15.1-5: The Fibonnacci numbers are defined by recurrence (3.22).

Give an $O(n)$-time dynamic-programming algorithm to compute the n^{th} Fibonnacci number. Draw the subproblem graph. How many vertices & edges are in the graph?

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

FIBONACCI directly implements the recurrence relation of the Fibonacci sequence. Each number in the sequence is the sum of the 2 previous numbers in the sequence. The running time is clearly $O(n)$.

The subproblem graph consists of $n+1$ vertices, $V_0, V_1, \ldots, V_n$. For $i = 2, 3, \ldots, n$, vertex V_i has two leaving edges: to vertex V_{i-1} & to vertex V_{i-2}. No edges leaves vertices V_0 & V_1. Thus, the subproblem graph has $2n-2$ edges.

15: Dynamic Programming

15.2: Matrix-multiplication

Counting the number of parenthesizations.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n \geq 2 \end{cases} \quad \Omega(2^n).$$

Recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j. \end{cases}$$

Exercise:

15.2-1: $((5\times10)(10\times3))((3\times12)(12\times5))((5\times50)(50\times6))$.

15.2-2:

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

15.2-3:

$$P(n) = \begin{cases} 1 & \text{if } n=1. \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n \geq 2. \end{cases}$$

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

$$P(n) \geq \sum_{k=1}^{n-1} c2^k \cdot c2^{n-k}$$

$$= \sum_{k=1}^{n-1} c^2 2^n$$

$$= c^2 (n-1) 2^n$$

$$\geq c^2 n \qquad (n>1)$$

$$\boxed{P(n) \geq c2^n} \qquad (c \geq 1)$$

15.2-4: The vertices of the subproblem graph are the ordered pairs V_{ij}, where $i \leq j$. If $i=j$, then there are no edges out of V_{ij}. If $i<j$, then for every k such that $i \leq k < j$, the subproblem graph contains edges (V_{ij}, V_{jk}) & $(V_{ij}, V_{k+1,j})$. These edges indicates 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

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

& The number of edges is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i) = \sum_{i=1}^{n} \sum_{t=0}^{n-i} t \qquad [\text{Substituting } t = j-i]$$

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

Substituting $r = n-i$ and reversing the order of summation, we obtain

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

$$= \frac{1}{2} \left(\frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} \right)$$

$$= \frac{(n-1)n(n+1)}{6}$$

Thus, the subproblem graph has $\Theta(n^2)$ vertices & $\Theta(n^3)$ edges.

15.2-5: Each time the l-loop executes, the i-loop executes $n-l+1$ times. Each time the i-loop executes, the k-loop executes $j-i = l-1$ times, each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n} (n-l+1)(l-1)2$.

Thus,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} R(i,j) = \sum_{l=2}^{n} (n-l+1)(l-1)2$$

$$= 2 \sum_{l=1}^{n-1} (n-l)l$$

$$= 2 \sum_{l=1}^{n-1} nl - 2 \sum_{l=1}^{n-1} l^2$$

$$= 2 \cdot \frac{n(n-1)n}{2} - 2 \frac{(n-1)n(2n-1)}{6}$$

$$= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3}$$

$$= \frac{n^3 - n}{3}$$ Hence proved.

15.2-6: Show that a full parenthesization of an n-element exp. has exactly $n-1$ pairs of parentheses.

We proceed by induction on the number of matrices.
A single matrix has no pairs of parantheses.
Assume that a full paranthesization of an n-element expression has exactly $n-1$ pairs of parentheses.
Given a full paranthesization of an $(n+1)$-element exp. 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 & C. By our induction hypothesis we have $k-1$ pairs of parenthesis for the full parenthesization of B & $n+1-k-1$ pairs of parantheses for the full parenthesization of C. Adding these together, plus the pair of outer parentheses for the entire exp. yields $k-1+n+1-k-1+1 = (n+1)-1$ parentheses.

15.3: Elements of dynamic programming

Exercise:

15.3-1 Recursive-matrix-chain

$$T(n) \le \begin{cases} c & \text{if } n=1, \\ c + \sum_{k=1}^{n+1} (T(k)+T(n-k)+c) & \text{if } n \ge 2. \end{cases}$$

If c instead of 1, then

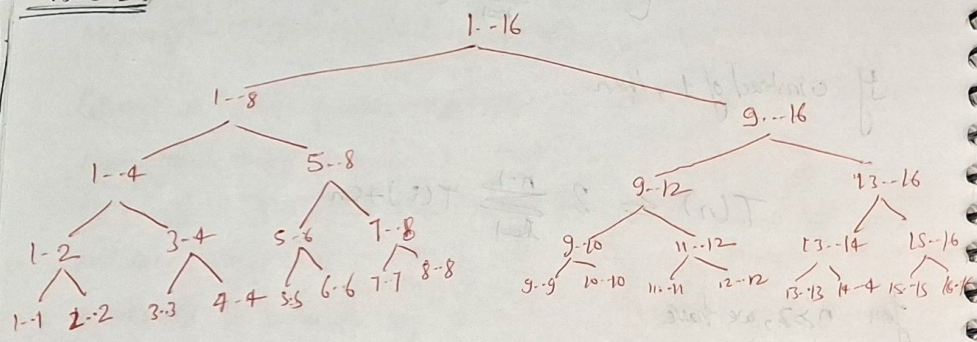
$$T(n) \le 2 \sum_{k=1}^{n-1} T(i) + cn$$

for $n \ge 2$, we have

$$\begin{aligned} T(n) &\le 2 \sum_{i=1}^{n-1} T(i) + cn \\ &\le 2 \sum_{i=1}^{n-1} c i 3^{i-1} + cn \\ &\le c \cdot \left(2 \sum_{i=1}^{n+1} i 3^{i-1} + n \right) \\ &= c \cdot \left(2 \cdot \left(\frac{n 3^{n+1}}{3-1} + \frac{1-3^n}{(3-1)^2} \right) + n \right) \\ &= c n 3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n \right) \\ &= c n 3^{n-1} + \frac{c}{2} (2n+1-3^n) \\ &\le c n 3^{n-1} \qquad (\forall\, c>0,\ n \ge 1) \end{aligned}$$

Running RECURSIVE-MATRIX-CHAIN takes $O(n3^{n-1})$ time & enumerating all parenthesizations takes $(4^n/n^{3/2})$ times & so RECURSIVE-MATRIX-CHAIN is more efficient than enumeration.

15.3-2:



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 & therefore memorization will not improve the running time.

15.3-3: Yes, this problem also exhibits optimal substructure. If we know that we need the subproblem $(A_i..A_j)$, 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: Suppose that we are given matrices A_1, A_2, A_3 & 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 $A_1A_2A_3$ & also A_4 which is solved automatically. By the algo., 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 $(((A_1(A_2A_3))A_4)$. The we would require only

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

Scalar multiplications, which is fewer than professor Copulet's method.

Therefore, the greedy approach yields a suboptimal soln.

16: Greedy Algorithms

Greedy vs. dynamic programming

The knapsack problem is a good example of the diff.

* 0-1 knapsack problem:

→ n items
→ Item i is worth v_i, weight w_i pounds.
→ Find a most valuable subset of items with total weight ≤ W.
→ Have to either take an item or not take it — can't take part of it.

* Fractional knapsack problem:

→ Like the 0-1 knapsack problem, but can take fractional of an item.
→ Both have optimal substructure.
→ But the fractional knapsack problem has the greedy-choice property & the 0-1 knapsack problem does not.
→ To solve the fractional problem, rank items by value/weight: $\frac{v_i}{w_i}$
→ Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.

```
FRACTIONAL-KNAPSACK (v, w, W)
  load = 0
  i = 1
  while load < W & i ≤ n
      do if w_i ≤ W - load
           then take all of item i
           else take (W - load)/w_i of item i
         add what was taken to load
         i = i+1
```

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$

Greedy Sol^n:
* Take item 1 & 2 & $\frac{2}{3}$ of 3
* Value = 160, Weight = 30
* Have 20 pounds of the capacity left over.

$\frac{20}{30} \times 120 = 80$

$160 + 80 = 240$ Ans.

Optimal Sol^n:
* Take item 2 & 3
* value = 220, weight = 50
* No leftover capacity

Exercise 16.2-1: Let I be the following instance of the knapsack problem: Let n be the number of items, let v_i be the value of ith item, let w_i be the weight of ith item & W be the capacity. Assume the items have ordered in increasing order by v_i/w_i & $W \geq w_n$.

Let $S = (s_1, s_2, \ldots s_n)$ be a solution. The greedy algorithm works by assigning $s_n = \min(w_n, W)$ & then continuing by solving 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 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 $\max(0, W - w_n)$ & decreasing s_n by the same amount, we get better sol^n. Since this a contradiction the assumption must be false. Hence the problem has the greedy-choice property.

16.2-2:

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ c[i-1,w] & \text{if } w_i > w, \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ \& } w \geq w_i \end{cases}$$

```
DYNAMIC-0-1-KNAPSACK (v, w, n, W)
    let c[0..n, 0..W] be an new array
    for w = 0 to W
        c[0, w] = 0
    for i = 1 to n
        c[i, 0] = 0
        for w = 1 to W
            if w[i] ≤ W
                if v[i] + c[i-1, w-w[i]] > c[i-1, w]
                    c[i, w] = v[i] + c[i-1, w-w[i]]
                else c[i, w] = c[i-1, w]
            else c[i, w] = c[i-1, w]
```

The above algorithm takes $\Theta(nW)$ time total:
→ $\Theta(nW)$ to fill in the c table: $(n+1) \cdot (W+1)$ entries, each requiring $\Theta(1)$ time to compute.
→ $O(n)$ time to trace the solution (since it starts in row n of the table & moves up on row at each step.

16.2-3: Suppose in an Optimal sol^n. we take an item with v_1, w_1 & drop an item with v_2, w_2 & $w_1 > w_2$, $v_1 < v_2$,

We can substitute 1 with 2 & get a better solution. Therefore we should always choose the items with the greatest values.

16.2-6: Show how to solve the fractional knapsack problem in $O(n)$ time.

Use a linear-time median algorithm to calculate the median m of the v_i/w_i ratios. Next, partition the items into three sets: ① $G = \{ i : v_i/w_i > m \}$ ② $E = \{ i : v_i/w_i = m \}$ ③ $L = \{ i : v_i/w_i < m \}$ } this step takes linear time

Compute $W_G = \sum_{i \in G} w_i$ and $W_E = \sum_{i \in E} w_i$, the total weight of the items in sets G & E, respectively.

→ If $W_G > W$, then do not yet take any items in set G & instead recurse on the set of items G & knapsack capacity W.

→ Otherwise ($W_G \leq W$), take all items in set G & take as much of the items in set E as will fit in the remaining capacity $W - W_G$.

→ If $W_G + W_E \geq W$ (i.e., there is no capacity left after taking all the items in set G & all the items in set E that fit in the remaining capacity $W - W_G$), then we are done.

→ Otherwise, ($W_G + W_E < W$), then after taking all the items in sets G & E, recurse on the set of items L and knapsack capacity $W - W_G - W_E$.

To analyze this algorithm, note that each recursive call takes linear time, exclusive of the time for a recursive call that it may make. When there is a recursive call, there is just one, & it's for a problem of at most half the size. Thus, the running recurrence $T(n) \leq T(n/2) + \Theta(n)$, whose solution is $T(n) = \Theta(n)$.

16.2-7: Sort A & B into monotonically decreasing order.

Consider any indices i & j such that $i < j$ & consider the terms $a_i^{b_i}$ & $a_j^{b_j}$. We want to show that it is no worse to include these terms in payoff than to include $a_i^{b_j}$ & $a_j^{b_i}$ i.e. $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$. Since A & B are sorted into monotonically decreasing order & $i < j$, we have $a_i \geq a_j$ & $b_i \geq b_j$. Since a_i & a_j are positive & $b_i - b_j$ is nonnegative, we have $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. Multiplying both sides yields $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

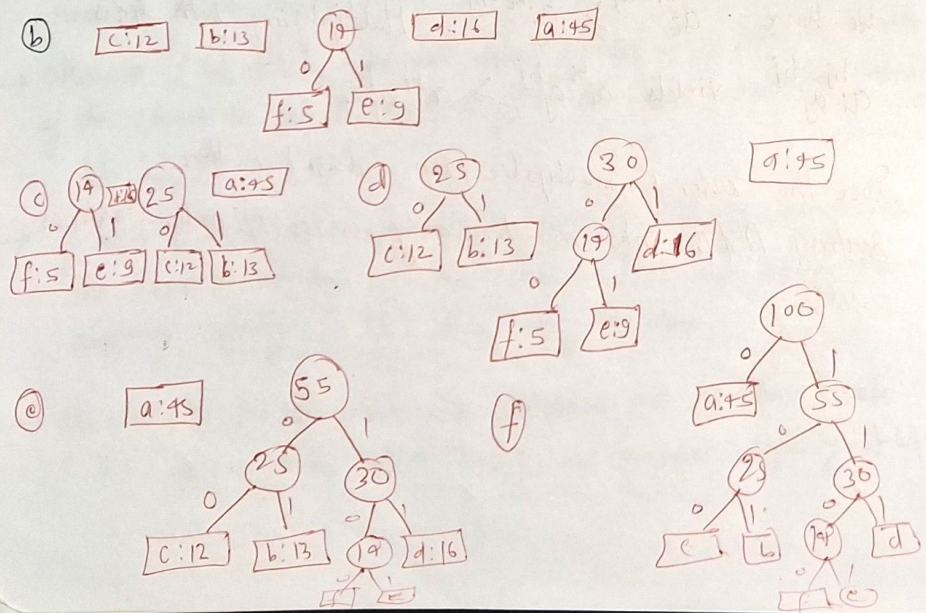
Since the order of multiplication doesn't matter, sorting A & B into monotonically increasing order work as well.

16.3 : Huffman Code — $O(n \lg n)$

Huffman invented a greedy algorithm that constructs an optimal prefix code called Huffman code.

```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n-1
      allocate a new node z
      z.left = x = EXTRACT-MIN(Q)
      z.right = y = EXTRACT-MIN(Q)
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q).
```

Figure 16.5: (a) f:5 | e:9 | c:12 | b:13 | d:16 | a:45



16.

Character	a	b	c	d	e	f
Freq.	45	13	12	16	9	5
Variable length	0	101	100	111	1101	1100
Fixed-length	000	001	010	011	100	101
	1	3	3	3	4	4

The No. of bits required = $\sum_{c \in C} c.freq \cdot d_T(c)$

$= \sum_{c \in C}$ codeword freq. * Fixed length of codeword.

$= 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4$

$= 45 + 39 + 36 + 48 + 36 + 20$

$= 224$ bits Ans

Exercise: 16.3-4: 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 say with the same parent z, then the cost of T is

$B(T) = f(x) d_T(x) + f(y) d_T(y)$

$= f(x) + f(y)$ since $d_T(x) = d_T(y) = 1$

$= f(\text{child}_1 \text{ of } z) + f(\text{child}_2 \text{ of } z)$.

Thus, the stmt of theorem is true. Now suppose $n > 2$ & also suppose that theorem is true for trees on $n-1$ leaves.

Let c_1 & c_2 are two sibling leaves in T such that they have the ~~true for~~ same parent p. Letting T be the tree obtained by deleting c_1 & c_2, by induction we know that

$$B(T) = \sum_{\text{leaves } l' \in T'} f(l') d_T(l')$$

$$= \sum_{\substack{\text{internal nodes} \\ i' \in T'}} f(\text{child}_1 \text{ of } i') + f(\text{child}_2 \text{ of } i').$$

Using this information, calculate the cost of T.

$$B(T) = \sum_{\text{leaves } l \in T} f(l) d_T(l)$$

$$= \sum_{l \neq c_1, c_2} f(l) d_T(l) + f(c_1) d_T(c_1) + 1 +$$

$$f(c_2) d_T(c_2) - 1 + f(c_1) + f(c_2)$$

$$= \sum_{\substack{\text{internal nodes} \\ i' \in T'}} f(\text{child}_1 \text{ of } i') + f(\text{child}_2 \text{ of } i') +$$

$$f(c_1) + f(c_2)$$

$$= \sum_{\substack{\text{internal nodes} \\ i \in T}} f(\text{child}_1 \text{ of } i) + f(\text{child}_2 \text{ of } i)$$

Thus stmt is true.