Introduction to Algorithms

chapter 15

exercises

> 15.1-4 Modify $\mathrm{MEMOIZED\text{-}CUT\text{-}ROD}$ to return not only the value but the actual solution, too.

```
MEMOIZED-CUT-ROD(p, n)
    let r[0..n] and s[0..n] be new arrays
    for i = 0 to n
        r[i] = -∞
    (val, s) = MEMOIZED-CUT-ROD-AUX(p, n, r, s)
    print "The optimal value is" val "and the cuts are at" s
    j = n
    while j > 0
        print s[j]
        j = j - s[j]
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r, s)
    if r[n] ≥ 0
        return r[n]
    if n == 0
        q = 0
    else q = -∞
        for i = 1 to n
            (val, s) = MEMOIZED-CUT-ROD-AUX(p, n - i, r, s)
            if q < p[i] + val
                q = p[i] + val
                s[n] = i
    r[n] = q
    return (q, s)
```

> 1The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

```
FIBONACCI(n)
    let fib[0..n] be a new array
    fib[0] = 1
    fib[1] = 1
    for i = 2 to n
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

There are $n + 1$ vertices in the subproblem graph, i.e., $v_0, v_1, \ldots, v_n$.

- For v_0, v_1, each has 0 leaving edge.
- For $v_2, v_3, \ldots, v_n$, each has 2 leaving edges.

Thus, there are $2n - 2$ edges in the subproblem graph.

> 15.4-2 Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

```
PRINT-LCS(c, X, Y, i, j)
    if c[i, j] == 0
        return
    if X[i] == Y[j]
        PRINT-LCS(c, X, Y, i - 1, j - 1)
        print X[i]
    else if c[i - 1, j] > c[i, j - 1]
        PRINT-LCS(c, X, Y, i - 1, j)
    else
        PRINT-LCS(c, X, Y, i, j - 1)
```

problems

> Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t . Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

Since any longest simple path must start by going through some edge out of s, and thereafter cannot pass through s because it must be simple, that is,

$$\mathrm{LONGEST}(G, s, t) = 1 + \max_{s \sim s'} \backslash LONGEST(G|_{V \backslash s}, s', t),$$

with the base case that if $s = t$ then we have a length of 0.

A naive bound would be to say that since the graph we are considering is a subset of the vertices, and the other two arguments to the substructure are distinguished vertices, then, the runtime will be $O(|V|^2 2^{|V|})$. We can see that we will actually have to consider this many possible subproblems by taking $|G|$ to be the complete graph on $|V|$ vertices.

> A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, $\mathrm{racecar}$, and $\mathrm{aibohphobia}$ (fear of palindromes).
>
> Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input $\mathrm{character}$, your algorithm should return carac. What is the running time of your algorithm?

Let $A[1..n]$ denote the array which contains the given word. First note that for a palindrome to be a subsequence we must be able to divide the input word at some position i, and then solve the longest common subsequence problem on $A[1..i]$ and $A[i + 1..n]$, possibly adding in an extra letter to account for palindromes with a central letter. Since there are n places at which we could split the input word and the

LCS problem takes time $O(n^2)$, we can solve the palindrome problem in time $O(n^3)$.

chapter 16

exercises

> 16.1-1 Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

```
DYNAMIC-ACTIVITY-SELECTOR(s, f, n)
    let c[0..n + 1, 0..n + 1] and act[0..n + 1, 0..n + 1] be new tables
    for i = 0 to n
        c[i, i] = 0
        c[i, i + 1] = 0
    c[n + 1, n + 1] = 0
    for l = 2 to n + 1
        for i = 0 to n - l + 1
            j = i + l
            c[i, j] = 0
            k = j - 1
            while f[i] < f[k]
                if f[i] ≤ s[k] and f[k] ≤ s[j] and c[i, k] + c[k, j] + 1 > c[i, j]
                    c[i, j] = c[i, k] + c[k, j] + 1
                    act[i, j] = k
                k = k - 1
    print "A maximum size set of mutually compatible activities has size" c[0, n + 1]
    print "The set contains"
    PRINT-ACTIVITIES(c, act, 0, n + 1)
```

```
PRINT-ACTIVITIES(c, act, i, j)
    if c[i, j] > 0
        k = act[i, j]
        print k
        PRINT-ACTIVITIES(c, act, i, k)
        PRINT-ACTIVITIES(c, act, k, j)
```

- GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$ time and
- DYNAMIC-ACTIVITY-SELECTOR runs in $O(n^3)$ time.

> Prove that the fractional knapsack problem has the greedy-choice property.

Let I be the following instance of the knapsack problem: Let n be the number of items, let v_i be the value of the ith item, let w_i be the weight of the ith item and let W be the capacity. Assume the items have been ordered in increasing order by v_i/w_i and that $W \geq w_n$.

Let $s = (s_1, s_2, \ldots, s_n)$ be a solution. The greedy algorithm works by assigning $s_n = \min(w_n, W)$, and then continuing by solving the subproblem

$$I' = (n-1, v_1, v_2, \ldots, v_{n-1}, w_1, w_2, \ldots, w_{n-1}, W - w_n)$$

until it either reaches the state $W = 0$ or $n = 0$.

We need to show that this strategy always gives an optimal solution. We prove this by contradiction. Suppose the optimal solution to I is $s_1, s_2, \ldots, s_n$, where $s_n < \min(w_n, W)$. Let i be the smallest number such that $s_i > 0$. By decreasing s_i to $\max(0, W - w_n)$ and increasing s_n by the same amount, we get a better solution. Since this a contradiction the assumption must be false. Hence the problem has the greedy-choice property.

> Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on $n - 1$ items with maximum weight $W - w$. Thus, to take a bottom-up approach we must solve the 0-1 knapsack problem for all items and possible weights smaller than W. We'll build an $n + 1$ by $W + 1$ table of values where the rows are indexed by item and the columns are indexed by total weight. (The first row and column of the table will be a dummy row).

For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of of a knapsack including items 1 through $i - 1$ with max weight j, and the total value of including items 1 through $i - 1$ with max weight $j - i.weight$ and also item i. To solve the problem, we simply examine the n, W entry of the table to determine the maximum value we can achieve. To read off the items we include, start with entry n, W. In general, proceed as follows: if entry i, j equals entry $i - 1, j$, don't include item i, and examine entry $i - 1, j$ next. If entry i, j doesn't equal entry $i - 1, j$, include item i and examine entry $i - 1, j - i$.weight next. See algorithm below for construction of table:

```
0-1-KNAPSACK(n, W)
    Initialize an (n + 1) by (W + 1) table K
    for i = 1 to n
        K[i, 0] = 0
    for j = 1 to W
        K[0, j] = 0
    for i = 1 to n
        for j = 1 to W
            if j < i.weight
                K[i, j] = K[i - 1, j]
            else
                K[i, j] = max(K[i - 1, j], K[i - 1, j - i.weight] + i.value)
```

problems

> 16-4 Consider the following algorithm for the problem from Section 16.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i. We consider the tasks in order of monotonically decreasing penalty. When considering task a_j, if there exists a time slot at or before a_j's deadline d_j that is still

> empty, assign a_j to the latest such slot, filling it. If there is no such slot, assign task a_j to the latest of the as yet unfilled slots.
>
> **a.** Argue that this algorithm always gives an optimal answer.
>
> **b.** Use the fast disjoint-set forest presented in Section 21.3 to implement the algorithm efficiently. Assume that the set of input tasks has already been sorted into monotonically decreasing order by penalty. Analyze the running time of your implementation.

a. Let O be an optimal solution. If a_j is scheduled before its deadline, we can always swap it with whichever activity is scheduled at its deadline without changing the penalty. If it is scheduled after its deadline but $a_j.deadline \leq j$ then there must exist a task from among the first j with penalty less than that of a_j . We can then swap aj with this task to reduce the overall penalty incurred. Since O is optimal, this can't happen. Finally, if a_j is scheduled after its deadline and $a_j.deadline > j$ we can swap a_j with any other late task without increasing the penalty incurred. Since the problem exhibits the greedy choice property as well, this greedy strategy always yields on optimal solution.

b. Assume that $\text{MAKE-SET}(x)$ returns a pointer to the element x which is now it its own set. Our disjoint sets will be collections of elements which have been scheduled at contiguous times. We'll use this structure to quickly find the next available time to schedule a task. Store attributes $x.low$ and $x.high$ at the representative x of each disjoint set. This will give the earliest and latest time of a scheduled task in the block. Assume that $\text{UNION}(x, y)$ maintains this attribute. This can be done in constant time, so it won't affect the asymptotics. Note that the attribute is well-defined under the union operation because we only union two blocks if they are contiguous.

Without loss of generality we may assume that task a_1 has the greatest penalty, task a_2 has the second greatest penalty, and so on, and they are given to us in the form of an array A where $A[i] = a_i$. We will maintain an array D such that $D[i]$ contains a pointer to the task with deadline i. We may assume that the size of D is at most n, since a task with deadline later than n can't possibly be scheduled on time. There are at most $3n$ total MAKE-SET, UNION, and FIND-SET operations, each of which occur at most n times, so by Theorem 21.14 the runtime is $O(n\alpha(n))$.

```
SCHEDULING-VARIATIONS(A)
    let D[1..n] be a new array
    for i = 1 to n
        a[i].time = a[i].deadline
        if D[a[i].deadline] != NIL
            y = FIND-SET(D[a[i].deadline])
            a[i].time = y.low - 1
        x = MAKE-SET(a[i])
        D[a[i].time] = x
        x.low = x.high = a[i].time
        if D[a[i].time - 1] != NIL
            UNION(D[a[i].time - 1], D[a[i].time])
        if D[a[i].time + 1] != NIL
            UNION(D[a[i].time], D[a[i].time + 1])
```

16-5 Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the ***cache***—a small but faster memory—overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, \ldots, r_n \rangle$ of n memory requests, where each request is for a particular data element. For example, a program that accesses 4 distinct elements a, b, c, d might make the sequence of requests $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Let k be the size of the cache. When the cache contains k elements and the program requests the $(k+1)$st element, the system must decide, for this and each subsequent request, which k elements to keep in the cache. More precisely, for each request r_i, the cache-management algorithm checks whether element r_i is already in the cache. If it is, then we have a ***cache hit***; otherwise, we have a cache miss. Upon a ***cache miss***, the system retrieves r_i from the main memory, and the cache-management algorithm must decide whether to keep r_i in the cache. If it decides to keep r_i and the cache already holds k elements, then it must evict one element to make room for r_i . The cache-management algorithm evicts data with the goal of minimizing the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of n requests and the cache size k, and we wish to minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called ***furthest-in-future***, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

a. Write pseudocode for a cache manager that uses the furthest-in-future strategy. The input should be a sequence $\langle r_1, r_2, \ldots, r_n \rangle$ of requests and a cache size k, and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm?

b. Show that the off-line caching problem exhibits optimal substructure.

c. Prove that furthest-in-future produces the minimum possible number of cache misses.

a. Suppose there are m distinct elements that could be requested. There may be some room for improvement in terms of keeping track of the furthest in future element at each position. If you maintain a (double circular) linked list with a node for each possible cache element and an array so that in index i there is a pointer corresponding to the node in the linked list corresponding to the possible cache request i. Then, starting with the elements in an arbitrary order, process the sequence $\langle r_1, \ldots, r_n \rangle$ from right to left. Upon processing a request move the node corresponding to that request to the beginning of the linked list and make a note in some other array of length n of the element at the end of the linked list. This element is tied for furthest-in-future. Then, just scan left to right through the sequence, each time just checking some set for which elements are currently in the cache. It can be done in constant time to check if an element is in the cache or not by a direct address table. If an element need be evicted, evict the furthest-in-future one noted earlier. This algorithm will take time $O(n+m)$ and use additional space $O(m+n)$.

If we were in the stupid case that $m > n$, we could restrict our attention to the possible cache requests that actually happen, so we have a solution that is $O(n)$ both in time and in additional space required.

b. Index the subproblems $c[i, S]$ by a number $i \in [n]$ and a subset $S \in \binom{[m]}{k}$. Which indicates the lowest number of misses that can be achieved with an initial cache of S starting after index i. Then,

$$c[i, S] = \min_{x \in S}(c[i+1, r_i \cup (S - x)] + (1 - \chi_{r_i}(x))),$$

which means that x is the element that is removed from the cache unless it is the current element being accessed, in which case there is no cost of eviction.

c. At each time we need to add something new, we can pick which entry to evict from the cache. We need to show the there is an exchange property. That is, if we are at round i and need to evict someone, suppose we evict x. Then, if we were to instead evict the furthest in future element y, we would have no more evictions than before. To see this, since we evicted x, we will have to evict someone else once we get to x, whereas, if we had used the other strategy, we wouldn't of had to evict anyone until we got to y. This is a point later in time than when we had to evict someone to put x back into the cache, so we could, at reloading y, just evict the person we would of evicted when we evicted someone to reload x. This causes the same number of misses unless there was an access to that element that wold of been evicted at reloading x some point in between when x any y were needed, in which case furthest in future would be better.

chapter 23

exercises

> 23.1-2 Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Let G be the graph with 4 vertices: u, v, w, z. Let the edges of the graph be $(u, v), (u, w), (w, z)$ with weights 3, 1, and 2 respectively.Suppose A is the set (u, w). Let $S = A$. Then S clearly respects A. Since G is a tree, its minimum spanning tree is itself, so A is trivially a subset of a minimum spanning tree.

Moreover, every edge is safe. In particular, (u, v) is safe but not a light edge for the cut. Therefore Professor Sabatier's conjecture is false.

> 23.1-3 Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Let T_0 and T_1 be the two trees that are obtained by removing edge (u, v) from a MST. Suppose that V_0 and V_1 are the vertices of T_0 and T_1 respectively.

Consider the cut which separates V_0 from V_1. Suppose to a contradiction that there is some edge that has weight less than that of (u, v) in this cut. Then, we could construct a minimum spanning tree of the whole graph by adding that edge to $T_1 \cup T_0$. This would result in a minimum spanning tree that has weight less than the original minimum spanning tree that contained (u, v)

> 23.1-5 Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - e)$ that is also a minimum spanning tree of G . That is, there is a minimum spanning tree of G that does not include e.

Let A be any cut that causes some vertices in the cycle on once side of the cut, and some vertices in the cycle on the other. For any of these cuts, we know that the edge e is not a light edge for this cut. Since all the other cuts won't have the edge e crossing it, we won't have that the edge is light for any of those cuts either. This means that we have that e is not safe.

> 23.1-7 Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

First, we show that the subset of edges of minimum total weight that connects all the vertices is a tree. To see this, suppose not, that it had a cycle. This would mean that removing any of the edges in this cycle would mean that the remaining edges would still connect all the vertices, but would have a total weight that's less by the weight of the edge that was removed. This would contradict the minimality of the total weight of the subset of vertices. Since the subset of edges forms a tree, and has minimal total weight, it must also be a minimum spanning tree.

Consider a graph with three vertices and the following set of edges:

- Edge 1: Weight 2
- Edge 2: Weight -1
- Edge 3: Weight 3

In this case, the subset of edges {Edge 1, Edge 2} connects all vertices (forms a cycle) but does not form a tree, even though the weights are not all positive. The presence of a nonpositive weight in this example allows for cycles with a lower total weight than some tree configurations. Therefore, the conclusion that a subset with the minimum total weight connecting all vertices must be a tree does not hold when nonpositive weights are allowed.

> 23.1-8 Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

Suppose that L' is another sorted list of edge weights of a minimum spanning tree. If $L' \neq L$, there must be a first edge (u, v) in T or T' which is of smaller weight than the corresponding edge (x, y) in the other set. Without loss of generality, assume (u, v) is in T.

Let C be the graph obtained by adding (u, v) to L'. Then we must have introduced a cycle. If there exists an edge on that cycle which is of larger weight than (u, v), we can remove it to obtain a tree C' of weight strictly smaller than the weight of T', contradicting the fact that T' is a minimum spanning tree.

Thus, every edge on the cycle must be of lesser or equal weight than (u, v). Suppose that every edge is of strictly smaller weight. Remove (u, v) from T to disconnect it into two components. There must exist some edge besides (u, v) on the cycle which would connect these, and since it has smaller weight we can use that edge instead to create a spanning tree with less weight than T, a contradiction. Thus, some edge on the cycle has the same weight as (u, v). Replace that edge by (u, v). The corresponding lists L and L' remain unchanged since we have swapped out an edge of equal weight, but the number of edges which T and T' have in common has increased by 1.

If we continue in this way, eventually they must have every edge in common, contradicting the fact that their edge weights differ somewhere. Therefore all minimum spanning trees have the same sorted list of edge weights.

> 23.2-2 Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

At each step of the algorithm we will add an edge from a vertex in the tree created so far to a vertex not in the tree, such that this edge has minimum weight. Thus, it will be useful to know, for each vertex not in the tree, the edge from that vertex to some vertex in the tree of minimal weight. We will store this information in an array A, where $A[u] = (v, w)$ if w is the weight of (u, v) and is minimal among the weights of edges from u to some vertex v in the tree built so far. We'll use $A[u].1$ to access v and $A[u].2$ to access w.

```
PRIM-ADJ(G, w, r)
    initialize A with every entry = (NIL, ∞)
    T = {r}
    for i = 1 to V
        if Adj[r, i] != 0
            A[i] = (r, w(r, i))
    for each u in V - T
        k = min(A[i].2)
        T = T U {k}
        k.π = A[k].1
        for i = 1 to V
            if Adf[k, i] != 0 and Adj[k, i] < A[i].2
                A[i] = (k, Adj[k, i])
```

> 23.2-5 Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

For the first case, we can use a van Emde Boas tree to improve the time bound to $O(E \lg \lg V)$. Comparing to the Fibonacci heap implementation, this improves the asymptotic running time only for sparse graphs, and it cannot improve the running time polynomially. An advantage of this implementation is that it may have a lower overhead.

For the second case, we can use a collection of doubly linked lists, each corresponding to an edge weight. This improves the bound to $O(E)$.

problems

> 23-4 In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.
>
> **a.**

```
MAYBE-MST-A(G, w)
sort the edges into nonincreasing order of edge weights w
T = E
for each edge e, taken in nonincreasing order by weight
  if T - {e} is a connected graph
      T = T - {e}
return T
```

b.

```
MAYBE-MST-B(G, w)
T = Ø
for each edge e, taken in arbitrary order
  if T U {e} has no cycles
      T = T U {e}
return T
```

c.

```
MAYBE-MST-C(G, w)
T = Ø
for each edge e, taken in arbitrary order
  T = T U {e}
  if T has a cycle c
      let e' be a maximum-weight edge on c
      T = T - {e}
return T
```

a. This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e. By exercise 23.1-5, there is a minimum spanning tree on G with edge e removed.

To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - e$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E))$. This dominates the running time, so the total time is $O(E^2)$.

b. This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the eges be (a, b), (b, c), and (c, a) with weights $3, 2$, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

An efficient implementation will use disjoint sets to keep track of connected components, as in MST-REDUCE in problem 23-2. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKESET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(E\alpha(V))$.

c. This does return an MST. To see this, we simply quote the result from exercise 23.1-5. The only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.