Chapter 16

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 16.1-1

The given algorithm would just stupidly compute the minimum of the $O(n)$ numbers or return zero depending on the size of S_{ij}. There are a possible number of subproblems that is $O(n^2)$ since we are selecting i and j so that $1 \leq i \leq j \leq n$. So, the runtime would be $O(n^3)$.

Exercise 16.1-2

This becomes exactly the same as the original problem if we imagine time running in reverse, so it produces an optimal solution for essentially the same reasons. It is greedy because we make the best looking choice at each step.

Exercise 16.1-3

As a counterexample to the optimality of greedily selecting the shortest, suppose our activity times are $\{(1,9),(8,11),(10,20)\}$ then, picking the shortest first, we have to eliminate the other two, where if we picked the other two instead, we would have two tasks not one.

As a counterexample to the optimality of greedily selecting the task that conflicts with the fewest remaining activities, suppose the activity times are $\{(-1,1),(2,5),(0,3),(0,3),(0,3),(4,7),(6,9),(8,11),(8,11),(8,11),(10,12)\}$. Then, by this greedy strategy, we would first pick $(4,7)$ since it only has a two conflicts. However, doing so would mean that we would not be able to pick the only optimal solution of $(-1,1),(2,5),(6,9),(10,12)$.

As a counterexample to the optimality of greedily selecting the earliest start times, suppose our activity times are $\{(1,10),(2,3),(4,5)\}$. If we pick the earliest start time, we will only have a single activity, $(1,10)$, whereas the optimal solution would be to pick the two other activities.

Exercise 16.1-4

Maintain a set of free (but already used) lecture halls F and currently busy lecture halls B. Sort the classes by start time. For each new start time which you encounter, remove a lecture hall from F, schedule the class in that room,

and add the lecture hall to B. If F is empty, add a new, unused lecture hall to F. When a class finishes, remove its lecture hall from B and add it to F. Why this is optimal: Suppose we have just started using the m^{th} lecture hall for the first time. This only happens when ever classroom ever used before is in B. But this means that there are m classes occurring simultaneously, so it is necessary to have m distinct lecture halls in use.

Exercise 16.1-5

Run a dynamic programming solution based off of the equation (16.2) where the second case has "1" replaced with "v_k". Since the subproblems are still indexed by a pair of activities, and each calculation requires taking the minimum over some set of size $\leq |S_{ij}| \in O(n)$. The total runtime is bounded by $O(n^3)$.

Exercise 16.2-1

A optimal solution to the fractional knapsack is one that has the highest total value density. Since we are always adding as much of the highest value density we can, we are going to end up with the highest total value density. Suppose that we had some other solution that used some amount of the lower value density object, we could substitute in some of the higher value density object meaning our original solution could not have been optimal.

Exercise 16.2-2

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:

Exercise 16.2-3

At each step just pick the lightest (and most valuable) item that you can pick. To see this solution is optimal, suppose that there were some item j that we included but some smaller, more valuable item i that we didn't. Then, we could replace the item j in our knapsack with the item i. it will definitely fit

Algorithm 1 0-1 Knapsack(n,W)

```
1: Initialize an n + 1 by W + 1 table K
2: for j = 1 to W do
3:     K[0, j] = 0
4: end for
5: for i = 1 to n do
6:     K[i, 0] = 0
7: end for
8: for i = 1 to n do
9:     for j = 1 to W do
10:        if j < i.weight then
11:            K[i, j] = K[i − 1, j]
12:        end if
13:        K[i, j] = max(K[i − 1, j], K[i − 1, j − i.weight] + i.value)
14:    end for
15: end for
```

because i is lighter, and it will also increase the total value because i is more valuable.

Exercise 16.2-4

The greedy solution solves this problem optimally, where we maximize distance we can cover from a particular point such that there still exists a place to get water before we run out. The first stop is at the furthest point from the starting position which is less than or equal to m miles away. The problem exhibits optimal substructure, since once we have chosen a first stopping point p, we solve the subproblem assuming we are starting at p. Combining these two plans yields an optimal solution for the usual cut-and-paste reasons. Now we must show that this greedy approach in fact yields a first stopping point which is contained in some optimal solution. Let O be any optimal solution which has the professor stop at positions $o_1, o_2, \ldots, o_k$. Let g_1 denote the furthest stopping point we can reach from the starting point. Then we may replace o_1 by g_2 to create a modified solution G, since $o_2 - o_1 < o_2 - g_1$. In other words, we can actually make it to the positions in G without running out of water. Since G has the same number of stops, we conclude that g_1 is contained in some optimal solution. Therefore the greedy strategy works.

Exercise 16.2-5

Consider the leftmost interval. It will do no good if it extends any further left than the leftmost point, however, we know that it must contain the leftmost point. So, we know that it's left hand side is exactly the leftmost point. So, we just remove any point that is within a unit distance of the left most point since

they are contained in this single interval. Then, we just repeat until all points are covered. Since at each step there is a clearly optimal choice for where to put the leftmost interval, this final solution is optimal.

Exercise 16.2-6

First compute the value of each item, defined to be it's worth divided by its weight. We use a recursive approach as follows: Find the item of median value, which can be done in linear time as shown in chapter 9. Then sum the weights of all items whose value exceeds the median and call it M. If M exceeds W then we know that the solution to the fractional knapsack problem lies in taking items from among this collection. In other words, we're now solving the fractional knapsack problem on input of size $n/2$. On the other hand, if the weight doesn't exceed W, then we must solve the fractional knapsack problem on the input of $n/2$ low-value items, with maximum weight $W - M$. Let $T(n)$ denote the runtime of the algorithm. Since we can solve the problem when there is only one item in constant time, the recursion for the runtime is $T(n) = T(n/2) + cn$ and $T(1) = d$, which gives runtime of $O(n)$.

Exercise 16.2-7

Since an identical permutation of both sets doesn't affect this product, suppose that A is sorted in ascending order. Then, we will prove that the product is maximized when B is also sorted in ascending order. To see this, suppose not, that is, there is some $i < j$ so that $a_i < a_j$ and $b_i > b_j$. Then, consider only the contribution to the product from the indices i and j. That is, $a_i^{b_i} a_j^{b_j}$, then, if we were to swap the order of b_1 and b_j, we would have that contribution be $a_i^{b_j} a_j^{b_i}$. we can see that this is larger than the previous expression because it differs by a factor of $(\frac{a_j}{a_i})^{b_i - b_j}$ which is bigger than one. So, we couldn't of maximized the product with this ordering on B.

Exercise 16.3-1

If we have that $x.freq = b.freq$, then we know that b is tied for lowest frequency. In particular, it means that there are at least two things with lowest frequency, so $y.freq = x.freq$. Also, since $x.freq \leq a.freq \leq b.freq = x.freq$, we must have $a.freq = x.freq$.

Exercise 16.3-2

Let T be a binary tree corresponding to an optimal prefix code and suppose that T is not full. Let node n have a single child x. Let T' be the tree obtained by removing n and replacing it by x. Let m be a leaf node which is a descendant of x. Then we have

$$cost(T') \leq \sum_{c \in C \setminus \{m\}} c.freq \cdot d_T(c) + m.freq(d_T(m) - 1) < \sum_{c \in C} c.freq \cdot d_T(c) = cost(T)$$

which contradicts the fact that T was optimal. Therefore every binary tree corresponding to an optimal prefix code is full.

Exercise 16.3-3

An optimal Huffman code would be

$$\begin{aligned} 0000000 &\to a \\ 0000001 &\to b \\ 000001 &\to c \\ 00001 &\to d \\ 0001 &\to e \\ 001 &\to f \\ 01 &\to g \\ 1 &\to h \end{aligned}$$

This generalizes to having the first n Fibonacci numbers as the frequencies in that the $k < n$th most frequent letter has codeword $0^{k-1}1$, and the nth most frequent letter having codeword 0^{n-1}. To see this holds, we will prove the recurrence

$$\sum_{i=0}^{n-1} F(i) = F(n+1) - 1$$

This will show that we should join together the letter with frequency $F(n)$ with the result of joining together the letters with smaller frequencies. We will prove it by induction. For $n = 1$ is is trivial to check. Now, suppose that we have $n - 1 \geq 1$, then,

$$F(n+1) - 1 = F(n) + F(n-1) - 1 = F(n-1) + \sum_{i=0}^{n-2} F(i) = \sum_{i=0}^{n-1} F(i)$$

See also Lemma 19.2.

To use this fact, to show the desired Huffman code is optimal, we claim that as we are greedily combining nodes, that at each stage we can maintain one node which contains all of the least frequent letters. Initially, this consists of just the least frequent letter. Then, at stage k, inductively, we assume that it contains the k least frequent letters. This means that it has weight $\sum_{i=0}^{k-1} F(i) = F(k-1) - 1$. All of the other nodes at this stage have weights $\{F(k), F(k+1), ...F(n-1)\}$. So, clearly the two lowest weight nodes are this

node containing the k least frequent letters and the node containing the k least frequent letter. Therefore, the greedy algorithm tells us that we should combine those nodes leaving us with a node containing the $k+1$ least frequent letters for stage $k+1$, completing the induction. Of course, the code that we have given is not uniquely optimal, because left and right children (represented by 0 and 1 respectively) are an artificial choice, we could assume that at each stage we are adding in the singleton as the 1 child of the new parent, getting us our code.

Exercise 16.3-4

Let x be a leaf node. Then $x.freq$ is added to the cost of each internal node which is an ancestor of x exactly once, so its total contribution to the new way of computing cost is $x.freq \cdot d_T(x)$, which is the same as its old contribution. Therefore the two ways of computing cost are equivalent.

Exercise 16.3-5

We construct this codeword with monotonically increasing lengths by always resolving ties in terms of which two nodes to join together by joining together those with the two latest occurring earliest elements. We will show that the ending codeword has that the least frequent words are all having longer codewords. Suppose to a contradiction that there were two words, w_1 and w_2 so that w_1 appears more frequently, but has a longer codeword. This means that it was involved in more merge operation than w_2 was. However, since we are always merging together the two sets of words with the lowest combined frequency, this would contradict the fact that w_1 has a higher frequency than w_2.

Exercise 16.3-6

First observe that any full binary tree has exactly $2n-1$ nodes. We can encode the structure of our full binary tree by performing a preorder traversal of T. For each node that we record in the traversal, write a 0 if it is an internal node and a 1 if it is a leaf node. Since we know the tree to be full, this uniquely determines its structure. Next, note that we can encode any character of C in $\lceil \lg n \rceil$ bits. Since there are n characters, we can encode them in order of appearance in our preorder traversal using $n\lceil \lg n \rceil$ bits.

Exercise 16.3-7

Instead of grouping together the two with lowest frequency into pairs that have the smallest total frequency, we will group together the three with lowest frequency in order to have a final result that is a ternary tree. The analysis of optimality is almost identical to the binary case. We are placing the symbols of lowest frequency lower down in the final tree and so they will have longer codewords than the more frequently occurring symbols

Exercise 16.3-8

For any 2 characters, the sum of their frequencies exceeds the frequency of any other character, so initially Huffman coding makes 128 small trees with 2 leaves each. At the next stage, no internal node has a label which is more than twice that of any other, so we are in the same setup as before. Continuing in this fashion, Huffman coding builds a complete binary tree of height $\lg(256) = 8$, which is no more efficient than ordinary 8-bit length codes.

Exercise 16.3-9

If every possible character is equally likely, then, when constructing the Huffman code, we will end up with a complete binary tree of depth 7. This means that every character, regardless of what it is will be represented using 8 bits. This is exactly as many bits as was originally used to represent those characters, so the total length of the file will not decrease at all.

Exercise 16.4-1

The first condition that S is a finite set is a given. To prove the second condition we assume that $k \geq 0$, this gets us that $\mathcal{I}_k$ is nonempty. Also, to prove the hereditary property, suppose $A \in \mathcal{I}_k$ this means that $|A| \leq k$. Then, if $B \subseteq A$, this means that $|B| \leq |A| \leq k$, so $B \in \mathcal{I}_k$. Lastly, we prove the exchange property by letting $A, B \in \mathcal{I}_k$ be such that $|A| < |B|$. Then, we can pick any element $x \in B \setminus A$, then, $|A \cup \{x\}| = |A| + 1 \leq |B| \leq k$, so, we can extend A to $A \cup \{x\} \in \mathcal{I}_k$.

Exercise 16.4-2

Let $c_1, \ldots, c_m$ be the columns of T. Suppose $C = \{c_{i_1}, \ldots, c_{i_k}\}$ is dependent. Then there exist scalars $d_1, \ldots, d_k$ not all zero such that $\sum_{j=1}^{k} d_j c_{i_j} = 0$. By adding columns to C and assigning them to have coefficient 0 in the sum, we see that any superset of C is also dependent. By contrapositive, any subset of an independent set must be independent. Now suppose that A and B are two independent sets of columns with $|A| > |B|$. If we couldn't add any column of A to be whilst preserving independence then it must be the case that every element of A is a linear combination of elements of B. But this implies that B spans a $|A|$-dimensional space, which is impossible. Therefore our independence system must satisfy the exchange property, so it is in fact a matroid.

Exercise 16.4-3

Condition one of being a matroid is still satisfied because the base set hasn't changed. Next we show that $\mathcal{I}'$ is nonempty. Let A be any maximal element of $\mathcal{I}$ then, we have that $S - A \in \mathcal{I}'$ because $S - (S - A) = A \subseteq A$ which is maximal in $\mathcal{I}$. Next we show the hereditary property, suppose that $B \subseteq A \in \mathcal{I}'$, then,

there exists some $A' \in \mathcal{I}$ so that $S - A \subseteq A'$, however, $S - B \supseteq S - A \subseteq A'$ so $B \in \mathcal{I}'$.

Lastly we prove the exchange property. That is, if we have $B, A \in \mathcal{I}'$ and $|B| < |A|$ we can find an element x in $A - B$ to add to B so that it stays independent. We will split into two cases.

Our first case is that $|A| = |B| + 1$. We clearly need to select x to be the single element in $A - B$. Since $S - B$ contains a maximal independent set

Our second case is if the first case does not hold. Let C be a maximal independent set of $\mathcal{I}$ contained in $S - A$. Pick an aribitrary set of size $|C| - 1$ from some maximal independent set contained in $S - B$, call it D. Since D is a subset of a maximal independent set, it is also independent, and so, by the exchange property, there is some $y \in C - D$ so that $D \cup \{y\}$ is a maximal independent set in $\mathcal{I}$. Then, we select x to be any element other than y in $A - B$. Then, $S - (B \cup \{x\})$ will still contain $D \cup \{y\}$. This means that $B \cup \{x\}$ is independent in $(I)'$

Exercise 16.4-4

Suppose $X \subset Y$ and $Y \in \mathcal{I}$. Then $(X \cap S_i) \subset (Y \cap S_i)$ for all i, so $|X \cap S_i| \leq |Y \cap S_i| \leq 1$ for all $1 \leq i \leq k$. Therefore $\mathcal{M}$ is closed under inclusion.

Now Let $A, B \in \mathcal{I}$ with $|A| = |B| + 1$. Then there must exist some j such that $|A \cap S_j| = 1$ but $B \cap S_j| = 0$. Let $a = A \cap S_j$. Then $a \notin B$ and $|(B \cup \{a\}) \cap S_j| = 1$. Since $|(B \cup \{a\}) \cap S_i| = |B \cap S_i|$ for all $i \neq j$, we must have $B \cup \{a\} \in \mathcal{I}$. Therefore $\mathcal{M}$ is a matroid.

Exercise 16.4-5

Suppose that W is the largest weight that any one element takes. Then, define the new weight function $w_2(x) = 1 + W - w(x)$. This then assigns a strictly positive weight, and we will show that any independent set that that has maximum weight with respect to w_2 will have minimum weight with respect to w. Recall Theorem 16.6 since we will be using it, suppose that for our matriod, all maximal independent sets have size S. Then, suppose M_1 and M_2 are maximal independent sets so that M_1 is maximal with respect to w_2 and M_2 is minimal with respect to w. Then, we need to show that $w(M_1) = w(M_2)$. Suppose not to achieve a contradiction, then, by minimality of M_2, $w(M_1) > w(M_2)$. Rewriting both sides in terms of w_2, we have $w_2(M_2) - (1 + W)S > w_2(M_1) - (1+W)S$, so, $w_2(M_2) > w_2(M_1)$. This however contradicts maximality of M_1 with respect to w_2. So, we must have that $w(M_1) = w(M_2)$. So, a maximal independent set that has the largest weight with respect to w_2 also has the smallest weight with respect to w.

Exercise 16.5-1

With the requested substitution, the instance of the problem becomes

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	10	20	30	40	50	60	70

We begin by just greedily constructing the matroid, adding the most costly to leave incomplete tasks first. So, we add tasks 7,6,5,4,3. Then, in order to schedule tasks 1 or 2 we need to leave incomplete more important tasks. So, our final schedule is $\langle 5, 3, 4, 6, 7, 1, 2\rangle$ to have a total penalty of only $w_1+w_2 = 30$.

Exercise 16.5-2

Create an array B of length n containing zeros in each entry. For each element $a \in A$, add 1 to $B[a.deadline]$. If $B[a.deadline] > a.deadline$, return that the set is not independent. Otherwise, continue. If successfully examine every element of A, return that the set is independent.

Problem 16-1

a. Always give the highest denomination coin that you can without going over. Then, repeat this process until the amount of remaining change drops to 0.

b. Given an optimal solution $(x_0, x_1, \ldots, x_k)$ where x_i indicates the number of coins of denomination c^i. We will first show that we must have $x_i < c$ for every $i < k$. Suppose that we had some $x_i \geq c$, then, we could decrease x_i by c and increase x_{i+1} by 1. This collection of coins has the same value and has $c-1$ fewer coins, so the original solution must of been non-optimal. This configuration of coins is exactly the same as you would get if you kept greedily picking the largest coin possible. This is because to get a total value of V, you would pick $x_k = \lfloor Vc^{-k} \rfloor$ and for $i < k$, $x_i \lfloor (V mod c^{i+1})c^{-i} \rfloor$. This is the only solution that satisfies the property that there aren't more than c of any but the largest denomination because the coin amounts are a base c representation of $V mod c^k$.

c. Let the coin denominations be $\{1, 3, 4\}$, and the value to make change for be 6. The greedy solution would result in the collection of coins $\{1, 1, 4\}$ but the optimal solution would be $\{3, 3\}$.

d. See algorithm MAKE-CHANGE(S,v) which does a dynamic programming solution. Since the first forloop runs n times, and the inner for loop runs k times, and the later while loop runs at most n times, the total running time is $O(nk)$.

Problem 16-2

a. Order the tasks by processing time from smallest to largest and run them in that order. To see that this greedy solution is optimal, first observe that the problem exhibits optimal substructure: if we run the first task in an optimal

Algorithm 2 MAKE-CHANGE(S,v)

```
Let numcoins and coin be empty arrays of length v, and any any attempt to
access them at indices in the range −max(S), −1 should return ∞
for i from 1 to v do
    bestcoin =nil
    bestnum = ∞
    for c in S do
        if numcoins[i − c] + 1 < bestnum then
            bestnum = numcoins[i-c]
            bestcoin = c
        end if
    end for
    numcoins[i] = bestnum
    coin[i] = bestcoin
end for
let change be an empty set
iter = v
while iter > 0 do
    add coin[iter] to change
    iter = iter − coin[iter]
end while
return change
```

solution, then we obtain an optimal solution by running the remaining tasks in a way which minimizes the average completion time. Let O be an optimal solution. Let a be the task which has the smallest processing time and let b be the first task run in O. Let G be the solution obtained by switching the order in which we run a and b in O. This amounts reducing the completion times of a and the completion times of all tasks in G between a and b by the difference in processing times of a and b. Since all other completion times remain the same, the average completion time of G is less than or equal to the average completion time of O, proving that the greedy solution gives an optimal solution. This has runtime $O(n \lg n)$ because we must first sort the elements.

b. Without loss of generality we my assume that every task is a unit time task. Apply the same strategy as in part (a), except this time if a task which we would like to add next to the schedule isn't allowed to run yet, we must skip over it. Since there could be many tasks of short processing time which have late release time, the runtime becomes $O(n^2)$ since we might have to spend $O(n)$ time deciding which task to add next at each step.

Problem 16-3

a. First, suppose that a set of columns is not linearly independent over $\mathbb{F}_2$ then, there is some subset of those columns, say S so that a linear combination of S is 0. However, over $\mathbb{F}_2$, since the only two elements are 1 and 0, a linear combination is a sum over some subset. Suppose that this subset is S', note that it has to be nonempty because of linear dependence. Now, consider the set of edges that these columns correspond to. Since the columns had their total incidence with each vertex 0 in $\mathbb{F}_2$, it is even. So, if we consider the subgraph on these edges, then every vertex has a even degree. Also, since our S' was nonempty, some component has an edge. Restrict our attention to any such component. Since this component is connected and has all even vertex degrees, it contains an Euler Circuit, which is a cycle.

   Now, suppose that our graph had some subset of edges which was a cycle. Then, the degree of any vertex with respect to this set of edges is even, so, when we add the corresponding columns, we will get a zero column in $\mathbb{F}_2$.

   Since sets of linear independent columns form a matroid, by problem 16.4-2, the acyclic sets of edges form a matroid as well.

b. One simple approach is to take the highest weight edge that doesn't complete a cycle. Another way to phrase this is by running Kruskal's algorithm (see Chapter 23) on the graph with negated edge weights.

c. Consider the digraph on $[3]$ with the edges $(1,2),(2,1),(2,3),(3,2),(3,1)$ where (u,v) indicates there is an edge from u to v. Then, consider the two acyclic subsets of edges $B = (3,1),(3,2),(2,1)$ and $A = (1,2),(2,3)$. Then, adding any edge in $B - A$ to A will create a cycle. So, the exchange property is violated.

d. Suppose that the graph contained a directed cycle consisting of edges corresponding to columns S. Then, since each vertex that is involved in this cycle has exactly as many edges going out of it as going into it, the rows corresponding to each vertex will add up to zero, since the outgoing edges count negative and the incoming vertices count positive. This means that the sum of the columns in S is zero, so, the columns were not linearly independent.

e. There is not a perfect correspondence because we didn't show that not containing a directed cycle means that the columns are linearly independent, so there is not perfect correspondence between these sets of independent columns (which we know to be a matriod) and the acyclic sets of edges (which we know not to be a matroid).

Problem 16-4

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 a_j 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 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 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 $O(n\alpha(n))$.

Problem 16-5

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 $O(m + n)$.

Algorithm 3 SCHEDULING-VARIATIONS(A)

```
1: Initialize an array D of size n.
2: for i = 1 to n do
3:     a_i.time = a_i.deadline
4:     if D[a_i.deadline] ≠ NIL then
5:         y = FIND-SET(D[a_i.deadline])
6:         a_i.time = y.low − 1
7:     end if
8:     x = MAKE-SET(a_i)
9:     D[a_i.time] = x
10:    x.low = x.high = a_i.time
11:    if D[a_i.time − 1] ≠ NIL then
12:        UNION(D[a_i.time − 1], D[a_i.time])
13:    end if
14:    if D[a_i.time + 1] ≠ NIL then
15:        UNION(D[a_i.time], D[a_i.time + 1])
16:    end if
17: end for
```

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.