i (which is an integer).
(a) Draw the graph G_{10}.
(b) Compute the MST of G_{10} using Prim's algorithm, using node 1 as the source vertex. State the cost of the MST.
(c) Repeat part(b) but using Kruskal's algorithm.
(d) Repeat part(b) but using Boruvka's algorithm. [Organize Boruvka's simulation in a Phase-by-Phase manner] ♢

Exercise 6.7: Consider the weighted bigraph G_7 in Figure 16.

Figure 16: Graph G_7

(a) Please compute the MST of G_7 using Kruskal's algorithm. What is the cost of the MST?

(b) Please compute the MST of G_7 using Prim's algorithm. You must also list the edges of the MST.

♢

Exercise 6.8: *MST Update Problem*: we are given a connected weighted bigraph $G = (V, E; C)$ where the cost of edges can change. Suppose you have already computed an MST T. Now $C(u - v)$ is changed for some edge $(u - v) \in E$. *This question is coupled with part(a) of the previous question: we assume you have computed an MST for the graph G_7 in Figure 16).* Consider two cases.

(i) Suppose $(u - v) \in T$, and $C(u - v)$ is increased. How can you update T? You do not need to justify why it is correct. Illustrate your method by explaining how MST T of G_7 is updated when $C(e - f)$ changes from 3 to 10.

(ii) Suppose $(u - v) \notin T$, and $C(u - v)$ is decreased. How can you update T? Illustrate your method by explaining how MST T of G_7 is updated when $C(c - d)$ changes from 11 to 3.

♢

Exercise 6.9: 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 6.10: Describe the rule for reconstructing the MST from the matrix M using in our hand-simulation of Prim's Algorithm. ♢

Exercise 6.11: Hand simulation of Kruskal's Algorithm on the graph of Figure 14. 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 6.12: 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 17). 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

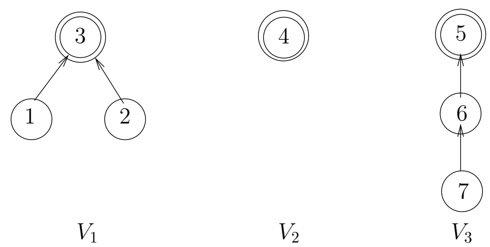


Figure 17: Directed graph $G(D)$ with three components (V_1, V_2, V_3)

component points to the root). E.g., in Figure 17, 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.
(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 17 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. ♢

Exercise 6.13: 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 $c_0 - C(e) > 0$. ♢

Exercise 6.14: 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? ♢

Exercise 6.15: 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. ♢

Exercise 6.16: 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. $\diamondsuit$

Exercise 6.17: Extend the algorithm to finding MIS in contracted matroids. $\diamondsuit$

Exercise 6.18: If $T \subseteq E$ is Prim-good, then clearly $G' = (V(T), T)$ is clearly a tree. Prove that T is actually an MST of the restricted graph G'. $\diamondsuit$

Exercise 6.19: Given a costed bigraph $G = (V, E; C)$, let $MST(G)$ denote the set of all MST's of G. Define **MST exchange graph** $Exch(G)$ of G to be the bigraph $(MST(G), H)$ whose nodes are MST's and whose edges $T - T' \in H$ are characterized by this property: the symmetric difference $T \oplus T' = (T \setminus T) \cup (T' \setminus T)$ has size 2.
(a) Let G_5 be bigraph in Figure 11. Draw the MST exchange graph $Exch(G_5)$, *assuming that $MST(G_5)$ is comprised of the 5 listed MST's.*
(b) Prove that if $T \neq T' \in MST(G)$, then there exists $e \in T \setminus T'$ and $e' \in T' \setminus T$ such that $T + e' - e \in MST(G)$. Conclude that $Exch(G)$ is a connected graph. $\diamondsuit$

Exercise 6.20: Refer to the graph in Figure 11.
(a) List all the X-good sets of size 1 for the graph in this figure, where X=Kruska, Boruvka or Prim. For Prim-goodness, assume that node a is the source in Figure 10.
(b) Repeat part(a) but for sets of size 2. $\diamondsuit$

Exercise 6.21: (a) Let G_5 be the bigraph in Figure 11. Prove that there are no other MST's other than the 5 which are shown.
(b) Draw the exchange graph if all the edges in Figure 11 are unit costs? $\diamondsuit$

Exercise 6.22:
(a) Enumerate the X-good sets of vertices in Figure 10. Here, X is 'minimally', 'Kruskal', 'Boruvka' or 'Prim'.
(b) Characterize the good singletons (relative to any of the three notions of goodness). $\diamondsuit$

Exercise 6.23: 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 $n/2 \leq P \leq n - 1$. 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 $([u] \overset{v}{-})$ such that there exists an edge $(u' - v') \in E$ where $u' \in [u]$ and $v' \in [v]$. The cost of $([u] \overset{v}{-})$ 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 6.24: (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 **minimally-good** if there is an MST that contains S but not containing any edge of R. Note that this extends our original definition of "minimally 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. ♢

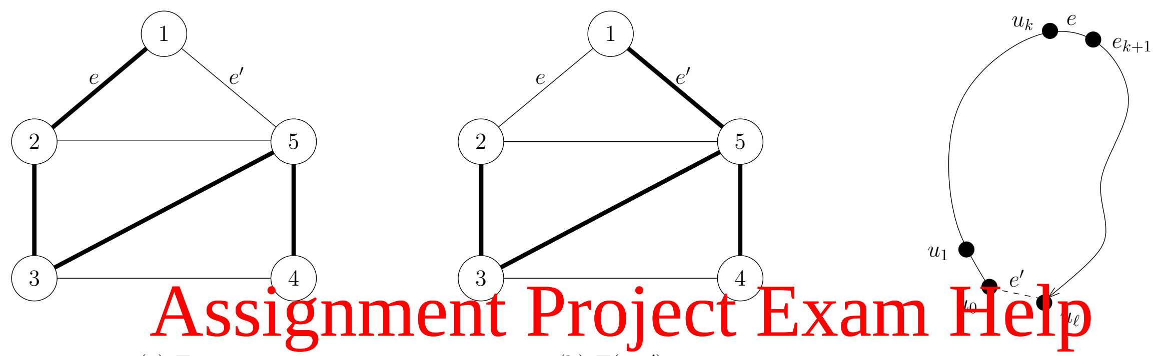
Exercise 6.25: With respect to the generic accept/reject version of MST:
(a) Give a counter example to the following rejection rule: let e and e' be two edges in a U-cut. If $C(e) \geq C(e')$ then we may reject e'.
(b) Can the rule in part (a) be fixed by some additional properties that we can maintain?
(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? ♢

Exercise 6.26: 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? ♢

Exercise 6.27: Is there an analogue of Prim and Boruvka's algorithm for the MIS problem for matroids? ♢

Exercise 6.28: 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 6.29: 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 $e' \in E \setminus T$.
(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 18(a), (b)).



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

Figure 18: (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]

(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 18(a), $e' = (1-5) \equiv (5-1)$. 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 18(a), $First[1, 5] = 2$ and $First[5, 1] = 3$. Show the matrix $First$ for the tree T in Figure 18(a). Similarly, give the matrix $First$ for the tree $T(e, e')$ in Figure 18(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 18(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?
(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. $\diamondsuit$

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.

¶43. Graphic matroids. Let $G = (V, S)$ be a bigraph. A subset $A \subseteq S$ is **acyclic** if it does not contain any cycle. The I comprising all acyclic subsets of S is called the **graphic matroid** of G. Let us prove 2 critical proper We note two properties of I:

Hereditary property: If $A \subseteq B$ and $B \in I$ then $A \in I$. In particular, the empty set belongs to 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. CLAIM: *There exists a component $U \subseteq V$ of $G|B$ that is not contained in any component of $G|A$.* [Pf: 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.] Let $T := B|U$ be the restriction of B to U. Since U is a component, the graph (U, T) is a spanning tree. Hence there exists 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: that is, $A + e$ contains no cycle and hence $A + e \in I$.

For example, in Figure 10 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.

¶44. Matroids. The above system (V, 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.

¶45. 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[19] 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.

¶46. 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 $M|R$.

¶47. 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 $r_M(R)$.

¶48. 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[20] 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:

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

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

Greedy Algorithm for Maximum Base:
Input: matroid $M = (S, I; C)$ with cost function C.
Output: a base $A \in I$ with maximum cost.
1. Sort $S = \{x_1, \ldots, x_n\}$ by cost.
   Suppose $C(x_1) \ge C(x_2) \ge \cdots \ge C(x_n)$.
2. Initialize $A \leftarrow \emptyset$.
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 17 (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 $C(x) \le C(z_{i+1})$.
3. Let $B = \{u_1, \ldots, u_k\}$ be an independent set where $C(u_1) \ge C(u_2) \ge \cdots \ge C(u_k)$. Then $k \le m$ and $C(u_i) \le C(z_i)$ for all i.

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 $(x_1, \ldots, x_m, \ldots, x_n)$.

¶49. 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 10. 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 10, 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. ♢

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 take a small detour from greedy methods in order to address this interesting topic. A survey of this classic problem is given by Sedgewick [9]. Perhaps the oldest incarnation of this problem is the "change ringing problem" of bell-ringers in early 17th Century English churches [8]. This calls for ringing a sequence of n bells in all $n!$ permutations.

¶50. Combinatorial Enumeration Problems. The problem of generating all permutations efficiently is representative of the important class of **combinatorial enumeration problems**. For instance, we might want to generate all k-sets (i.e., subsets of size k) 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 set X is a surjective function $p : \{1, \ldots, n\} \to X$. Surjectivity of p implies $|X| \le n$. The function p may be represented by the sequence $(p(1), p(2), \ldots, p(n))$. If $n > |X|$, then we could have $p(i) = p(j)$ for some $i \ne j$. 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 $(p(1), p(2), \ldots, p(n))$.

Example: let $X = \{a, b, c\}$ and $n = 4$. Two 4-permutations of X are $(a - a - b - c)$ and $(b - c - a - b)$. However, $(a - b - c)$ and $(a - b - a - b)$ are not 4-permutations of X.

The set of all n-permutations of $X = \{1, 2, \ldots, n\}$ is rather special, and is denoted S_n. Each element of S_n is simply called an **n-permutation** (i.e., X is implicitly $\{1, \ldots, n\}$). Note that $|S_n| = n!$. Here is an enumeration of S_3:

$$(1-2-3), \quad (1-3-2), \quad (3-1-2); \quad (3-2-1), \quad (2-3-1), \quad (2-1-3). \qquad (51)$$

Two n-permutations $\pi = (x_1 - \cdots - x_n)$ and $\pi' = (x'_1 - \cdots - x'_n)$ are **adjacent** (to each other) if the two permutations are identical except that for some $i = 2, \ldots, n$, we have $(x'_{i-1}, x'_i) = (x_i, x_{i-1})$. In other words, (x'_{i-1}, x'_i) and (x_{i-1}, x_i) are interchanged. We write $\pi' = Exch_i(\pi)$ in this case, and call π' an **exchange** of π (and vice-versa). 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 pair of consecutive permutations in this listing are adjacent. For instance, the listing of S_3 in (51) is an adjacency ordering. The **adjacency graph** of S_n is the bigraph with vertex set S_n and edges given by adjacency permutations. For example, the adjacency graph of S_3 consists of one cycle given by (51): consecutive permutations in (51) are edges of this graph, and also the first $(1 - 2 - 3)$ and last $(2 - 1 - 3)$ permutations. Since each permutation in S_3 is adjacent to exactly two other permutations, there are no other edges. Thus this cycle is unique. 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 $i = 1, \ldots, n$. When $i = n$, we intepret "$Ext_n(\pi)$" to mean that we append '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 $(3 - 1 - 2), (1 - 3 - 2), (1 - 2 - 3)$.

¶51. The Johnson-Trotter Ordering. Among the several known methods to generate 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 [8]. 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

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

or the reverse sequence

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

E.g.,

$$\begin{array}{rcllll} \texttt{UP}(1-2-3) & \equiv & (4-1-2-3), & (1-4-2-3), & (1-2-4-3), & (1-2-3-4). \\ \texttt{DOWN}(1-3-2) & \equiv & (1-3-2-4), & (1-3-4-2), & (1-4-3-2), & (4-1-3-2). \end{array}$$

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

$$\texttt{UP}(\pi); \texttt{DOWN}(\pi')$$

and

$$\texttt{DOWN}(\pi); \texttt{UP}(\pi')$$

are both adjacency orderings. We have thus shown:

Lemma 18 (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*

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

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 $\texttt{DOWN}(\pi_1); \texttt{UP}(\pi_2)$ is an adjacency ordering. Indeed, this is the ordering shown in (54).

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 18 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

$$(\texttt{DOWN}(\pi_1); \texttt{UP}(\pi_2); \cdots ; \texttt{UP}(\pi_m))$$

is a Hamiltonian cycle for G_n.

¶52. 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[21] the generator repeatedly, where each call to the generator returns the next permutation. Alternatively, we view the generator as a skeleton (or shell) program with the consumer program as a subroutine (or macro). We prefer the latter view, since this fits the established paradigm of BFS and DFS as skeleton programs (see Chapter 4).

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

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

$$per[i] \Leftrightarrow per[i-1] \tag{52}$$

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

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

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

¶53. 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. Hence, the counter array C can hold $n!$ 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., the level of $C[1..5] = [1, 2, 3, 4, 5]$ is 1, but the level of $C[1..5] = [1, 2, 2, 1, 5]$ is 4.

We define the **increment** of this counter array as follows: Let ℓ be the level of the counter. If $\ell = 1$, then we reset each $C[i]$ to 1. Otherwise, we increment $C[\ell]$ and reset $C[i] \leftarrow 1$ for all $i > \ell$. E.g., the increment of $C[1..5] = [1, 2, 3, 4, 5]$ is $[1, 1, 1, 1, 1]$, and the increment of $C[1..5] = [1, 2, 2, 1, 5]$ is $[1, 2, 2, 2, 1]$. 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). E.g., for $n = 4$ and starting with the initial value $C[2, 3, 4] = [1, 1, 1]$, the successive increments of this array produce the following cyclic sequence:

$$\begin{array}{rcl} C[2,3,4] & = & [1,1,1] \to [1,1,2] \to [1,1,3] \to [1,1,4] \\ & \to & [1,2,1] \to [1,2,2] \to [1,2,3] \to [1,2,4] \\ & \to & [1,3,1] \to \cdots \\ & \to & [2,1,1] \to \cdots \\ & \to & [2,2,1] \to \cdots \\ & \to & [2,3,1] \to [2,3,2] \to [2,3,3] \to [2,3,4] \\ & \to & [1,1,1] \to \cdots . \end{array} \tag{53}$$

Let the cost of incrementing the counter array be equal to $n + 1 - \ell$ where ℓ is the level.

Lemma 19 *The cost to increment the counter array from* $[1, 1, \ldots, 1]$ *to* $[1, 2, 3, \ldots, n]$ *is* $< 4(n!)$.

Proof. The ℓth counter $C[\ell]$ is updated at every $(n!/\ell!)$th step. In particular, $C[n]$ is updated at every step. So that overall, $C[\ell]$ is updated $\ell!$ times. The total cost to update $C[\ell]$ is therefore $(n+1-\ell)\ell!$. Summing up these costs for $\ell = 2, \ldots, n$,

$$1 \cdot n! + 2 \cdot (n-1)! + \cdots + (n-1) \cdot 2! < n! \sum_{i=1}^{\infty} i 2^{1-i} = 4(n!).$$

Q.E.D.

We can now describe a preliminary version our permutation generator, viewed[22] as a "Shell Program" with macros:

Johnson-Trotter Generator (Version 1)

Input: (n, I) where $n \geq 2$ and I is application-dependent data

▷ *Initialization*

$per[1..n] \leftarrow [1, 2, \ldots, n]$. ◁ *Initial permutation*

$C[2..n] \leftarrow [1, 1, \ldots, 1]$. ◁ *Initial counter value*

$\boxed{\mathrm{INIT}(n, I)}$

▷ *Main Loop*

do

$\ell \leftarrow Inc(C)$

Update(ℓ) ◁ *The permutation is updated*

$\boxed{\mathrm{CONSUME}(per, I)}$ ◁ *Permutation is consumed*

While $(\ell > 1)$

The input I is application-dependent. In shell programming, the output is established by convention to be the value of some global variable(s). We have two shell macros $\boxed{\mathrm{INIT}(n, I)}$ and $\boxed{\mathrm{CONSUME}(per, I)}$. However, Update($\ell$) is not a macro, but a subroutine to be described next – its role is to generate the next permutation encoded in the array per.

Here are two simple applications of this shell:

- Printing all n-permutations: the input I is empty and $\boxed{\mathrm{INIT}(n, I)}$ is a NO-OP. The macro $\boxed{\mathrm{CONSUME}(per)}$ simply print the permutation per.
- Computing $Opt(\boldsymbol{w})$ in the global bin packing problem. Here the input I is $\boldsymbol{w} = (w_1, \ldots, w_n)$ where each $0 < w_i < 1$. Recall that $Opt(\boldsymbol{w})$ is the minimum number of

[22] In the style of tree traversal (Chapter 3) or graph traversal (Chapter 4).

unit bins that can hold all the weights in $\boldsymbol{w}$. As output, we define a global integer variable b whose final value will be $Opt(\boldsymbol{w})$. $\boxed{\text{INIT}(n, \boldsymbol{w})}$ simply initializes a global variable $b \leftarrow n$. The macro $\boxed{\text{CONSUME}(per)}$ is given by

$$\boxed{\text{CONSUME}(per)} \equiv b \leftarrow \min\{b, \texttt{Grd}(\boldsymbol{w})\}$$

where `Grd` is the greedy algorithm for linear bin packing.

¶54. 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 (53) correspond to the following sequence of permutations:

$$\begin{array}{l} \xrightarrow{[1,1,1]} (1-2-3-\underline{4}) \xrightarrow{[1,1,2]} (1-2-\underline{4}-3) \xrightarrow{[1,1,3]} (1-\underline{4}-2-3) \xrightarrow{[1,1,4]} (4-1-2-\underline{3}) \xrightarrow{[1,2,1]} (\underline{4}-1-3-2) \\ \quad \xrightarrow{[1,2,2]} \quad (1-\underline{4}-3-2) \xrightarrow{[1,2,3]} (1-3-\underline{4}-2) \xrightarrow{[1,2,4]} (1-\underline{3}-2-4) \xrightarrow{[1,3,1]} (3-1-2-\underline{4}) \xrightarrow{[1,3,2]} \cdots \\ \quad \xrightarrow{[2,3,3]} \quad (1-\underline{4}-2-3) \xrightarrow{[2,3,4]} (1-2-\underline{4}-3) \xrightarrow{[1,1,1]} (1-2-3-\underline{4}) \longrightarrow \cdots. \end{array} \tag{54}$$

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[23] 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 (54), 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 $\texttt{UP}[1..n]$ be a Boolean array where $\texttt{UP}[\ell]$ is true in the UP phase, and false in the DOWN phase when we are incrementing a counter at level ℓ. Moreover, the value of $\texttt{UP}[\ell]$ is changed (flipped) each time $C[\ell]$ is reinitialized to 1. For instance, in the first row of (54), $\texttt{UP}[4] = \texttt{false}$ and so the entry 4 is moving down with each swap involving 4. In the next row, $\texttt{UP}[4] = \texttt{true}$ and so the entry 4 is moving up with each swap.

We modify our previous Inc subroutine to include this update:

Increment(C)

Output: Increments C, updates UP, and returns the previous level of C
$\ell \leftarrow n$.
While $(C[\ell] = \ell) \wedge (\ell > 1)$ ◁ *Loop to find the counter level*
  $C[\ell] \leftarrow 1$;
  $\texttt{UP}[\ell] \leftarrow \neg\texttt{UP}[\ell]$; ◁ *Flips the boolean value* $\texttt{UP}[\ell]$
  ℓ--.
If $(\ell > 1)$
  $C[\ell]$++.
Return(ℓ).

[23] In case we want to continue, the case $\ell = 1$ is treated as if $\ell = n$. E.g., in (54), the case $\ell = 1$ is treated as $\ell = 4$.

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])
    $per[pos[\ell]] \Leftrightarrow per[pos[\ell]+1]$; ◁ *modify permutation*
    $pos[per[pos[\ell]]] \leftarrow pos[\ell]$; ◁ *update position array*
    $pos[\ell]$++; ◁ *update position array*
  else
    $per[pos[\ell]] \Leftrightarrow per[pos[\ell]-1]$;
    $pos[per[pos[\ell]]] \leftarrow pos[\ell]$;
    $pos[\ell]$−−;

The final version of our Johnson-Trotter algorithm is:

Johnson-Trotter Generator

Input: (n, I) where $n \geq 2$ and I is application-dependent data
▷ *Initialization*
  $per[1..n] \leftarrow [1, 2, \ldots, n]$. ◁ *Initial permutation*
  $pos[1..n] \leftarrow [1, 2, \ldots, n]$. ◁ *Initial positions*
  $C[2..n] \leftarrow [1, 1, \ldots, 1]$. ◁ *Initial counter value*
  $\boxed{\text{INIT}(n, I)}$
▷ *Main Loop*
  do
    $\ell \leftarrow Increment(C)$;
    Update(ℓ); ◁ *The permutation is updated*
    $\boxed{\text{CONSUME}(per, I)}$ ◁ *Permutation is consumed*
  While($\ell > 1$)

The Java code for this algorithm is presented in an appendix of this chapter. We conclude with this theorem about complexity.

Theorem 20 (Complexity of Johnson-Trotter Generator) *he time complexity to generate each n-permutation is $O(n)$. However, this complexity is $O(1)$ in an*[24] *amortized sense.*

[24] Amortized cost is $T(n)/n!$ where $T(n)$ is the total cost T to generate all $n!$ permutations. In Chapter 6, we study amortization in general.

Proof. Inside the main loop, we call two subroutines, Increment(C) and Update. By Lemma 19, the cost of calling Inc(C) (and similarly for Increment(C)) is $O(n)$ in the worst case, but $O(1)$ when amortized over the entire algorithm. We also observe that Update is worst case $O(1)$. **Q.E.D.**

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 [7]. 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. ♢

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

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

§9 APPENDIX: Java Code for Permutations

```
/****************************************************
 * Per(mutations)
 *      This generates the Johnson-Trotter permutation order.
 *      An n-permutation is 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 = (args.length>0) ?  Integer.valueOf(args[0]) : 3;
  int val = (args.length>1) ? Integer.valueOf(args[1]) : 0;
  boolean verbose = (val>0)? true : false;

  //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] G. Dósa and J. Sgall. First Fit bin packing: A tight analysis. In *Proc. 30th STACS*, volume LIPIcs vol.20, pages 538–549. Schloss Dagstuhl, 2013. Proc. 31st ICALP.

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

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

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

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

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

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

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