15 Dynamic Programming

15.1 Rod cutting

15.1-1

> Show that equation (15.4) follows from equation (15.3) and the initial condition $T(0) = 1$.

- For $n = 0$, this holds since $2^0 = 1$.
- For $n > 0$, substituting into the recurrence, we have

  $$ \begin{aligned} T(n) &= 1 + \sum_{j = 0}^{n - 1} 2^j \\ &= 1 + (2^n - 1) \\ &= 2^n. \end{aligned} $$

15.1-2

> Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i / i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

The counterexample:

$$ \begin{array}{c|cccc} \text{length i} & 1 & 2 & 3 & 4 \\ \hline \text{price p_i} & 1 & 20 & 33 & 36 \\ p_i / i & 1 & 10 & 11 & 9 \end{array} $$

15.1-3

> Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

We can modify BOTTOM-UP-CUT-ROD algorithm from section 15.1 as follows:

```
MODIFIED-CUT-ROD(p, n, c)
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n
        q = p[j]
        for i = 1 to j - 1
            q = max(q, p[i] + r[j - i] - c)
        r[j] = q
    return r[n]
```

We need to account for cost c on every iteration of the loop in lines 5-6 but the last one, when $i = j$ (no cuts).

We make the loop run to $j - 1$ instead of j, make sure c is subtracted from the candidate revenue in line 6, then pick the greater of current best revenue q and $p[j]$ (no cuts) in line 7.

15.1-4

> Modify MEMOIZED-CUT-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)
```

15.1-5

> The 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., $v_0, v_1, \dots, v_n$.

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

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

15.2 Matrix-chain multiplication

15.2-1

> Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

$$((5 \times 10)(10 \times 3))(((3 \times 12)(12 \times 5))((5 \times 50)(50 \times 6))).$$

15.2-2

> Give a recursive algorithm $\text{MATRIX-CHAIN-MULTIPLY}(A, s, i, j)$ that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots ,A_n \rangle$, the s table computed by $\text{MATRIX-CHAIN-ORDER}$, and the indices i and j. (The initial call would be $\text{MATRIX-CHAIN-MULTIPLY}(A, s, 1, n)$.)

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
    if i == j
        return A[i]
    if i + 1 == j
        return A[i] * A[j]
    b = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
    c = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
    return b * c
```

15.2-3

> Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

Suppose $P(n) \ge c2^n$,

$$ \begin{aligned} P(n) &\ge \sum_{k = 1}^{n - 1} c2^k \cdot c2^{n - k} \\ &= \sum_{k = 1}^{n - 1} c^2 2^n \\ &= c^2 (n - 1) 2^n \\ &\ge c^2 2^n & (n > 1) \\ &\ge c 2^n. & (c \ge 1) \end{aligned} $$

15.2-4

> Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

The vertices of the subproblem graph are the ordered pair v_{ij}, where $i \le j$.

- If $i = j$, the vertex v_{ij} has no output edge.
- If $i < j$, for each k, s.t. $i \le k < j$, the subproblem graph contains edges (v_{ij}, v_{ik}) and $(v_{ij}, v_{k+1, j})$, and these edges indicate that to solve the subproblem of optimally parenthesizing the product $A_i \cdots A_j$, we need to solve subproblems of optimally parenthesizing the products $A_i \cdots A_k$ and $A_{k + 1} \cdots A_j$.

The number of vertices is

$$\sum_{i = 1}^n \sum_{j = i}^n = \frac{n(n + 1)}{2}.$$

The number of edges is

$$\sum_{i = 1}^n \sum_{j = i}^n (j - i) = \frac{(n - 1)n(n + 1)}{6}.$$

15.2-5

> Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of $\text{MATRIX-CHAIN-ORDER}$. Show that the total number of references for the entire table is
>
> $$\sum_{i = 1}^n \sum_{j = i}^n R(i, j) = \frac{n^3 - n}{3}.$$
>
> ($\textit{Hint:}$ You may find equation (A.3) useful.)

We count the number of times that we reference a different entry in m than the one we are computing, that is, 2 times the number of times that line 10 runs.

$$ \begin{aligned} \sum_{l = 2}^n \sum_{i = 1}^{n - l + 1} \sum_{k = i}^{i + l - 2} 2 &= \sum_{l = 2}^n \sum_{i = 1}^{n - l + 1} 2(l - 1) \\ &= \sum_{l = 2}^n 2(l - 1)(n - l + 1) \\ &= \sum_{l = 1}^{n - 1} 2l(n - l) \\ &= 2n \sum_{l = 1}^{n - 1} l - 2 \sum_{l = 1}^{n - 1} l^2 \\ &= n^2(n - 1) - 2 \cdot \frac{(n - 1)n(2n - 1)}{6} \\ &= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\ &= \frac{n^3 - n}{3}. \end{aligned} $$

15.2-6

> Show that a full parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses.

We proceed by induction on the number of matrices. A single matrix has no pairs of parentheses. Assume that a full parenthesization of an n-element expression has exactly $n – 1$ pairs of parentheses. Given a full parenthesization of an $(n + 1)$-element expression, there must exist some k such that we first multiply $B = A_1 \cdots A_k$ in some way, then multiply $C = A_{k + 1} \cdots A_{n + 1}$ in some way, then multiply B and C. By our induction hypothesis, we have $k – 1$ pairs of parentheses for the full parenthesization of B and $n + 1 – k – 1$ pairs of parentheses for the full parenthesization of C. Adding these together, plus the pair of outer parentheses for the entire expression, yields $k - 1 + n + 1 - k - 1 + 1 = (n + 1) - 1$ parentheses, as desired.

15.3 Elements of dynamic programming

15.3-1

> Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running $\text{RECURSIVE-MATRIX-CHAIN}$? Justify your answer.

Running $\text{RECURSIVE-MATRIX-CHAIN}$ is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

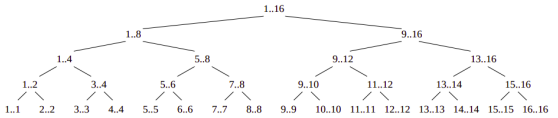
Consider the treatment of subproblems by each approach:

1. For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left and right half subproblem results is thus the product of the number of ways to parenthesize the left half and the number of ways to parenthesize the right half.
2. For each possible place to split the matrix chain, $\text{RECURSIVE-MATRIX-CHAIN}$ finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left and right half subproblem results is $O(1)$.

15.3-2

> Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

Draw a recursion tree.



The MERGE-SORT procedure performs at most a single call to any pair of indices of the array that is being sorted. In other words, the subproblems do not overlap and therefore memoization will not improve the running time.

15.3-3

> Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

Yes, this problem also exhibits optimal substructure. If we know that we need the subproduct $(A_l \cdot A_r)$, then we should still find the most expensive way to compute it — otherwise, we could do better by substituting in the most expensive way.

15.3-4

> As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i + 1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i - 1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Suppose that we are given matrices A_1, A_2, A_3, and A_4 with dimensions such that

$$p_0, p_1, p_2, p_3, p_4 = 1000, 100, 20, 10, 1000.$$

Then $p_0 p_k p_4$ is minimized when $k = 3$, so we need to solve the subproblem of multiplying $A_1 A_2 A_3$, and also A_4 which is solved automatically. By her algorithm, this is solved by splitting at $k = 2$. Thus, the full parenthesization is $(((A_1A_2)A_3)A_4)$.

This requires

$$1000 \cdot 100 \cdot 20 + 1000 \cdot 20 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 12200000$$

scalar multiplications.

On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1(A_2A_3))A_4)$. Then we would only require

$$100 \cdot 20 \cdot 10 + 1000 \cdot 100 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 11020000$$

scalar multiplications, which is fewer than Professor Capulet's method.

Therefore her greedy approach yields a suboptimal solution.

15.3-5

> Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

The optimal substructure property doesn't hold because the number of pieces of length i used on one side of the cut affects the number allowed on the other. That is, there is information about the particular solution on one side of the cut that changes what is allowed on the other.

To make this more concrete, suppose the rod was length 4, the values were $l_1 = 2, L_2 = L_3 = L_4 = 1$, and each piece has the same worth regardless of length. Then, if we make our first cut in the middle, we have that the optimal solution for the two rods left over is to cut it in the middle, which isn't allowed because it increases the total number of rods of length 1 to be too large.

15.3-6

> Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \ldots, n$, where you start with currency 1 and wish to wind up with currency n. You are given, for each pair of currencies i and j , an exchange rate r_{ij}, meaning that if you start with d units of currency i , you can trade for dr_{ij} units of currency j. A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \ldots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

First we assume that the commission is always zero. Let k denote a currency which appears in an optimal sequence s of trades to go from currency 1 to currency n. p_k denote the first part of this sequence which changes currencies from 1 to k and q_k denote the rest of the sequence. Then p_k and q_k are both optimal sequences for changing from 1 to k and k to n respectively. To see this, suppose that p_k wasn't optimal but that p_k' was. Then by changing currencies according to the sequence $p_k'q_k$ we would have a sequence of changes which is better than s, a contradiction since s was optimal. The same argument applies to q_k.

Now suppose that the commissions can take on arbitrary values. Suppose we have currencies 1 through 6, and $r_{12} = r_{23} = r_{34} = r_{45} = 2$, $r_{13} = r_{35} = 6$, and all other exchanges are such that $r_{ij} = 100$. Let $c_1 = 0$, $c_2 = 1$, and $c_k = 10$ for $k \ge 3$.

The optimal solution in this setup is to change 1 to 3, then 3 to 5, for a total cost of 13. An optimal solution for changing 1 to 3 involves changing 1 to 2 then 2 to 3, for a cost of 5, and an optimal solution for changing 3 to 5 is to change 3 to 4 then 4 to 5, for a total cost of 5. However, combining these optimal solutions to subproblems means making more exchanges overall, and the total cost of combining them is 18, which is not optimal.

15.4 Longest common subsequence

15.4-1

> Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

$\langle 1, 0, 0, 1, 1, 0 \rangle$ or $\langle 1, 0, 1, 0, 1, 0 \rangle$.

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)
```

15.4-3

> Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

```
MEMOIZED-LCS-LENGTH(X, Y, i, j)
    if c[i, j] > -1
        return c[i, j]
    if i == 0 or j == 0
        return c[i, j] = 0
    if x[i] == y[j]
        return c[i, j] = LCS-LENGTH(X, Y, i - 1, j - 1) + 1
    return c[i, j] = max(LCS-LENGTH(X, Y, i - 1, j), LCS-LENGTH(X, Y, i, j - 1))
```

15.4-4

> Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

Since we only use the previous row of the c table to compute the current row, we compute as normal, but when we go to compute row k, we free row $k - 2$ since we will never need it again to compute the length. To use even less space, observe that to compute $c[i, j]$, all we need are the entries $c[i - 1, j]$, $c[i - 1, j - 1]$, and $c[i, j - 1]$. Thus, we can free up entry-by-entry those from the previous row which we will never need again, reducing the space requirement to $\min(m, n)$. Computing the next entry from the three that it depends on takes $O(1)$ time and space.

15.4-5

> Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Given a list of numbers L, make a copy of L called L' and then sort L'.

```
PRINT-LCS(c, X, Y)
    n = c[X.length, Y.length]
    let s[1..n] be a new array
    i = X.length
    j = Y.length
    while i > 0 and j > 0
        if x[i] == y[j]
            s[n] = x[i]
            n = n - 1
            i = i - 1
            j = j - 1
        else if c[i - 1, j] ≥ c[i, j - 1]
            i = i - 1
        else j = j - 1
    for i = 1 to s.length
        print s[i]
```

```

```

```
MEMO-LCS-LENGTH-AUX(X, Y, c, b)
    m = |X|
    n = |Y|
    if c[m, n] != 0 or m == 0 or n == 0
        return
    if x[m] == y[n]
        b[m, n] = ↖
        c[m, n] = MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y[1..n - 1], c, b) + 1
    else if MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y, c, b) ≥ MEMO-LCS-LENGTH-AUX(X, Y[1..n - 1], c, b)
        b[m, n] = ↑
        c[m, n] = MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y, c, b)
    else
        b[m, n] = ←
        c[m, n] = MEMO-LCS-LENGTH-AUX(X, Y[1..n - 1], c, b)
```

```
MEMO-LCS-LENGTH(X, Y)
    let c[1..|X|, 1..|Y|] and b[1..|X|, 1..|Y|] be new tables
    MEMO-LCS-LENGTH-AUX(X, Y, c, b)
    return c and b
```

Then, just run the LCS algorithm on these two lists. The longest common subsequence must be monotone increasing because it is a subsequence of L' which is sorted. It is also the longest monotone increasing subsequence because being a subsequence of L' only adds the restriction that the subsequence must be monotone increasing. Since $|L| = |L'| = n$, and sorting L can be done in $o(n^2)$ time, the final running time will be $O(|L|lL'|) = O(n^2)$.

15.4-6 $\star$

> Give an $O(n\lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. ($\textit{Hint:}$ Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

The algorithm $\text{LONG-MONOTONIC}(A)$ returns the longest monotonically increasing subsequence of A, where A has length n.

The algorithm works as follows: a new array B will be created such that $B[i]$ contains the last value of a longest monotonically increasing subsequence of length i. A new array C will be such that $C[i]$ contains the monotonically increasing subsequence of length i with smallest last element seen so far.

To analyze the runtime, observe that the entries of B are in sorted order, so we can execute line 9 in $O(\lg n)$ time. Since every other line in the for-loop takes constant time, the total run-time is $O(n\lg n)$.

```
LONG-MONOTONIC(A)
    let B[1..n] be a new array where every value = ∞
    let C[1..n] be a new array
    L = 1
    for i = 1 to n
        if A[i] < B[1]
            B[1] = A[i]
            C[1].head.key = A[i]
        else
            let j be the largest index of B such that B[j] < A[i]
            B[j + 1] = A[i]
            C[j + 1] = C[j]
            INSERT(C[j + 1], A[i])
            if j + 1 > L
                L = L + 1
    print C[L]
```

15.5 Optimal binary search trees

15.5-1

> Write pseudocode for the procedure $\text{CONSTRUCT-OPTIMAL-BST}(root)$ which, given the table $root$, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure
>
> $$ \begin{aligned} & \text{k_2 is the root} \\ & \text{k_1 is the left child of k_2} \\ & \text{d_0 is the left child of k_1} \\ & \text{d_1 is the right child of k_1} \\ & \text{k_5 is the right child of k_2} \\ & \text{k_4 is the left child of k_5} \\ & \text{k_3 is the left child of k_4} \\ & \text{d_2 is the left child of k_3} \\ & \text{d_3 is the right child of k_3} \\ & \text{d_4 is the right child of k_4} \\ & \text{d_5 is the right child of k_5} \end{aligned} $$
>
> corresponding to the optimal binary search tree shown in Figure 15.9(b).

```
CONSTRUCT-OPTIMAL-BST(root, i, j, last)
    if i == j
        return
    if last == 0
        print root[i, j] + "is the root"
    else if j < last
        print root[i, j] + "is the left child of" + last
    else
        print root[i, j] + "is the right child of" + last
    CONSTRUCT-OPTIMAL-BST(root, i, root[i, j] - 1, root[i, j])
    CONSTRUCT-OPTIMAL-BST(root, root[i, j] + 1, j, root[i, j])
```

15.5-2

> Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities
>
> $$ \begin{array}{c|ccccccc} i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline p_i & & 0.04 & 0.06 & 0.08 & 0.02 & 0.10 & 0.12 & 0.14 \\ q_i & 0.06 & 0.06 & 0.06 & 0.06 & 0.05 & 0.05 & 0.05 & 0.05 \end{array} $$

Cost is 3.12.

$$ \begin{aligned} & \text{k_5 is the root} \\ & \text{k_2 is the left child of k_5} \\ & \text{k_1 is the left child of k_2} \\ & \text{d_0 is the left child of k_1} \\ & \text{d_1 is the right child of k_1} \\ & \text{k_3 is the right child of k_2} \\ & \text{d_2 is the left child of k_3} \\ & \text{k_4 is the right child of k_3} \\ & \text{d_3 is the left child of k_4} \\ & \text{d_4 is the right child of k_4} \\ & \text{k_7 is the right child of k_5} \\ & \text{k_6 is the left child of k_7} \\ & \text{d_5 is the left child of k_6} \\ & \text{d_6 is the right child of k_6} \\ & \text{d_7 is the right child of k_7} \end{aligned} $$

15.5-3

> Suppose that instead of maintaining the table $w[i, j]$, we computed the value of $w(i, j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

Computing $w(i, j)$ from the equation is $\Theta(j - i)$, since the loop below on lines 10-14 is also $\Theta(j - i)$, it wouldn't affect the asymptotic running time of OPTIMAL-BST which would stay $\Theta(n^3)$.

15.5-4 $\star$

> Knuth [212] has shown that there are always roots of optimal subtrees such that $root[i, j - 1] \le root[i, j] \le root[i + 1, j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Change the **for** loop of line 10 in OPTIMAL-BST to

```
    for r = r[i, j - 1] to r[i + 1, j]
```

Knuth's result implies that it is sufficient to only check these values because optimal root found in this range is in fact the optimal root of some binary search tree. The time spent within the **for** loop of line 6 is now $\Theta(n)$. This is because the bounds on r in the new **for** loop of line 10 are nonoverlapping.

To see this, suppose we have fixed l and i. On one iteration of the **for** loop of line 6, the upper bound on r is

$$r[i + 1, j] = r[i + 1, i + l - 1].$$

When we increment i by 1 we increase j by 1. However, the lower bound on r for the next iteration subtracts this, so the lower bound on the next iteration is

$$r[i + 1, j + 1 - 1] = r[i + 1, j].$$

Thus, the total time spent in the **for** loop of line 6 is $\Theta(n)$. Since we iterate the outer **for** loop of line 5 n times, the total runtime is $\Theta(n^2)$.

Problem 15-1 Longest simple path in a directed acyclic graph

> 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,

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

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.

Problem 15-10 Planning an investment strategy

> Your knowledge of algorithms helps you obtain an exciting job with the Acme Computer Company, along with a $$10,000$ signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use the Amalgamated Investment Company to manage your investments. Amalgamated Investments requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij} . In other words, if you invest d dollars in investment i in year j, then at the end of year j , you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investement. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$.
>
> **a.** The problem, as stated, allows you to invest your money inmultiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)
>
> **b.** Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.
>
> **c.** Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?
>
> **d.** Suppose that Amalgamated Investments imposed the additional restriction that, at any point, you can have no more than $$15,000$ in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

a. Without loss of generality, suppose that there exists an optimal solution S which involves investing d_1 dollars into investment k and d_2 dollars into investement m in year 1. Further, suppose in this optimal solution, you don't move your money for the first j years. If $r_{k1} + r_{k2} + \ldots + r_{kj} > r_{m1} +r_{m2} + \ldots + r_{mj}$ then we can perform the usual cut-and-paste maneuver and instead invest $d_1 + d_2$ dollars into investment k for j years. Keeping all other investments the same, this results in a strategy which is at least as profitable as S, but has reduced the number of different investments in a given span of years by 1. Continuing in this way, we can reduce the optimal strategy to consist of only a single investment each year.

b. If a particular investment strategy is the year-one-plan for a optimal investment strategy, then we must solve two kinds of optimal suproblem: either we maintain the strategy for an additional year, not incurring the moneymoving fee, or we move the money, which amounts to solving the problem where we ignore all information from year 1. Thus, the problem exhibits optimal substructure.

c. The algorithm works as follows: We build tables I and R of size 10 such that $I[i]$ tells which investment should be made (with all money) in year i, and $R[i]$ gives the total return on the investment strategy in years i through 10.

```
INVEST(d, n)
    let I[1..10] and R[1..10] be new tables
    for k = 10 downto 1
        q = 1
        for i = 1 to n
            if r[i, k] > r[q, k]   // i now holds the investment which looks
best for a given year
                q = i
        if R[k + 1] + dr_{I[k + 1]k} - f[1] > R[k + 1] +  dr[q, k] - f[2]  //
If revenue is greater when money is not moved
            R[k] = R[k + 1] + dr_{I[k + 1]k} - f[1]
            I[k] = I[k + 1]
        else
            R[k] = R[k + 1] + dr[q, k] - f[2]
            I[k] = q
    return I as an optimal stategy with return R[1]
```

d. The previous investment strategy was independent of the amount of money you started with. When there is a cap on the amount you can invest, the amount you have to invest in the next year becomes relevant. If we know the year-one-strategy of an optimal investment, and we know that we need to move money after the first year, we're left with the problem of investing a different initial amount of money, so we'd have to solve a subproblem for every possible initial amount of money. Since there is no bound on the returns, there's also no bound on the number of subproblems we need to solve.

Problem 15-11 Inventory planning

> The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i = 1}^n d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if, at the end of a month, the company is holding any unsold machines, it must pay inventory costs. The cost for holding j machines is given as a function $h(j)$ for $j = 1, 2, \ldots, D$, where $h(j) \ge 0$ for $1 \le j \le D$ and $h(j) \le h(j + 1)$ for $1 \le j \le D - 1$.
>
> Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polyomial in n and D.

Our subproblems will be indexed by and integer $i \in [n]$ and another integer $j \in [D]$. i will indicate how many months have passed, that is, we will restrict ourselves to only caring about $(d_i, \dots, d_n)$. j will indicate how many machines we have in stock initially. Then, the recurrence we will use will try producing all possible numbers of machines from 1 to $[D]$. Since the index space has size $O(nD)$ and we are only running through and taking the minimum cost from D many options when computing a particular subproblem, the total runtime will be $O(nD^2)$.

Problem 15-12 Signing free-agent baseball players

> Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $$X$ to spend on free agents. You are allowed to spend less than $$X$ altogether, but the owner will fire you if you spend any more than $$X$.
>
> You are considering N different positions, and for each position, P free-agent players who play that position are available. Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)
>
> To determine how valuable a player is going to be, you decide to use a sabermetric statistic known as "VORP", or "value over replacement player". A player with a higher VORP is more valuable than a player with a lower VORP. A player with a higher VORP is not necessarily more expensive to sign than a player with a lower VORP, because factors other than a player's value determine how much it costs to sign him.
>
> For each available free-agent player, you have three pieces of information:
>
> - the player's position,
> - the amount of money it will cost to sign the player, and
> - the player's VORP.
>
> Devise an algorithm that maximizes the total VORP of the players you sign while spending no more than $$X$ altogether. You may assume that each player signs for a multiple of $100,000$. Your algorithm should output the total VORP of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

We will make an $N + 1$ by $X + 1$ by $P + 1$ table. The runtime of the algorithm is $O(NXP)$.

```
BASEBALL(N, X, P)
    initialize a table B of size (N + 1) by (X + 1)
    initialize an array P of length N
    for i = 0 to N
        B[i, 0] = 0
    for j = 1 to X
        B[0, j] = 0
    for i = 1 to N
        for j = 1 to X
            if j < i.cost
                B[i, j] = B[i - 1, j]
            q = B[i - 1, j]
            p = 0
            for k = 1 to P
                if j >= i.cost
                    t = B[i - 1, j - i.cost] + i.value
                    if t > q
                        q = t
                        p = k
            B[i, j] = q
            P[i] = p
    print("The total VORP is", B[N, X], "and the players are:")
    i = N
    j = X
    C = 0
    for k = 1 to N // prints the players from the table
        if B[i, j] != B[i - 1, j]
            print(P[i])
            j = j - i.cost
            C = C + i.cost
        i = i - 1
    print("The total cost is", C)
```

Problem 15-2 Longest palindrome subsequence

> 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, racecar, and 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 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 $O(n^3)$.

Problem 15-3 Bitonic euclidean

> In the ***euclidean traveling-salesman problem***, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).
>
> J. L. Bentley has suggested that we simplify the problem by restricting our attention to ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
>
> Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. ($\textit{Hint:}$ Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

First sort all the points based on their x coordinate. To index our subproblem, we will give the rightmost point for both the path going to the left and the path going to the right. Then, we have that the desired result will be the subproblem indexed by v, where v is the rightmost point.

Suppose by symmetry that we are further along on the left-going path, that the leftmost path is going to the ith one and the right going path is going until the jth one. Then, if we have that $i > j + 1$, then we have that the cost must be the distance from the $i - 1$st point to the ith plus the solution to the subproblem obtained where we replace i with $i - 1$. There can be at most $O(n^2)$ of these subproblem, but solving them only requires considering a constant number of cases. The other possibility for a subproblem is that $j \le i \le j + 1$. In this case, we consider for every k from 1 to j the subproblem where we replace i with k plus the cost from kth point to the ith point and take the minimum over all of them. This case requires considering $O(n)$ things, but there are only $O(n)$ such cases. So, the final runtime is $O(n^2)$.

Problem 15-4 Printing neatly

> Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \le j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k = i}^j l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

First observe that the problem exhibits optimal substructure in the following way: Suppose we know that an optimal solution has k words on the first line. Then we must solve the subproblem of printing neatly words $l_{k + 1}, \ldots, l_n$. We build a table of optimal solutions solutions to solve the problem using dynamic programming. If $n - 1 + \sum_{k = 1}^n l_k < M$ then put all words on a single line for an optimal solution. In the following algorithm $\text{PRINT-NEATLY}(n)$, $C[k]$ contains the cost of printing neatly words l_k through l_n. We can determine the cost of an optimal solution upon termination by examining $C[1]$. The entry $P[k]$ contains the position of the last word which should appear on the first line of the optimal solution of words $l_k, \ldots, l_n$. Thus, to obtain the optimal way to place the words, we make $l_{P[1]}$ the last word on the first line, $l_{P[P[1]]}$ the last word on the second line, and so on.

```
PRINT-NEATLY(n)
    let P[1..n] and C[1..n] be new tables
    for k = n downto 1
        if sum_{i = k}^n l_i + n - k < M
            C[k] = 0
        q = ∞
        for j = 1 to n - k
            cost = sum_{m = 1}^j l_{k + m} + m - 1
            if cost < M and (M - cost)^3 + C[k + m + 1] < q
                q = (M - cost)^3 + C[k + m + 1]
                P[k] = k + j
        C[k] = q
```

Problem 15-5 Edit distance

In order to transform one source string of text $x[1..m]$ to a target string $y[1..n]$, we can perform various transformation operations. Our goal is, given x and y, to produce a series of transformations that change x to y. We use an array z—assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. We maintain current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$.

We may choose from among six transformation operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i + 1]$ and $z[j + 1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i + 1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

As an example, one way to transform the source string algorithm to the target string altruistic is to use the following sequence of operations, where the underlined characters are $x[i]$ and $z[j]$ after the operation:

$$ \begin{array}{lll} \text{Operation} & x & z \\ \hline \textit{initial strings} & \underline algorithm & \text{}\textunderscore} \\ \text{copy} & a\underline lgorithm & a\text{}\textunderscore} \\ \text{copy} & al\underline gorithm & al\text{}\textunderscore} \\ \text{replace by t} & alg\underline orithm & alt\text{}\textunderscore} \\ \text{delete} & algo\underline rithm & alt\text{}\textunderscore} \\ \text{copy} & algor\underline ithm & altr\text{}\textunderscore} \\ \text{insert u} & algor\underline ithm & altru\text{}\textunderscore} \\ \text{insert i} & algor\underline ithm & altrui\text{}\textunderscore} \\ \text{insert s} & algor\underline ithm & altruis\text{}\textunderscore} \\ \text{twiddle} & algorit\underline hm & altruisti\text{}\textunderscore} \\ \text{insert c} & algorit\underline hm & altruistic\text{}\textunderscore} \\ \text{kill} & algorithm\text{}\textunderscore} & altruistic\text{}\textunderscore} \end{array} $$

Note that there are several other sequences of transformation operations that transform algorithm to altruistic.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming algorithm to altruistic is

$$\text{($3 \cdot$ cost(copy)) + cost(replace) + cost(delete) + ($4 \cdot$ cost(insert)) + cost(twiddle) + cost(kill)}.$$

a. Given two sequences $x[1..m]$ and $y[1..n]$ and set of transformation-operation costs, the ***edit distance*** from x to y is the cost of the least expensive operatoin sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1..m]$ to $y[1..n]$ and prints an optimal opeartion sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [310, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \ne y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x = \text{GATCGGCAT}$ and $y = \text{CAATGTGAATC}$, one alignment is

$$ \begin{array}{ccccccccccc} \text G & & \text A & \text T & \text C & \text G & & \text G & \text C & \text A & \text T \\ \text C & \text A & \text A & \text T & & \text G & \text T & \text G & \text A & \text A & \text T & \text C \\

- & * & - & + & - & + & - & * & - & + & - & * & - & + & - & * & - & + & - & * & - \\
& - & - & + & - & + & - & * \end{array} $$

\end{array} $$

A $+$ under a position indicates a score of $+1$ for that position, a $-$ indicates a score of -1, and a $*$ indicates a score of -2, so that this alignment has a total score of $6 \cdot -2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

a. We will index our subproblems by two integers, $1 \le i \le m$ and $1 \le j \le n$. We will let i indicate the rightmost element of x we have not processed and j indicate the rightmost element of y we have not yet found matches for. For a solution, we call $\text{EDIT}(x, y, i, j)$.

b. We will set

$$\text{cost(delete)} = \text{cost(insert)} = 2,$$

$$\text{cost(copy)} = -1,$$

$$\text{cost(replace)} = 1,$$

and

$$\text{cost(twiddle)} = \text{cost(kill)} = \infty.$$

Then a minimum cost translation of the first string into the second corresponds to an alignment.

We view

- a copy or a replace as incrementing a pointer for both strings,
- a insert as putting a space at the current position of the pointer in the first string, and
- a delete operation means putting a space in the current position in the second string.

Since twiddles and kills have infinite costs, we will have neither of them in a minimal cost solution. The final value for the alignment will be the negative of the minimum cost sequence of edits.

```
EDIT(x, y, i, j)
    let m = x.length
    let n = y.length
    if i == m
        return (n − j)cost(insert)
    if j == n
        return min{(m − i)cost(delete), cost(kill)}
    initialize o1, ..., o5 to ∞
    if x[i] == y[j]
        o1 = cost(copy) + EDIT(x, y, i + 1, j + 1)
    o2 = cost(replace) + EDIT(x, y, i + 1, j + 1)
    o3 = cost(delete) + EDIT(x, y, i + 1, j)
    o4 = cost(insert) + EDIT(x, y, i, j + 1)
    if i < m − 1 and j < n − 1
        if x[i] == y[j + 1] and x[i + 1] == y[j]
            o5 = cost(twiddle) + EDIT(x, y, i + 2, j + 2)
    return min_{i ∈ [5]}{o_i}
```

Problem 15-6 Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

The problem exhibits optimal substructure in the following way: If the root r is included in an optimal solution, then we must solve the optimal subproblems rooted at the grandchildren of r. If r is not included, then we must solve the optimal subproblems on trees rooted at the children of r. The dynamic programming algorithm to solve this problem works as follows: We make a table C indexed by vertices which tells us the optimal conviviality ranking of a guest list obtained from the subtree with root at that vertex. We also make a table G such that $G[i]$ tells us the guest list we would use when vertex i is at the root. Let T be the tree of guests. To solve the problem, we need to examine the guest list stored at $G[T.root]$. First solve the problem at each leaf L. If the conviviality ranking at L is positive, $G[L] = \{L\}$ and $C[L] = L.conviv$. Otherwise $G[L] = \emptyset$ and $C[L] = 0$. Iteratively solve the subproblems located at parents of nodes at which the subproblem has been solved. In general for a node x,

☑ = \max(\sum_{y\text{ is a child of } x} C[y], x.conviv + \sum_{y\text{ is a grandchild of } x} C[y]).$$

The runtime is $O(n)$ since each node appears in at most two of the sums (because each node has at most 1 parent and 1 grandparent) and each node is solved once.

Problem 15-7 Viterbi algorithm

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds producted by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. ($\textit{Hint:}$ You may find concepts from Chapter 22 useful.)

Now, suppose that every edge $(u, v) \in E$ has an associated nonnegatve probability $p(u, v)$ of traversing the edge (u, v) from vertex u and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a "random walk" beginning at v_0 will follow the specified path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at v_0 and having label s. Analyze the running time of your algorithm.

a. Our substructure will consist of trying to find suffixes of s of length one less starting at all the edges leaving v_0 with label σ_0. if any of them have a solution, then, there is a solution. If none do, then there is none. See the algorithm VITERBI for details.

```
VITERBI(G, s, v[0])
    if s.length = 0
        return v[0]
    for edges(v[0], v[1]) in V for some v[1]
        if sigma(v[0], v[1]) = sigma[1]
            res = VITERBI(G, (sigma[2], ..., sigma[k]), v[1])
            if res != NO-SUCH-PATH
                return (v[0], res)
    return NO-SUCH-PATH
```

Since the subproblems are indexed by a suffix of s (of which there are only k) and a vertex in the graph, there are at most $O(k|V|)$ different possible arguments. Since each run may require testing a edge going to every other vertex, and each iteration of the **for** loop takes at most a constant amount of time other than the call to PROB-VITERBI, the final runtime is $O(k|V|^2)$.

b. For this modification, we will need to try all the possible edges leaving from v_0 instead of stopping as soon as we find one that works. The substructure is very similar. We'll make it so that instead of just returning the sequence, we'll have the algorithm also return the probability of that maximum probability sequence, calling the fields seq and prob respectively. See the algorithm PROB-VITERBI.

Since the runtime is indexed by the same things, we have that we will call it with at most $O(k|V|)$ different possible arguments. Since each run may require testing a edge going to every other vertex, and each iteration of the **for** loop takes at most a constant amount of time other than the call to PROB-VITERBI, the final runtime is $O(k|V|^2)$.

```
PROB-VITERBI(G, s, v[0])
    if s.length = 0
        return v[0]
    sols.seq = NO-SUCH-PATH
    sols.prob = 0
    for edges(v[0], v[1]) in V for some v[1]
        if sigma(v[0], v[1]) = sigma[1]
            res = PROB-VITERBI(G, (sigma[2], ..., sigma[k]), v[1])
            if p(v[0], v[1]) * res.prob ≥ sols.prob
                sols.prob = p(v[0], v[1]) * res.prob and sols.seq = v[0],
res.seq
    return sols
```

Problem 15-8 Image compression by seam carving

We are given a color picture consisting of an $m \times n$ array $A[1..m, 1..n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a "seam" from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

a. If $n > 1$ then for every choice of pixel at a given row, we have at least 2 choices of pixel in the next row to add to the seam (3 if we're not in column 1 or n). Thus the total number of possibilities is bounded below by 2^m.

b. We create a table $D[1..m, 1..n]$ such that $D[i, j]$ stores the disruption of an optimal seam ending at position $[i, j]$, which started in row 1. We also create a table $S[i, j]$ which stores the list of ordered pairs indicating which pixels were used to create the optimal seam ending at position (i, j). To find the solution to the problem, we look for the minimum k entry in row m of table D, and use the list of pixels stored at $S[m, k]$ to determine the optimal seam. To simplify the algorithm $\text{Seam}(A)$, let $\text{MIN}(a, b, c)$ be the function which returns -1 if a is the minimum, 0 if b is the minimum, and 1 if c is the minimum value from among a, b, and c. The time complexity of the algorithm is $O(mn)$.

```
SEAM(A)
    let D[1..m, 1..n] be a table with zeros
    let S[1..m, 1..n] be a table with empty lists
    for i = 1 to n
        S[1, i] = (1, i)
        D[1, i] = d_{1i}
    for i = 2 to m
        for j = 1 to n
            if j == 1 // left-edge case
                if D[i - 1, j] < D[i - 1, j + 1]
                    D[i, j] = D[i - 1, j] + d_{ij}
                    S[i, j] = S[i - 1, j].insert(i, j)
                else
                    D[i, j] = D[i - 1, j + 1] + d_{ij}
                    S[i, j] = S[i - 1, j + 1].insert(i, j)
            else if j == n // right-edge case
                if D[i - 1, j - 1] < D[i - 1, j]
                    D[i, j] = D[i - 1, j - 1] + d_{ij}
                    S[i, j] = S[i - 1, j - 1].insert(i, j)
                else
                    D[i, j] = D[i - 1, j] + d_{ij}
                    S[i, j] = S[i - 1, j].insert(i, j)
            x = MIN(D[i - 1, j - 1], D[i - 1, j], D[i - 1, j + 1])
            D[i, j] = D[i - 1, j + x]
            S[i, j] = S[i - 1, j + x].insert(i, j)
    q = 1
    for j = 1 to n
        if D[m, j] < D[m, q]
            q = j
    print(S[m, q])
```

Problem 15-9 Breaking a string

> A certain string-processing language allows a programmer to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose a programmer wants to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that the programmer wants to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If she programs the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If she programs the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, she could break first at 8 (costing
>
> 20), then break the left piece at 2 (costing 8), and finally the right piece at 10 (costing 12), for a total cost of 40.
>
> Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given a string S with n characters and an array $L[1..m]$ containing the break points, com- pute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

The subproblems will be indexed by contiguous subarrays of the arrays of cuts needed to be made. We try making each possible cut, and take the one with cheapest cost. Since there are m to try, and there are at most m^2 possible things to index the subproblems with, we have that the m dependence is that the solution is $O(m^3)$. Also, since each of the additions is of a number that is $O(n)$, each of the iterations of the for loop may take time $O(\lg n + \lg m)$, so, the final runtime is $O(m^3 \lg n)$. The given algorithm will return $(cost, seq)$ where $cost$ is the cost of the cheapest sequence, and seq is the sequence of cuts to make.

```
CUT-STRING(L, i, j, l, r)
    if l == r
        return (0, [])
    minCost = ∞
    for k = i to j
        if l + r + CUT-STRING(L, i, k, l, L[k]).cost + CUT-STRING(L, k, j,
L[k], j).cost < minCost
            minCost = r - l + CUT-STRING(L, i, k, l, L[k]).cost + CUT-
STRING(L, k + 1, j, L[k], j).cost
            minSeq = L[k] + CUT-STRING(L, i, k, l, L[k]) + CUT-STRING(L, i, k
+ 1, l, L[k])
    return (minCost, minSeq)
```

Sample call: ```cpp L = [3, 8, 10] S = 20 CUT-STRING(L, 0, len(L), 0, s)

```
<!-- hotfix: KaTeX -->
<!-- https://github.com/yzane/vscode-markdown-pdf/issues/21/ -->
<script type="text/javascript"
src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-
MML_HTMLorMML"></script>
<script type="text/x-mathjax-config">MathJax.Hub.Config({ tex2jax: {
inlineMath: [['$', '$']]}, messageStyle: "none" });</script>
```

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 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.

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 $\leq |S_{ij}| \in O(n)$. 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 $(a_0, a_1, \ldots, a_{n-1})$.
2. For each $0 \leq i < n$ 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 $S_0, S_1, \ldots, S_{n-1}$ 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 $W \geq w_n$.

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.

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 $i - 1, j - i.\text{weight}$ 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 $o_1, o_2, \ldots, o_k$. 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.

16.2-5

> Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ 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 $x.\text{freq} = b.\text{freq}$, then we must have $a.\text{freq} = b.\text{freq} = x.\text{freq} = y.\text{freq}$.

If we have that $x.\text{freq} = b.\text{freq}$, 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 $y.\text{freq} = x.\text{freq}$. Also, since $x.\text{freq} \leq a.\text{freq} \leq b.\text{freq} = x.\text{freq}$, we must have $a.\text{freq} = x.\text{freq}$.

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 $\Sigma = \{c_i \mid 1 \leq i \leq n\}$ such that for each i we have $c_i.\text{freq} = a_i$. 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:

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

By $\text{code}(c)$ 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 $z.\text{freq}$ is sum of frequencies of all leafs of the subtree of T rooted in z.

Consider tree T_n inductively defined by

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

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 $Q = (a_{i+2}, T_{i+1}, a_{i+3}, \ldots, a_n)$ with a_{i+2} being the minimal element for each $1 \leq i < n$. (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 $Q = (a_3, T_2, a_4, \ldots, a_n)$.
- for $2 \leq i < n$ 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 $i - \text{th}$ execution of the for-cycle we have the following content of the priority queue $Q = (a_{i+1}, T_i, a_{i+2}, \ldots, a_n)$. Thus line 5 of HUFFMAN extracts a_{i+1} and sets $z.\text{left} = a_{i+1}$ and line 6 of HUFFMAN extracts T_i and sets $z.\text{right} = T_i$. 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 $z.\text{freq} = T_{i+1}.\text{freq} = \sum_{i=1}^{i+1} a_i$. Thus using (CLAIM 1) one can see that $a_{i+2} < T_{i+1}.\text{freq} < a_{i+3}$. Therefore for the content of the priority queue Q just after the i-th execution of the for-cycle we have $Q = (a_{i+2}, T_{i+2}, a_{i+3}, \ldots, a_n)$.

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

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

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

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

Using this information, calculates the cost of T.

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

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. $a_1.\text{freq} \geq a_2.\text{freq} \geq \ldots \geq a_n$. 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 $\text{cw}_B(c)$. 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 $|\text{cw}(a_i)| \leq |\text{cw}(a_{i+1})|$. This assumption can be made since for any $a_i.\text{freq} = a_{i+1}.\text{freq}$ for which $|\text{cw}(a_i)| > |\text{cw}(a_{i+1})|$ 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

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

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 $C = \{0, 1, \ldots, n-1\}$ 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 $(S, \square_k)$ is a matroid, where S is any finite set and $\square_k$ is the set of all subsets of S of size at most k, where $k \leq |S|$.

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 $\square_k$ is nonempty. Also, to prove the hereditary property, suppose $A \in \square_k$ this means that $|A| \leq k$. Then, if $B \subseteq A$, this means that $|B| \leq |A| \leq k$, so $B \in \square_k$. Lastly, we prove the exchange property by letting $A, B \in \square_k$ be such that $|A| < |B|$. Then, we can pick any element $x \in B \backslash A$, then,

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

so, we can extend A to $A \cup \{x\} \in \square_k$.

16.4-2 *

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

Let $c_1, \ldots, c_m$ be the columns of T. Suppose $C = \{c_{i1}, \ldots, c_{ik}\}$ is dependent. Then there exist scalars $d_1, \ldots, d_k$ 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 $(S, \square)$ is a matroid, then $(S, \square')$ is a matroid, where
>
> $\square' = \{A' : S - A' \text{ contains some maximal } A \in \square\}$.
>
> That is, the maximal independent sets of $(S, \square')$ are just the complements of the maximal independent sets of $(S, \square)$.

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

Next we show the hereditary property, suppose that $B \subseteq A \in \square'$, then, there exists some $A' \in \square$ so that $S - A \subseteq A'$, however, $S - B \supseteq S - A \subseteq A$ so $B \in \square'$.

Last, we prove the exchange property. That is, if we have $B, A \in \square'$ 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 $B \cup \{x\} = A \in \square'$.
- The second case is if the first case does not hold. Let C be a maximal independent set of $\square$ 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 $S_1, S_2, \ldots, S_k$ be a partition of S into nonempty disjoint subsets. Define the structure $(S, \square)$ by the condition that $\square = \{A : |A \cap S_i| \leq 1 \text{ for } i = 1, 2, \ldots, k\}$. Show that $(S, \square)$ 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 $Y \in \square$. 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 $\square$ is closed under inclusion.

Now Let $A, B \in \square$ 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

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

for all $i \neq j$, we must have $B \cup \{a\} \in \square$. Therefore $\square$ 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 $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. 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 $(x_0, x_1, \ldots, x_k)$ 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 Scheduling to minimize average completion time

> Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ 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 $a_j.\text{deadline} \leq j$ 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 $a_j.\text{deadline} > j$ 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 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 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 $\langle r_1, r_2, \ldots, r_n \rangle$ 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 $O(m + n)$.

```
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.

17 Amortized Analysis

17.1 Aggregate analysis

17.1-1

If the set of stack operations included a MULTIPUSH operation, which pushses k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

No. The time complexity of such a series of operations depends on the number of pushes (pops vice versa) could be made. Since one MULTIPUSH needs $\Theta(k)$ time, performing n MULTIPUSH operations, each with k elements, would take $\Theta(kn)$ time, leading to amortized cost of $\Theta(k)$.

17.1-2

Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\Theta(nk)$ time.

The logarithmic bit flipping predicate does not hold, and indeed a sequence of events could consist of the incrementation of all 1s and decrementation of all 0s; yielding $\Theta(nk)$.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Let n be arbitrary, and have the cost of operation i be $c(i)$. Then we have,

$$\begin{aligned}\sum_{i=1}^{n} c(i) &= \sum_{i=1}^{\lceil \lg n \rceil} 2^i + \sum_{i \leq n \text{ is not a power of 2}} 1 \\ &\leq \sum_{i=1}^{\lceil \lg n \rceil} 2^i + n \\ &= 2^{1+\lceil \lg n \rceil} - 1 + n \\ &\leq 2n - 1 + n \\ &\leq 3n \in O(n).\end{aligned}$$

To find the average, we divide by n, and the amortized cost per operation is $O(1)$.

17.2 The accounting method

17.2-1

> Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

For every stack operation, we charge twice.

- First, we charge the actual cost of the stack operation.
- Second, we charge the cost of copying an element later on.

Since we have the size of the stack never exceed k, and there are always k operations between backups, we always overpay by at least enough. Therefore, the amortized cost of the operation is constant, and the cost of the n operation is $O(n)$.

17.2-2

> Redo Exercise 17.1-3 using an accounting method of analysis.

Let c_i = cost of ith operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

Charge 3 (amortized cost $\hat{c_i}$) for each operation.

- If i is not an exact power of 2, pay

  $$1\$, and store$$

  2 as credit.
- If i is an exact power of 2, pay $$i$, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is $$3$ per operation, $\sum\limits_{i = 1}^n \hat c_i = 3n$.

We know from Exercise 17.1-3 that $\sum_{i=1}^{n} \hat{c_i} < 3n$.

Then we have

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i \Rightarrow \text{credit} = \text{amortized cose} - \text{actual cost} \geq 0.$$

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

17.2-3

> Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (Hint: Keep a pointer to the high-order 1.)

(Removed)

17.3 The potential method

17.3-1

> Suppose we have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential fuction Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

Define the potential function $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ for all $i \geq 1$.

Then

$$\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0,$$

and

$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \geq 0.$$

The amortized cost is

$$\begin{aligned} \hat{c_i}' &= c_i + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \hat{c_i}. \end{aligned}$$

17.3-2

> Redo Exercise 17.1-3 using a potential method of analysis.

Define the potential function $\Phi(D_0) = 0$, and $\Phi(D_i) = 2i - 2^{1+\lfloor \lg i \rfloor}$ for $i > 0$. For operation 1,

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2^{1+\lfloor \lg i \rfloor} - 0 = 1.$$

For operation $i (i > 1)$, if i is not a power of 2, then

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2^{1+\lfloor \lg i \rfloor} - (2(i-1) - 2^{1+\lfloor \lg (i-1) \rfloor}) = 3.$$

If $i = 2^j$ for some $j \in \mathbb{N}$, then

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 2i - 2^{1+j} - (2(i-1) - 2^{1+j-1}) = i + 2i - 2i - 2i + 2 + i = 2.$$

Thus, the amortized cost is 3 per operation.

17.3-3

> Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

Make the potential function be equal to $\sum_{i=1}^{n} \lg i$ where n is the size of the min-heap. Then, there is still a cost of $O(\lg n)$ to INSERT, since only an amount of amortization that is $\lg n$ was spent to increase the size of the heap by 1.

However, the amortized cost of EXTRACT-MIN is 0, as all its actual cost is compensated.

17.3-4

> What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

Let Φ be the potential function that returns the number of elements in the stack. We know that for this potential function, we have amortized cost 2 for PUSH operation and amortized cost 0 for POP and MULTIPOP operations.

The total amortized cost is

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Using the potential function and the known amortized costs, we can rewrite the equation as

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c_i} + \Phi(D_0) - \Phi(D_n) \\ &= \sum_{i=1}^{n} \hat{c_i} + s_0 - s_n \\ &\leq 2n + s_0 - s_n, \end{aligned}$$

which gives us the total cost of $O(n + (s_0 - s_n))$. If $s_n \geq s_0$, then this equals to $O(n)$, that is, if the stack grows, then the work done is limited by the number of operations.

(Note that it does not matter here that the potential may go below the starting potential. The condition $\Phi(D_n) \geq \Phi(D_0)$ for all n is only required to have $\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$, but we do not need for that to hold in this application.)

17.3-5

> Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that b is constant.)

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c_i} - \Phi(D_n) + \Phi(D_0) \\ &= n - x + b \\ &\leq n - x + n \\ &= O(n). \end{aligned}$$

17.3-6

> Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

We'll use the accounting method for the analysis. Assign cost 3 to the ENQUEUE operation and 0 to the DEQUEUE operation. Recall the implementation of 10.1-6 where we enqueue by pushing on to the top of stack 1, and dequeue by popping from stack 2.

If stack 2 is empty, then we must pop every element from stack 1 and push it onto stack 2 before popping the top element from stack 2. For each item that we enqueue we accumulate 2 credits. Before we can dequeue an element, it must be moved to stack 2. Note: this might happen prior to the time at which we wish to dequeue it, but it will happen only once overall. One of the 2 credits will be used for this move. Once an item is on stack 2 its pop only costs 1 credit, which is exactly the remaining credit associated to the element. Since each operation's cost is $O(1)$, the amortized cost per operation is $O(1)$.

17.3-7

> Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:
>
> INSERT(S, x) inserts x into S.
>
> DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.
>
> Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

We'll store all our elements in an array, and if ever it is too large, we will copy all the elements out into an array of twice the length.

To delete the larger half, we first find the element m with order statistic $\lceil |S|/2 \rceil$ by the algorithm presented in section 9.3. Then, scan through the array and copy out the elements that are smaller or equal to m into an array of half the size.

Since the delete half operation takes time $O(|S|)$ and reduces the number of elements by $\lfloor |S|/2 \rfloor \in \Omega(|S|)$, we can make these operations take ammortized constant time by selecting our potential function to be linear in $|S|$.

Since the insert operation only increases $|S|$ by one, we have that there is only a constant amount of work going towards satisfying the potential, so the total ammortized cost of an insertion is still constant. To output all the elements just iterate through the array and output each.

17.4 Dynamic tables

17.4-1

> Suppose that we wish to implement a dynamic, open-address hash table. Why might we consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

By theorems 11.6-11.8, the expected cost of performing insertions and searches in an open address hash table approaches infinity as the load factor approaches one, for any load factor fixed away from 1, the expected time is bounded by a constant though. The expected value of the actual cost my not be $O(1)$ for every insertion because the actual cost may include copying out the current values from the current table into a larger table because it became too full. This would take time that is linear in the number of elements stored.

17.4-2

> Show that if $\alpha_{i-1} \geq 1/2$ and the ith operation on a dynamic table is TABLE-DELETE, then the amortized cost of the operation with respect to the potential function (17.6) is bounded above by a constant.

If $\alpha_{i-1} \geq 1/2$, TABLE-DELETE cannot ***contract***, so $c_i = 1$ and $\text{size}_i = \text{size}_{i-1}$.

- **Case 1:** if $\alpha_i \geq 1/2$,

  $$\begin{aligned} \hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2 \cdot (\text{num}_{i-1} - 1) - \text{size}_{i-1}) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= -1. \end{aligned}$$

- **Case 2:** if $\alpha_i < 1/2$,

  $$\begin{aligned} \hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i/2 - \text{num}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (\text{size}_{i-1}/2 - (\text{num}_{i-1} - 1)) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 2 + \frac{3}{2}\text{size}_{i-1} - 3 \cdot \text{num}_{i-1} \\ &= 2 + \frac{3}{2}\text{size}_{i-1} - 3\alpha_{i-1}\,\text{size}_{i-1} \\ &\leq 2 + \frac{3}{2}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} \\ &= 2. \end{aligned}$$

17.4-3

> Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function
>
> $$\Phi(T) = |2 \cdot T.\text{num} - T.\text{size}|,$$
>
> show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

If $1/3 < \alpha_i \leq 1/2$,

$$\begin{aligned} \hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_i - 2 \cdot (\text{num}_i + 1)) \\ &= 3. \end{aligned}$$

If the ith operation does trigger a contraction,

$$\begin{aligned} \frac{1}{3}\text{size}_{i-1} &= \text{num}_i + 1 \\ \text{size}_{i-1} &= 3(\text{num}_i + 1) \\ \text{size}_i &= \frac{2}{3}\text{size}_{i-1} = 2(\text{num}_i + 1). \end{aligned}$$

$$\begin{aligned} \hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= (\text{num}_i + 1) + [2 \cdot (\text{num}_i + 1) - 2 \cdot \text{num}_i] - [3 \cdot (\text{num}_i + 1) - 2 \cdot (\text{num}_i + 1)] \\ &= 2. \end{aligned}$$

Problem 17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or FFT. The first step of the FFT algorithm performs a ***bit-reversal permutation*** on an input array $A[0..n-1]$ whose length is $n = 2^k$ for some nonnegative integer k. This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k-bit sequence $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$, where $a = \sum_{i=0}^{k-1} a_i 2^i$. We define

$$\mathrm{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle;$$

thus,

$$\mathrm{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\mathrm{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12.

a. Given a function rev_k that runs in $\Theta(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure BIT-REVERSED-INCREMENT that, when given a bit-reversed-counter value a, produces $\mathrm{rev}_k(\mathrm{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to BIT-REVERSED-INCREMENT produce the sequence

$$0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots .$$

b. Assume that the words in your computer store k-bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe an implementation of the BIT-REVERSED-INCREMENT procedure that allows the bit-reversal permutation on an n-element array to be performed in a total of $O(n)$ time.

c. Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an $O(n)$-time bit-reversal permutation?

a. Initialize a second array of length n to all trues, then, going through the indices of the original array in any order, if the corresponding entry in the second array is true, then swap the element at the current index with the element at the bit-reversed position, and set the entry in the second array corresponding to the bit-reversed index equal to false. Since we are running $\mathrm{rev}_k < n$ times, the total runtime is $O(nk)$.

b. Doing a bit reversed increment is the same thing as adding a one to the leftmost position where all carries are going to the left instead of the right.

```
BIT-REVERSED-INCREMENT(a)
    let m be a 1 followed by k - 1 0s
    while m bitwise-AND is not zero
        a = a bitwise-XOR m
        shift m right by 1
    m bitwise-OR a
```

By a similar analysis to the binary counter (just look at the problem in a mirror), this BIT-REVERSED-INCREMENT will take constant ammortized time. So, to perform the bit-reversed permutation, have a normal binary counter and a bit reversed counter, then, swap the values of the two counters and increment. Do not swap however if those pairs of elements have already been swapped, which can be kept track of in a auxiliary array.

c. The BIT-REVERSED-INCREMENT procedure given in the previous part only uses single shifts to the right, not arbitrary shifts.

Problem 17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

c. Discuss how to implement DELETE.

a. We linearly go through the lists and binary search each one since we don't know the relationship between one list an another. In the worst case, every list is actually used. Since list i has length 2^i and it's sorted, we can search it in $O(i)$ time. Since i varies from 0 to $O(\lg n)$, the runtime of SEARCH is $O(\lg^2 n)$.

b. To insert, we put the new element into A_0 and update the lists accordingly. In the worst case, we must combine lists $A_0, A_1, \ldots, A_{m-1}$ into list Am. Since merging two sorted lists can be done linearly in the total length of the lists, the time this takes is $O(2^m)$. In the worst case, this takes time $O(n)$ since m could equal k.

We'll use the accounting method to analyse the amoritized cost. Assign a cost of $\lg n$ to each insertion. Thus, each item carries $\lg n$ credit to pay for its later merges as additional items are inserted. Since an individual item can only be merged into a larger list and there are only $\lg n$ lists, the credit pays for all future costs the item might incur. Thus, the amortized cost is $O(\lg n)$.

c. Find the smallest m such that $n_m \neq 0$ in the binary representation of n. If the item to be deleted is not in list A_m, remove it from its list and swap in an item from Am, arbitrarily. This can be done in $O(\lg n)$ time since we may need to search list A_k to find the element to be deleted. Now simply break list A_m into lists $A_0, A_1, \ldots, A_{m-1}$ by index. Since the lists are already sorted, the runtime comes entirely from making the splits, which takes $O(m)$ time. In the worst case, this is $O(\lg n)$.

Problem 17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is ***α-balanced*** if $x.left.size \leq \alpha \cdot x.size$ and $x.right.size \leq \alpha \cdot x.size$. The tree as a whole is ***α-balanced*** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$-balanced. Your algorithm should run in time $\Theta(x.size)$, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that we implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then we "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\Delta(x) = |x.left.size - x.right.size|,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and that a $1/2$-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

a. Since we have $O(x.size)$ auxiliary space, we will take the tree rooted at x and write down an inorder traversal of the tree into the extra space. This will only take linear time to do because it will visit each node thrice, once when passing to its left child, once when the nodes value is output and passing to the right child, and once when passing to the parent. Then, once the inorder traversal is written down, we can convert it back to a binary tree by selecting the median of the list to be the root, and recursing on the two halves of the list that remain on both sides. Since we can index into the middle element of a list in constant time, we will have the recurrence

$$T(n) = 2T(n/2) + 1,$$

which has solution that is linear. Since both trees come from the same underlying inorder traversal, the result is a BST since the original was. Also, since the root at each point was selected so that half the elements are larger and half the elements are smaller, it is a $1/2$-balanced tree.

b. We will show by induction that any tree with $\leq \alpha^{-d} + d$ elements has a depth of at most d. This is clearly true for $d = 0$ because any tree with a single node has depth 0, and since $\alpha^0 = 1$, we have that our restriction on the number of elements requires there to only be one. Now, suppose that in some inductive step we had a contradiction, that is, some tree of depth d that is α balanced but has more than $\alpha - d$ elements.

We know that both of the subtrees are alpha balanced, and by being alpha balanced at the root, we have

$$root.left.size \leq \alpha \cdot root.size,$$

which implies

$$root.right.size > root.size - \alpha \cdot root.size - 1.$$

So,

$$\begin{aligned} root.right.size &> (1-\alpha) root.size - 1 \\ &> (1-\alpha)\alpha - d + d - 1 \\ &= (\alpha - 1 - 1)\alpha - d + 1 + d - 1 \\ &\geq \alpha - d + 1 + d - 1, \end{aligned}$$

which is a contradiction to the fact that it held for all smaller values of d because any child of a tree of depth d has depth $d-1$.

c. The potential function is a sum of $\Delta(x)$ each of which is the absolute value of a quantity, so, since it is a sum of nonnegative values, it is nonnegative regardless of the input BST.

If we suppose that our tree is $1/2$-balanced, then, for every node x, we'll have that $\Delta(x) \leq 1$, so, the sum we compute to find the potential will be over no nonzero terms.

d.

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ O(1) &= m + \Phi(D_i) - \Phi(D_{i-1}) \\ \Phi(D_{i-1}) &= m + \Phi(D_i) \\ \Phi(D_{i-1}) &\geq m. \end{aligned}$$

$$\begin{aligned} \Delta(x) &= x.left.size - x.right.size \\ &\geq \alpha \cdot m - ((1-\alpha)m - 1) \\ &= (2\alpha - 1)m + 1. \end{aligned}$$

$$\begin{aligned} m &\leq c((2\alpha - 1)m + 1) \\ c &\geq \frac{m}{(2\alpha - 1)m + 1} \\ &\geq \frac{1}{2\alpha}. \end{aligned}$$

e. Suppose that our tree is α balanced. Then, we know that performing a search takes time $O(\lg(n))$. So, we perform that search and insert the element that we need to insert or delete the element we found. Then, we may have made the tree become unbalanced. However, we know that since we only changed one position, we have only changed the Δ value for all of the parents of the node that we either inserted or deleted. Therefore, we can rebuild the balanced properties starting at the lowest such unbalanced node and working up.

Since each one only takes ammortized constant time, and there are $O(\lg(n))$ many trees made unbalanced, tot total time to rebalanced every subtree is $O(\lg(n))$ ammortized time.

Problem 17-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n+1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case. Note that we count each color change as a structural modification.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (Hint: Look at Figures 13.5, 13.6, and 13.7.)

We shall first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

d. When we insert a node into a red-black tree using RB-INSERT, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red,} \\ 1 & \text{if } x \text{ is black and has no red children,} \\ 0 & \text{if } x \text{ is black and has one red children,} \\ 2 & \text{if } x \text{ is black and has two red children.} \end{cases}$$

Now we redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x),$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

a. If we insert a node into a complete binary search tree whose lowest level is all red, then there will be $\Omega(\lg n)$ instances of case 1 required to switch the colors all the way up the tree. If we delete a node from an all-black,

complete binary tree then this also requires $\Omega(\lg n)$ time because there will be instances of case 2 at each iteration of the **while** loop.

b. For RB-INSERT, cases 2 and 3 are terminating. For RB-DELETE, cases 1 and 3 are terminating.

c. After applying case 1, z's parent and uncle have been changed to black and z's grandparent is changed to red. Thus, there is a ned loss of one red node, so $\Phi(T') = \Phi(T) - 1$.

d. For case 1, there is a single decrease in the number of red nodes, and thus a decrease in the potential function. However, a single call to RB-INSERTFIXUP could result in $\Omega(\lg n)$ instances of case 1. For cases 2 and 3, the colors stay the same and each performs a rotation.

e. Since each instance of case 1 requires a specific node to be red, it can't decrease the number of red nodes by more than $\Phi(T)$. Therefore the potential function is always non-negative. Any insert can increase the number of red nodes by at most 1, and one unit of potential can pay for any structural modifications of any of the 3 cases. Note that in the worst case, the call to RB-INSERT has to perform k case-1 operations, where k is equal to $\Phi(T_i) - \Phi(T_{i-1})$. Thus, the total amortized cost is bounded above by $2(\Phi(T_n) - \Phi(T_0)) \leq n$, so the amortized cost of each insert is $O(1)$.

f. In case 1 of RB-INSERT, we reduce the number of black nodes with two red children by 1 and we at most increase the number of black nodes with no red children by 1, leaving a net loss of at most 1 to the potential function. In our new potential function, $\Phi(T_n) - \Phi(T_0) \leq n$. Since one unit of potential pays for each operation and the terminating cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-INSERT-FIXUP $O(1)$.

g. In case 2 of RB-DELETE, we reduce the number of black nodes with two red children by 1, thereby reducing the potential function by 2. Since the change in potential is at least negative 1, it pays for the structural modifications. Since the other cases cause constant structural changes, the total amortized cost is $O(n)$ making the amortized cost of each RB-DELETE-FIXUP $O(1)$.

h. As described above, whether we insert or delete in any of the cases, the potential function always pays for the changes made if they're nonterminating. If they're terminating then they already take constant time, so the amortized cost of any operation in a sequence of m inserts and deletes is $O(1)$, making the toal amortized cost $O(m)$.

Problem 17-5 Competitive analysis of self-organizing lists with move-to-front

A ***self-organizing list*** is a linked list of n elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.

A self-organizing list has two important properties:

1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the kth element from the start of the list, then the cost to find the element is k.
2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of n elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements-switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than 4 times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a ***competitive analysis***.

For a heuristic H and a given initial ordering of the list, denote the access cost of sequence σ by $C_H(\sigma)$ Let m be the number of accesses in σ.

a. Argue that if heuristic H does not know the access sequence in advance, then the worst-case cost for H on an access sequence σ is $C_H(\sigma) = \Omega(mn)$.

With the ***move-to-front*** heuristic, immediately after searching for an element x, we move x to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element x in list L, that is, the position of x in list L. For example, if x is the fourth element in L, then $\text{rank}_L(x) = 4$. Let c_i denote the cost of access σ_i using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

b. Show that if σ_i accesses element x in list L using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Now we compare move-to-front with any other heuristic H that processes an access sequence according to the two properties above. Heuristic H may transpose elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let L_i be the list after access σ_i using move-to-front, and let L_i^* be the list after access σ_i using heuristic H. We denote the cost of access σ_i by c_i for move-to-front and by c_i^* for heuristic H. Suppose that heuristic H performs t_i^* transpositions during access σ_i.

c. In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

We define an ***inversion*** in list L_i as a pair of elements y and z such that y precedes z in L_i and z precedes y in list L_i^*. Suppose that list L_i has q_i inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_i \rangle$. Then, we define a potential function Φ that maps L_i to a real number by $\Phi(L_i) = 2q_i$. For example, if L_i has the elements $\langle e, c, a, d, b \rangle$ and L_i^* has the elements $\langle c, a, b, d, e \rangle$, then L_i has 5 inversions $((e, c), (e, a), (e, d), (e, b), (d, b))$, and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all i and that, if move-to-front and heuristic H start with the same list L_0, then $\Phi(L_0) = 0$.

d. Argue that a transposition either increases the potential by 2 or decreases the potential by 2.

Suppose that access σ_i finds the element x. To understand how the potential changes due to σ_i, let us partition the elements other than x into four sets, depending on where they are in the lists just before the ith access:

- Set A consists of elements that precede x in both L_{i-1} and L_{i-1}^*.
- Set B consists of elements that precede x in L_{i-1} and follow x in L_{i-1}^*.
- Set C consists of elements that follow x in L_{i-1} and precede x in L_{i-1}^*.
- Set D consists of elements that follow x in both L_{i-1} and L_{i-1}^*.

e. Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

f. Show that access σ_i causes a change in potential of

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*),$$

where, as before, heuristic H performs t_i^* transpositions during access σ_i.

Define the amortized cost $\hat{c_i}$ of access σ_i by $\hat{c_i} = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

g. Show that the amortized cost $\hat{c_i}$ of access σ_i is bounded from above by $4c_i^*$.

h. Conclude that the cost $C_{MTF}(\sigma)$ of access sequence σ with move-to-front is at most 4 times the cost $C_H(\sigma)$ of σ withany other heuristic H, assuming that both heuristics start with the same list.

a. Since the heuristic is picked in advance, given any sequence of requests given so far, we can simulate what ordering the heuristic will call for, then, we will pick our next request to be whatever element will of been in the last position of the list. Continuing until all the requests have been made, we have that the cost of this sequence of accesses is $= mn$.

b. The cost of finding an element is $= \text{rank}_L(x)$ and since it needs to be swapped with all the elements before it, of which there are $\text{rank}_L(x) - 1$, the total cost is $2 \cdot \text{rank}_L(x) - 1$.

c. Regardless of the heuristic used, we first need to locate the element, which is left where ever it was after the previous step, so, needs $\text{rank}_{L_{i-1}}(x)$. After that, by definition, there are t_i transpositions made, so, $c_i^* = \text{rank}_{L_{i-1}}(x) + t_i^*$.

d. If we perform a transposition of elements y and z, where y is towards the left. Then there are two cases. The first is that the final ordering of the list in L_i^* is with y in front of z, in which case we have just increased the number of inversions by 1, so the potential increases by 2. The second is that in $L_i^* z$ occurs before y, in which case, we have just reduced the number of inversions by one, reducing the potential by 2.

In both cases, whether or not there is an inversion between y or z and any other element has not changed, since the transposition only changed the relative ordering of those two elements.

e. By definition, A and B are the only two of the four categories to place elements that precede x in L_{i-1}, since there are $|A| + |B|$ elements preceding it, it's rank in L_{i-1} is $|A| + |B| + 1$. Similarly, the two categories in which an element can be if it precedes x in L_{i-1}^* are A and C, so, in L_{i-1}^*, x has $\text{rank} |A| + |C| + 1$.

f. We have from part d that the potential increases by 2 if we transpose two elements that are being swapped so that their relative order in the final ordering is being screwed up, and decreases by two if they are begin placed into their correct order in L_i^*.

In particular, they increase it by at most 2, since we are keeping track of the number of inversions that may not be the direct effect of the transpositions that heuristic H made, we see which ones the Move to front heuristic may of added. In particular, since the move to front heuristic only changed the relative order of x with respect to the other elements, moving it in front of the elements that preceded it in L_{i-1}, we only care about sets A and B. For an element in A, moving it to be behind A created an inversion, since that element preceded x in L_i^*. However, if the element were in B, we are removing an inversion by placing x in front of it.

g.

$$\begin{aligned}
\hat{c_i} &\leq 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t_i^*) \\
&= 4|A| + 1 + 2t_i^* \\
&\leq 4(|A| + |C| + 1 + t_i^*) \\
&= 4c_i^*.
\end{aligned}$$

h. We showed that the amortized cost of each operation under the move to front heuristic was at most four times the cost of the operation using any other heuristic. Since the amortized cost added up over all these operation is at most the total (real) cost, so we have that the total cost with movetofront is at most four times the total cost with an arbitrary other heuristic.