16 Greedy Algorithms

16.1 An activity-selection problem

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 $\text{GREEDY-ACTIVITY-SELECTOR}$.

```
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)
```

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

16.1-2

> Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

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.

16.1-3

> Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

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.

16.1-4

> Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.
>
> (This problem is also known as the ***interval-graph coloring problem***. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

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. This is optimal for following reason, suppose we have just started using the mth 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.

16.1-5

> Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Easy and straightforward solution is to run a dynamic programming solution based on 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 $\le |S_{ij}| \in O(n)$. The total runtime is bounded by $O(n^3)$.

However, if we are cunning a little, we can be more efficient and give the algorithm which runs in $O(n \log n)$.

INPUT: n activities with values.

IDEA OF ALGORITHM:

1. Sort input vector of activities according their finish times in ascending order. Let us denote the activities in this sorted vector by $(a_0, a_1, \dots, a_{n-1})$.
2. For each $0 \le i < n$ construct partial solution S_i. By a partial solution S_i, we mean a solution to the problem but considering only activities with indexes lower or equal to i. Remember value of each partial solution.
3. Clearly $S_0 = \{a_0\}$.
4. We can construct S_{i+1} as follows. Possible values of S_{i+1} is either S_i or the solution obtained by joining the activity a_{i+1} with partial solution S_j where $j < i + 1$ is the index of activity such that a_j is compatible with a_{i+1} but a_{j+1} is not compatible with a_{i+1}. Pick the one of these two possible solutions, which has greater value. Ties can be resolved arbitrarily.
5. Therefore we can construct partial solutions in order $S_0, S_1, \dots, S_{n-1}$ using (3) for S_0 and (4) for all the others.
6. Give S_{n-1} as the solution for problem.

ANALYSIS OF TIME COMPLEXITY:

- Sorting of activities can be done in $O(n \log n)$ time.
- Finding the value of S_0 is in $O(1)$.

- Any S_{i+1} can be found in $O(\log n)$. It is thanks to the fact that we have properly sorted activities. Therefore we can for each $i+1$ find the proper j in $O(\log n)$ using the binary search. If we have the proper j, the rest can be done in $O(1)$.
- Therefore, we have $O(n \log n)$ time for constructing of all S_i's.

IMPLEMENTATION DETAILS:

- Use the dynamic programming.
- It is important not to remember too much for each S_i. Do not construct S_i's directly (you can end up in $\Omega(n^2)$ time if you do so). For each S_i it is sufficient to remember:
  - it's value
  - whether or not it includes the activity a_i
  - the value of j (from (4)).
- Using these information obtained by the run of described algorithm you can reconstruct the solution in $O(n)$ time, which does not violate final time complexity.

PROOF OF CORRECTNESS: (sketched)

- Clearly, $S_0 = \{a_0\}$.
- For S_{i+1} we argue by (4). Partial solution S_{i+1} either includes the activity a_{i+1} or doesn't include it, there is no third way.
  - If it does not include a_{i+1}, then clearly $S_{i+1} = S_i$.
  - If it includes a_{i+1}, then S_{i+1} consists of a_{i+1} and partial solution which uses all activities compatible with a_{i+1} with indexes lower than $i+1$. Since activities are sorted according their finish times, activities with indexes j and lower are compatible and activities with index $j+1$ and higher up to $i+1$ are not compatible. We do not consider all the other activities for S_{i+1}. Therefore setting $S_{i+1} = \{a_{i+1}\} \cup S_j$ gives correct answer in this case. The fact that we need S_j and not some other solution for activities with indexes up to j can be easily shown by the standard cut-and-paste argument.
- Since for S_{n-1} we consider all of the activities, it is actually the solution of the problem.

16.2 Elements of the greedy strategy

16.2-1

> 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 $W \ge w_n$.

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

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

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.

16.2-2

> 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 $j - i.\text{weight}$ 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)
```

16.2-3

> Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

Suppose in an optimal solution we take an item with v_1, w_1, and drop an item with v_2, w_2, and $w_1 > w_2$, $v_1 < v_2$, we can substitute 1 with 2 and get a better solution. Therefore we should always choose the items with the greatest values.

16.2-4

> Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations.
>
> The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

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 $o_1, o_2, \dots, o_k$. 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.

16.2-5

> Describe an efficient algorithm that, given a set $\{x_1, x_2, \dots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

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.

16.2-6 ⋆

> Show how to solve the fractional knapsack problem in $O(n)$ time.

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 $O(n)$.

16.2-7

> Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Since an idential 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_i 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 $\left(\frac{a_j}{a_i}\right)^{b_i - b_j}$ which is bigger than one. So, we couldn't of maximized the product with this ordering on B.

16.3 Huffman codes

16.3-1

> Explain why, in the proof of Lemma 16.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

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 $x.freq \le a.freq \le b.freq = x.freq$, we must have $a.freq = x.freq$.

16.3-2

> Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

Let T be a binary tree that is not full. T represents a binary prefix code for a file composed of characters from alphabet C, where $c \in C$, $f(c)$ is th number of occurrences of c in the file. The cost of tree T, or the number of bits in the encoding, is $\sum_{c \in C} d_T(c) \cdot f(c)$, where $d_T(c)$ is the depth of character c in tree T.

Let N be a node of greatest depth that has exactly one child. If N is the root of T, N can be removed and the deepth of each node reduced by one, yielding a tree representing the same alphabet with a lower cost. This mean the original code was not optimal.

Otherwise, let M be the parent of N, let T_1 be the (possibly non-existent) sibling of N, and let T_2 be the subtree rooted at the child of N. Replace M by N, making T_1 be the children of N. If T_1 is empty, repeat the process. We have a new prefix code of lower cost, so the original was not optimal.

16.3-3

> What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?
>
> $$a:1 \quad b:1 \quad c:2 \quad d:3 \quad e:5 \quad f:8 \quad g:13 \quad h:21$$
>
> Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

a	1111111
b	1111110
c	111110
d	11110
e	1110
f	110
g	10
h	0

GENERALIZATION

In what follows we use a_i to denote i-th Fibonacci number. To avoid any confusiion we stress that we consider Fibonacci's sequence beginning 1, 1, i.e. $a_1 = a_2 = 1$.

Let us consider a set of n symbols $\Sigma = \{c_i \mid 1 \le i \le n\}$ such that for each i we have $c_i.freq = a_i$. We shall prove that the Huffman code for this set of symbols given by the run of algorithm HUFFMAN from CLRS is the following code:

- $code(c_n) = 0$
- $code(c_{i-1}) = 1code(c_i)$ for $2 \le i \le n-1$ (i.e. we take a code for symbol c_i and add 1 to the beginning)
- $code(c_1) = 1^{n-1}$

By $code(c)$ we mean the codeword assigned to the symbol c_i by the run of HUFFMAN(Σ) for any $c \in \Sigma$.

First we state two technical claims which can be easily proven using the proper induction. Following good manners of our field we leave the proofs to the reader 😃

- (HELPFUL CLAIM 1) $(\forall k \in \mathbb{N})\ \sum_{i=1}^{k} a_i = a_{k+2} - 1$
- (HELPFUL CLAIM 2) Let z be an inner node of tree T constructed by the algorithm HUFFMAN. Then $z.freq$ is sum of frequencies of all leafs of the subtree of T rooted in z.

Consider tree T_n inductively defined by

- $T_2.left = c_2$, $T_2.right = c_1$ and $T_2.freq = c_1.freq + c_2.freq = 2$
- $(\forall i; 3 \le i \le n)\ T_i.left = c_i$, $T_i.right = T_{i-1}$ and $T_i.freq = c_i.freq + T_{i-1}.freq$

We shall prove that T_n is the tree produced by the run of HUFFMAN(Σ).

KEY CLAIM: T_{i+1} is exactly the node z constructed in i-th run of the for-cycle of HUFFMAN(Σ) and the content of the priority queue Q just after i-th run of the for-cycle is exactly $Q = (a_{i+2}, T_{i+1}, a_{i+3}, \dots, a_n)$ with a_{i+2} being the minimal element for each $1 \le i < n$. (Since we prefer not to overload our formal notation we just note that for $i = n - 1$ we claim that $Q = (T_n)$ and our notation grasp this fact in a sense.)

PROOF OF KEY CLAIM by induction on i.

- for $i = 1$ we see that the characters with lowest frequencies are exactly c_1 and c_2, thus obviously the algorithm HUFFMAN(Σ) constructs T_2 in the first run of its for-cycle. Also it is obvious that just after this run of the for-cycle we have $Q = (a_3, T_2, a_4, \dots, a_n)$.
- for $2 \le i < n$ we suppose that our claim is true for all $j < i$ and prove the claim for i. Since the claim is true for $i - 1$, we know that just before $i - th$ execution of the for-cycle we have the following content of the priority queue $Q = (a_{i+1}, T_i, a_{i+2}, \dots, a_n)$. Thus line 5 of HUFFMAN extracts a_{i+1} and sets $z.left = a_{i+1}$ and line 6 of HUFFMAN extracts T_i and sets $z.right = T_i$. Now we can see that indeed z is exactly T_{i+1}. Using (CLAIM 2) and observing the way T_{i+1} is defined we get that $z.freq = T_{i+1}.freq = \sum_{i=1}^{i+1} a_i$. Thus using (CLAIM 1) one can see that $a_{i+2} < T_{i+1}.freq < a_{i+3}$.
  Therefore for the content of the priority queue Q just after the i-th execution of the for-cycle we have $Q = (a_{i+2}, T_{i+2}, a_{i+3}, \dots, a_n)$.

KEY CLAIM tells us that just after the last execution of the for-cycle we have $Q = (T_n)$ and therefore the line 9 of HUFFMAN returns T_n as the result. One can easily see that the code given in the beginning is exactly the code which corresponds to the code-tree T_n.

16.3-4

> Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

Let tree be a full binary tree with n leaves. Apply induction hypothesis on the number of leaves in T. When $n = 2$ (the case $n = 1$ is trivially true), there are two leaves x and y with the same parent z, then the cost of T is

$$\begin{aligned} B(T) &= f(x)d_T(x) + f(y)d_T(y) \\ &= f(x) + f(y) && \text{since } d_T(x) = d_T(y) = 1 \\ &= f(\text{child}_1 \text{ of } z) + f(\text{child}_2 \text{ of } z). \end{aligned}$$

Thus, the statement of theorem is true. Now suppose $n > 2$ and also suppose that theorem is true for trees on $n - 1$ leaves. Let c_1 and c_2 are two sibling leaves in T such that they have the same parent p. Letting T' be the tree obtained by deleting c_1 and c_2, by induction we know that

$$\begin{aligned} B(T) &= \sum_{\text{leaves } l' \in T'} f(l')d_T(l') \\ &= \sum_{\text{internal nodes } i' \in T'} f(\text{child}_1 \text{ of } i') + f(\text{child}_2 \text{ of } i'). \end{aligned}$$

Using this information, calculates the cost of T.

$$\begin{aligned} B(T) &= \sum_{\text{leaves } l \in T} f(l)d_T(l) \\ &= \sum_{l \ne c_1, c_2} f(l)d_T(l) + f(c_1)d_T(c_1) - 1 + f(c_2)d_T(c_2) - 1 + f(c_1) + f(c_2) \\ &= \sum_{\text{internal nodes } i' \in T'} f(\text{child}_1 \text{ of } i') + f(\text{child}_2 \text{ of } i') + f(c_1) + f(c_2) \\ &= \sum_{\text{internal nodes } i \in T} f(\text{child}_1 \text{ of } i) + f(\text{child}_1 \text{ of } i). \end{aligned}$$

Thus the statement is true.

16.3-5

> Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

Little formal-mathematical note here: We are required to prove existence of an optimal code with some property. Therefore we are required also to show, that some optimal code exists. It is trivial in this case, since we know that the code produced by a run of Huffman's algorithm produce one such code for us. However, it is good to be aware of this. Proving just the implication "if a code is optimal then it has the desired property" doesn't suffice.

OK, now we are ready to prove the already mentioned implication "if a code is optimal then it has the desired property". Main idea of our proof is that if the code violates desired property, then we find two symbols which violate the property and 'fix the code'. For the formal proof we go as follows.

Suppose that we have an alphabet $C = a_1, \ldots, a_n$ where the characters are written in monotonically decreasing order, i.e. $a_1.\text{freq} \ge a_2.\text{freq} \ge \ldots \ge a_n$. Let us consider an optimal code B for C. Let us denote the codeword for the character $c \in C$ in the code B by $cw_B(c)$. W.l.o.g. we can assume that for any i such that $a_i.\text{freq} = a_{i+1}.\text{freq}$ it holds that $|cw(a_i)| \le |cw(a_{i+1})|$. This assumption can be made since for any $a_i.\text{freq} = a_{i+1}.\text{freq}$ for which $|cw(a_i)| > |cw(a_{i+1})|$ we can simply swap codewords for a_i and a_{i+1} and obtain a code with desired property and the same cost as is the cost of B. We prove that B has the desired property, i.e., its codeword lengths are monotonically increasing.

We proceed by contradiction. If lengths of the codewords are not monotonically increasing, then there exist an index i such that $|cw_B(a_i)| > |cw_B(a_{i+1})|$. Using our assumptions on C and B we get that $a_i.freq > a_{i+1}.freq$. Define new code B' for C such that for a_j such that $j \neq i$ and $j \neq i+1$ we keep $cw_{B'}(a_j) = cw_B(a_j)$ and we swap codewords for a_i and a_{j+1}, i.e. we set $cw_{B'}(a_i) = cw_B(a_{i+1})$ and $cw_{B'}(a_{i+1}) = cw_B(a_i)$. Now compare costs of the codes B and B'. It holds that

$$\begin{aligned} cost(B') &= cost(B) - (|cw_B(a_i)|(a_i.freq) + |cw_B(a_{i+1})|(a_{i+1}.freq)) \\ &+ (|cw_B(a_i)|(a_{i+1}.freq) + |cw_B(a_{i+1})|(a_i.freq)) \\ &= cost(B) + |cw_B(a_i)|(a_{i+1}.freq - a_i.freq) + |cw_B(a_{i+1})|(a_i.freq - a_{i+1}.freq) \end{aligned}$$

For better readability now denote $a_i.freq - a_{i+1}.freq = \phi$. Since $a_i.freq > a_{i+1}.freq$, we get $\phi > 0$ and we can write

$$cost(B') = cost(B) - \phi|cw_B(a_i)| + \phi|cw_B(a_{i+1})| = cost(B) - \phi(|cw_B(a_i)| - |cw_B(a_{i+1})|)$$

Since $|cw_B(a_i)| > |cw_B(a_{i+1})|$, we get $|cw_B(a_i)| - |cw_B(a_{i+1})| > 0$. Thus $\phi(|cw_B(a_i)| - |cw_B(a_{i+1})|) > 0$ which imply $cost(B') < cost(B)$. Therefore the code B is not optimal, a contradiction.

Therefore, we conclude that codeword lengths of B are monotonically increasing and the proof is complete.

Note: For those not familiar with mathematical parlance, w.l.o.g means without loss of generality.

16.3-6

> Suppose we have an optimal prefix code on a set $C = \{0, 1, \dots, n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n\lceil \lg n \rceil$ bits. (Hint: Use $2n-1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

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.

16.3-7

> Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

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.

16.3-8

> Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

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.

16.3-9

> Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. $($Hint: Compare the number of possible files with the number of possible encoded files.)

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 7 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.

16.4 Matroids and greedy methods

16.4-1

> Show that $(S, \mathcal{I}_k)$ is a matroid, where S is any finite set and $\mathcal{I}_k$ is the set of all subsets of S of size at most k, where $k \le |S|$.

The first condition that S is a finite set is a given. To prove the second condition we assume that $k \ge 0$, this gets us that $\mathcal{I}_k$ is nonempty. Also, to prove the hereditary property, suppose $A \in \mathcal{I}_k$ this means that $|A| \le k$. Then, if $B \subseteq A$, this means that $|B| \le |A| \le k$, 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 $x \in B \backslash A$, then,

$$|A \cup x| = |A| + 1 \le |B| \le k,$$

so, we can extend A to $A \cup \{x\} \in \mathcal{I}_k$.

16.4-2 ⋆

> Given an $m \times n$ matrix T over some field (such as the reals), show that $(S, \mathcal{I})$ is a matroid, where S is the set of columns of T and $A \in \mathcal{I}$ if and only if the columns in A are linearly independent.

Let $c_1, \dots, c_m$ be the columns of T. Suppose $C = \{c_{i1}, \dots, c_{ik}\}$ is dependent. Then there exist scalars $d_1, \dots, d_k$ not all zero such that $\sum_{j=1}^{k} d_j c_{ij} = 0$. 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.

16.4-3 ⋆

> Show that if $(S, \mathcal{I})$ is a matroid, then $(S, \mathcal{I}')$ is a matroid, where
>
> $\mathcal{I}' = \{A' : S - A'$ contains some maximal $A \in \mathcal{I}\}$.
>
> That is, the maximal independent sets of $(S, \mathcal{I}')$ are just the complements of the maximal independent sets of $(S, \mathcal{I})$.

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 $B \in \mathcal{I}'$.

Last, 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:

- The first case is that $|A - B| = 1$. Let $x \in A - B$ be the only element in $A - B$. Since $|A| > |B|$ and $|A - B| = 1$, it follows in this case $B \subset A$. We extend B by x and we have $B \cup \{x\} = A \in \mathcal{I}'$.
- The 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 $$. 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 $\mathcal I'$.

16.4-4 ⋆

> Let S be a finite set and let $S_1, S_2, \dots, S_k$ be a partition of S into nonempty disjoint subsets. Define the structure $(S, \mathcal{I})$ by the condition that $\mathcal{I} = \{A : \mid A \cap S_i \mid \le 1$ for $i = 1, 2, \dots, k\}$. Show that $(S, \mathcal{I})$ is a matroid. That is, the set of all sets A that contain at most one member of each subset in the partition determines the independent sets of a matroid.

Suppose $X \subset Y$ and $Y \in □$. Then $(X \cap S_i) \subset (Y \cap S_i)$ for all i, so

$$|X \cap S_i| \le |Y \cap S_i| \le 1$$

for all $1 \le i \le k$. Therefore □ is closed under inclusion.

Now Let $A, B \in □$ with $|A| > |B|$. Then there must exist some j such that $|A \cap S_j| = 1$ but $|B \cap S_j| = 0$. Let $a \in A \cap S_j$. Then $a \notin B$ and $|(B \cup \{a\}) \cap S_j| = 1$. Since

$$|(B \cup \{a\}) \cap S_i| = |B \cap S_i| \le 1$$

for all $i \ne j$, we must have $B \cup \{a\} \in □$. Therefore □ is a matroid.

16.4-5

> Show how to transform the weight function of a weighted matroid problem, where the desired optimal solution is a *minimum-weight* maximal independent subset, to make it a standard weighted-matroid problem. Argue carefully that your transformation is correct.

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.

16.5 A task-scheduling problem as a matroid

16.5-1

> Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty w_i replaced by $80 - w_i$.

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 $w_1 + w_2 = 30$.

16.5-2

> Show how to use property 2 of Lemma 16.12 to determine in time $O(|A|)$ whether or not a given set A of tasks is independent.

We provide a pseudocode which grasps main ideas of an algorithm.

```
IS-INDEPENDENT(A)
    n = A.length
    let Nts[0..n] be an array filled with 0s
    for each a in A
        if a.deadline >= n
            Nts[n] = Nts[n] + 1
        else
            Nts[d] = Nts[d] + 1
    for i = 1 to n
        Nts[i] = Nts[i] + Nts[i - 1]
    // at this moment, Nts[i] holds value of N_i(A)
    for i = 1 to n
        if Nts[i] > i
            return false
    return true
```

Problem 16-1 Coin changing

> Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.
>
> **a.** Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
>
> **b.** Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, \dots, c^k$ for some integers $c > 1$ and $k \ge 1$. Show that the greedy algorithm always yields an optimal solution.
>
> **c.** Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

> **d.** Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

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 $(x_0, x_1, \dots, x_k)$ 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 $\text{MAKE-CHANGE}(S, v)$ 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 Scheduling to minimize average completion time

> Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the ***completion time*** of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.
>
> **a.** Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.
>
> **b.** Suppose now that the tasks are not all available at once. That is, each task cannot start until its ***release time*** r_i. Suppose also that we allow ***preemption***, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run

> for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

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 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 Acyclic subgraphs

> **a.** The ***incidence matrix*** for an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = 1$ if edge e is incident on vertex v, and $M_{ve} = 0$ otherwise. Argue that a set of columns of M is linearly independent over the field of integers modulo 2 if and only if the corresponding set of edges is acyclic. Then, use the result of Exercise 16.4-2 to provide an alternate proof that $(E, \square)$ of part (a) is a matroid.
>
> **b.** Suppose that we associate a nonnegative weight $w(e)$ with each edge in an undirected graph $G = (V, E)$. Give an efficient algorithm to find an acyclic subset of E of maximum total weight.
>
> **c.** Let $G(V, E)$ be an arbitrary directed graph, and let $(E, \square)$ be defined so that $A \in \square$ if and only if A does not contain any directed cycles. Give an example of a directed graph G such that the associated system $(E, \square)$ is not a matroid. Specify which defining condition for a matroid fails to hold.
>
> **d.** The ***incidence matrix*** for a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix M such that $M_{ve} = -1$ if edge e leaves vertex v, $M_{ve} = 1$ if edge e enters vertex v, and $M_{ve} = 0$ otherwise. Argue that if a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle.
>
> **e.** Exercise 16.4-2 tells us that the set of linearly independent sets of columns of any matrix M forms a matroid. Explain carefully why the results of parts (d) and (e) are not contradictory. How can there fail to be a perfect correspondence between the notion of a set of edges being acyclic and the notion of the associated set of columns of the incidence matrix being linearly independent?

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 Scheduling variations

> 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 $a_j.\text{deadline} \le j$ 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 $a_j.\text{deadline} > j$ 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 $x.\text{low}$ and $x.\text{high}$ 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])
```

Problem 16-5 Off-line caching

> 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 $\langle r_1, r_2, \dots, r_n \rangle$ 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 $\langle r_1, r_2, \dots, r_n \rangle$ 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 $\langle r_1, \dots, r_n \rangle$ 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 $O(m + n)$.

```
SCHEDULING-VARIATIONS(A)
    let D be an array of size n
    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])
```

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.