Topic 3: Advanced Design Techniques

Agenda

- Dynamic Programming
- Greedy Algorithms

Fibonacci's Recursion

$$F(n) = \begin{cases} n, & \text{if } n = 0, 1 \\ F(n-1) + F(n-2), & \text{if } n > 1. \end{cases}$$

$\textsc{Fibonacci-Recursive}(n)$

if $n < 2$
    return n
else
    return $\textsc{Fibonacci-Recursive}(n-1) + \textsc{Fibonacci-Recursive}(n-2)$

Run-time: $T(n) = T(n-1) + T(n-2) + 1$.

- Boundary Conditions: $T(0) = T(1) = 1$.
- Run-time is a Fibonacci sequence itself, i.e., $T(n) = \Omega(\phi^n)$, where $\phi = (\sqrt{5} + 1)/2$.
- However, if we solve it formally, we get $T(n) = \Theta(2^n)$.

Fibonacci's Recursion - Memoization

Memoization:
How about tabulating all the previous computations in an array?

Donald Michie, "Memo Functions and Machine Learning," Nature, no. 218, pp. 19-22, 1968.

FIBONACCI-MEMO(n)

if $n < 2$
  return n
else
  if $F[n]$ is undefined
    $F[n] = \text{Fibonacci-Memo}(n-1) + \text{Fibonacci-Memo}(n-2)$
  return $F[n]$

By design, $F(k)$ is evaluated only once!

- Exponential improvement: $O(n)$ additions.

Fibonacci - Dynamic Programming

Do we really need recursive calls?

```
FIBONACCI-DP(n)
F[0] = 0
F[1] = 1
for i = 2 to n
        F[n] = F[n − 1] + F[n − 2]
return F[n]
```

Dynamic Programming (DP)[1]:

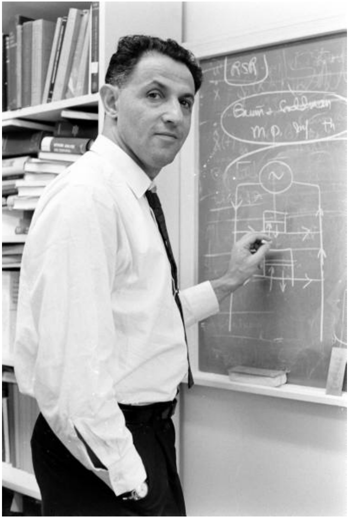
- Recursion
- Memoization

However, there is more to DP!

[1]Developed back in the day when programming meant tabular method (like linear programming). Doesnt really refer to computer programming.

Dynamic Programming

- Solves multi-stage decision problems in control theory.
- Proposed by Richard Bellman in early-1950s.



Dynamic Programming

DP is not just recursion without repetition,
it is *smart recursion*.

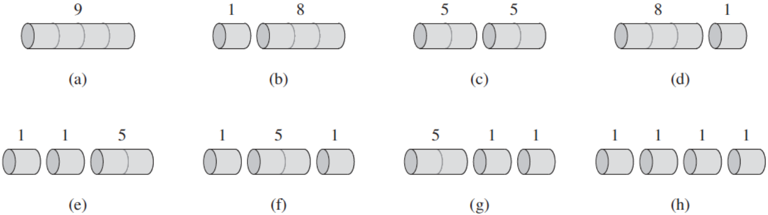
Four-step method:

1. Characterize the recursive structure in the problem.
2. Recursively define the optimal solution and its corresponding value for each subproblem.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Example 2: Rod Cutting

- Given: Rod of size n units
- p_i = Sale price of a rod of size i units.
- Goal: Maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- Assume we have zero cost for cutting.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Example 2: Rod Cutting (cont.)

- Total number of possible cuts: 2^{n-1}.
- Search over exponentially growing number of possibilities.

However, we have a structure:

- Let the rod be cut into K parts, each of size $\{i_1, \cdots, i_K\}$.
- The net revenue due to this choice is $\displaystyle\sum_{k=1}^{K} p_{i_k}$.

Goal:

$$
\begin{aligned}
r_n = \underset{\{i_1, \cdots, i_K\}}{\text{maximize}} \quad & \sum_{k=1}^{K} p_{i_k} \\
\text{subject to} \quad & 1.\ \sum_{k=1}^{K} i_k = n.
\end{aligned}
$$

Example 2: Rod Cutting (cont.)

$$
\begin{aligned}
r_n = \underset{\{i_1, \cdots, i_K\}}{\text{maximize}} \quad & \sum_{k=1}^{K} p_{i_k} \\
\text{subject to} \quad & 1. \ \sum_{k=1}^{K} i_k = n.
\end{aligned}
$$

Let us first see how the problem decomposes with each cut.

- Say, we first made a cut of size i.
- Assume $r_0 = 0$. Then, we have the following recursion:

$$
r_n = \underset{0 \leq i \leq n}{\text{maximize}} \left[p_i + r_{n-i} \right].
$$

This is the famous **Bellman equation**.

Example 2: Rod Cutting (cont.)

We can write a recursive top-down program using Bellman equation:

$\textsc{Cut-Rod}(p, n)$

```
if n = 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n − i))
return q
```

Run-time: $T(n) = 1 + \sum_{i=1}^{n} T(n-i) = O(2^n)$.

How about *memoization*?

Example 2: Rod Cutting (cont.)

While both top-down and bottom-up approaches are both identical in terms of run-time performance, bottom-up version is more elegant since we start with smaller problems first.

$\textsc{Cut-Rod-DP}(p, n)$

// Let $r[0:n]$ be a new array
$r[0] = 0$
for $j = 1$ **to** n
  $q = -\infty$
  for $i = 1$ **to** j
    $q = \max(q, p[i] + r[j-i])$
  $r[j] = q$
return $r[n]$

Note the nested loop structure is similar to $\textsc{Insertion-Sort}$. Therefore, the run-time is $\Theta(n^2)$.

A Formal Description of DP

- Let s_i denote the state of the system at the i^{th} time, for all $i = 1, \cdots, n$.
- Let x_i denote the decision made at the i^{th} time, for all $i = 1, \cdots, n-1$.

$$s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-1}} s_n$$

In other words, for all $i = 1, \cdots, n-1$, let the state transitions be denoted as

$$s_{i+1} = f_i(s_i, x_i).$$

A Formal Description of DP (cont.)

- Let the cost of changing the state from s_i to s_{i+1} be denoted as $g_i(s_i, x_i)$
- Then, the total cost is given by

$$C(x, s_1) = g_n(s_n) + \sum_{i=1}^{n-1} g_i(s_i, x_i).$$

Our goal is to *minimize* the total cost by choosing the optimal *policy* (sequence of decisions) $x = \{x_1, \cdots, x_{n-1}\}$.

$$\underset{\{x_1, \cdots, x_{n-1}\}}{\text{minimize}} \quad g_n(s_n) + \sum_{i=1}^{n-1} g_i(s_i, x_i)$$

Principle of Optimality

Let $x^* = \{x_1^*, \cdots, x_{n-1}^*\}$ denote the optimal policy.
Note that the "tail subproblem" at time k is to find the "tail policy" $\{x_k^*, \cdots, x_{n-1}^*\}$ that minimizes the "cost-to-go" (value function) from time i to time n, i.e.

$$V_k[s_k] = \underset{\{x_k, \cdots, x_{n-1}\}}{\text{minimize}} \;\; g_n(s_n) + \sum_{i=k}^{n-1} g_i(s_i, x_i)$$

Memoization: Store $V_k[s_k]$ for future computations.

Note the following recursive structure (Bellman equation):

$$V_k[s_k] = \underset{x_k}{\text{minimize}} \;\; g_k(s_k, x_k) + V_{k+1}[f(s_k, x_k)]$$

Exercise: Prove $V_1[s_1] = C(x^*, s_1)$ using induction principles.

Design Methodology for DP

Four-step method:

1. Characterize the Bellman equation for the problem.
2. Recursively find the optimal solution and its corresponding value for each tail problem.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Example 3: 0-1 Knapsack Problem

- Say, we want to store n files (where the size of i^{th} file is w_i bytes)
- However, we can recompute the i^{th} file in v_i minutes.
- We have a total memory of W bytes.

Goal: Minimize recomputing time.

Intuitively, we may store a subset of files which satisfy the following two properties:

- the combined size of these files is at most W bytes, and
- the chosen files have the largest recomputing time.

But, how do we know that this is the optimal solution?

Example 3: 0-1 Knapsack Problem (cont.)

Let S denote the set of files stored in the memory.
Formally, our goal is to find the files with maximum storage time:

$$\begin{aligned} \underset{S \subseteq \{1, \cdots, n\}}{\text{maximize}} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq W \end{aligned}$$

This is a binary program – 2^n possible subsets.

Construct an array $V[1:n, 0:W]$, where $V[i,w]$ stores the combined computing time for the files $\{1, \cdots, i\}$ with a total size of at most w bytes.

Then, the optimal solution is $V[n,W]$.

Example 3: 0-1 Knapsack Problem (cont.)

- **Policy:** $X_k = \{x_1, \cdots, x_k\}$, where $x_i \in \{0, 1\}$. Note that $x_k = 1$ only if $w_k + \sum_{i=1}^{k-1} x_i w_i \leq W$.
- **State:** Total size of stored files $W_k = \sum_{i=1}^{k} x_i w_i$.

Then, the Bellman equation is

$$\begin{aligned} V[i, w] = \quad & \underset{x_i \in \{0,1\}}{\text{maximize}} \quad x_i v_i + V[i-1, w - x_i w_i] \\ & \text{subject to} \quad x_i w_i \leq w. \end{aligned}$$

for all $1 \leq i \leq n$ and $0 \leq w \leq W$.

Example 3: 0-1 Knapsack Problem (cont.)

Note that the input arguments are

- Files: $\{1, \cdots, n\}$
- Total Available Memory: W
- Recomputing time: $\boldsymbol{v} = \{v_1, \cdots, v_n\}$
- File size: $\boldsymbol{w} = \{w_1, \cdots, w_n\}$

$\textsc{Knapsack-DP}(n, W, \boldsymbol{v}, \boldsymbol{w})$

```
V = 0 // Initialize the matrix V as an all-zero matrix of size n × (W + 1).
for i = 1 to n
    for j = 0 to W
        if w[i] ≤ j and V[i − 1, j] ≤ v_i + V[i − 1, j − w[i]]
            V[i, j] = v_i + V[i − 1, j − w[i]]
        else
            V[i, j] = V[i − 1, j]
return V[n, W]
```

Exercise: Coin change problem

If we want to make change for N cents, and we have infinite supply of each of $S = \{S_1, S_2, .., S_m\}$ valued coins, find the minimum number of coins needed to make the change?

Example: Let $N = 4$ and $S = \{1, 2, 3\}$, there are four solutions: $\{1, 1, 1, 1\}, \{1, 1, 2\}, \{2, 2\}, \{1, 3\}$. So output should be 2.

Example: For $N = 10$ and $S = \{2, 5, 3, 6\}$, there are five solutions: $\{2, 2, 2, 2, 2\}, \{2, 2, 3, 3\}, \{2, 2, 6\}, \{2, 3, 5\}, \{5, 5\}$. So the output should be 2.

Example 4: Matrix Chain Multiplication

Consider the problem of multiplying a chain of matrices as shown below:

$$A_1 \cdot A_2 \cdots A_n,$$

where the size of A_i is $p_{i-1} \times p_i$.

Note that the product of two matrices $k \times l$ and $l \times m$ is an expensive operation: $\Theta(klm)$

So, assume the cost of computing the product of matrices of sizes $k \times l$ and $l \times m$ is klm.

Goal: Find the cheapest way to compute the product of a chain of n matrices.

Example 4: Matrix Chain Multipl. (cont.)

Modeling as a multi-stage decision problem:
Iteratively pair the matrices into a product of two chains.

For example, if $n = 4$, we have three possible ways to parenthesizing the chain of matrices in the first stage:

- $(A_1 \cdot A_2 \cdot A_3) \cdot A_4$
- $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$
- $A_1 \cdot (A_2 \cdot A_3 \cdot A_4)$

$$\# \text{ Possibilities: } P(n) = \begin{cases} \displaystyle\sum_{i=1}^{n-1} P(i)P(n-i), & \text{if } n \geq 2 \\ 1, & \text{if } n = 1. \end{cases}$$

Related to Catalan numbers: $P(n) = \Omega\left(\dfrac{4^n}{n^{3/2}}\right)$.

Example 4: Matrix Chain Multipl. (cont.)

Let the sequence of decisions be denoted as $k_1, \cdots, k_L$. In such a case, since the cost of computing the product $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$ is $p_{i-1} \cdot p_k \cdot p_j$, then the problem can be formulated as

$$\underset{i \leq k_{i,j} \leq j, \ \forall (A_i \cdots A_j)}{\text{minimize}} \quad \sum_{\substack{i,j \\ i \leq j}} p_{i-1} \cdot p_{k_{i,j}} \cdot p_j$$

What are the state/decision variables?

At some intermediate step where we encounter an atrbitrary matrix chain $A_i \cdots A_j$ for some $i \leq j$, pair them as matrix chains $A_i \cdots A_{k_{i,j}}$ and $A_{k_{i,j}+1} \cdots A_j$.

Decision variable: $k_{i,j} \in \{i, \cdots, j\}$.

Example 4: Matrix Chain Multipl. (cont.)

Define a matrix M, where the $(i,j)^{th}$ entry $m[i,j]$ be the minimum cost needed to compute $A_i \cdots A_j$, i.e.,

$$m[i,j] = \begin{cases} \min\limits_{i \le k \le j} m[i,k] + m[k+1,j] + p_i p_j p_k, & \text{if } i > j \\ 0, & \text{if } i = j. \end{cases}$$

In such a case, $m[1,n]$ denotes the optimal solution to our problem.

Note: We only need a part of this matrix, as $m[i,j]$ is meaningful only when $i < j$.

Example 4: Matrix Chain Multipl. (cont.)

$$m[i,j] = \min_{i \leq k \leq j} m[i,k] + m[k+1,j] + p_i p_j p_k$$

Say, $\boldsymbol{p} = \{p_0, \cdots, p_n\}$.

MATRIX-CHAIN-ORDER($\boldsymbol{p}$)

$m = 0$ // Initialize m as an all-zero matrix of size $n \times n$
for $\ell = 2$ **to** n
    for $i = 1$ **to** $n - \ell + 1$
        $j = i + \ell - 1$
        $m[i,j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i,k] + m[k+1,j] + p_i p_j p_k$
            if $q < m[i,j]$
                $m[i,j] = q$
return $m[1,n]$

Introduction to Greedy Algorithms

Before we study greedy algorithms, let us revisit the philosophy of dynamic programming.

- Recursion in Bellman's equation:

$$V_k[s_k] = \min_{x_k} \underbrace{g_k(s_k, x_k)}_{\text{Cost of the current state transition}} + \underbrace{V_{k+1}[f(s_k, x_k)]}_{\text{Cost of the future state transition}}$$

- Consider both the current and future costs in each decision stage.
- Solve the problem in bottom-top approach.

What if, we do not care about the future costs?

Properties of Greedy Algorithms

Make whatever choice seems best at the moment and then solve the subproblem that remains.

Unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems.

Greedy choices are locally-optimal:

1. **Myopic:** Ignore all future decisions.
2. Assume there is only one decision stage (which is the current one). Then, what would be your decision?

Example 3: 0-1 Knapsack Problem

- Say, we want to store n files (where the recomputing time and size of i^{th} file are v_i mins. and w_i bytes respectively.)
- We have a total memory of W bytes.

Goal: Minimize recomputing time.

$$\begin{aligned} \underset{S \subseteq \{1, \cdots, n\}}{\text{maximize}} \quad & \sum_{i \in S} v_i \\ \text{subject to} \quad & \sum_{i \in S} w_i \leq W \end{aligned}$$

Bellman Equation: For all $1 \leq i \leq n$ and $0 \leq w \leq W$,

$$\begin{aligned} V[i, w] = \quad \underset{x_i \in \{0,1\}}{\text{maximize}} \quad & x_i v_i + V[i-1, w - x_i w_i] \\ \text{subject to} \quad & x_i w_i \leq w. \end{aligned}$$

Example 3: 0-1 Knapsack Problem (cont.)

However, a possible greedy choice could be as follows.

- Assume the files are sorted in a monotonically decreasing order of normalized recomputing time (recomputing time per byte).
- Then for all $1 \leq i \leq n$, define

$$V[i] = V[i-1] + \left(\begin{array}{ll} \underset{x_i \in \{0,1\}}{\text{maximize}} & x_i v_i \\ \\ \text{subject to} & x_i w_i \leq W - \displaystyle\sum_{k=1}^{i-1} x_k w_k. \end{array} \right)$$

Example 3: 0-1 Knapsack Problem (cont.)

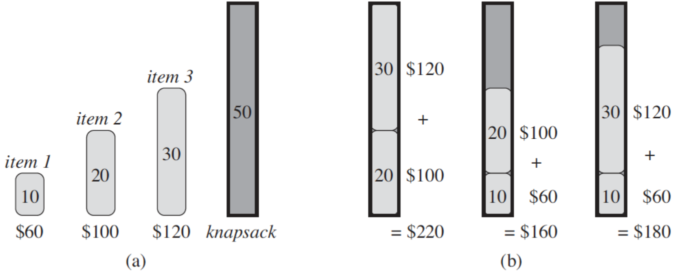
$\textsc{Knapsack-Greedy}(n, W, \boldsymbol{v}, \boldsymbol{w})$

$V = 0$
$m = W$
Sort $\boldsymbol{v}, \boldsymbol{w}$ in a decreasing order of v_i/w_i.
for $i = 1$ **to** n
    if $w[i] \leq m$
        $V = V + v_i$
        $m = m - w[i]$
return $V[n]$

Example 3: 0-1 Knapsack Problem (cont.)

Greedy solutions to 0-1 Knapsack problems are not optimal!

- Say, a thief has n items to steal (where the value and weight of i^{th} item are v_i dollars and w_i pounds respectively.)
- His knapsack can accommodate a total weight of W pounds.



Why Greedy Algorithms?

- What if, the multi-stage decision problem cannot be reduced into a Bellman equation?
- Even if we have the Bellman equation, what if there is no memory available for memoization?
- Sometimes, greedy choices may result in the optimal value.

Can greedy algorithms produce optimal solutions?

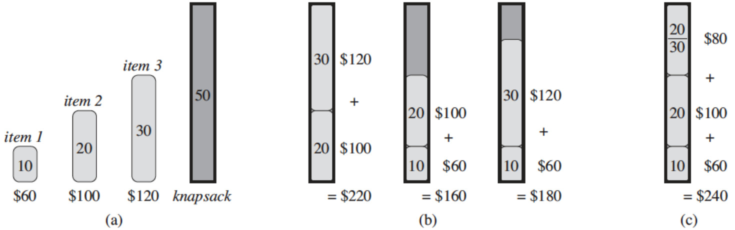
When we have a choice to make, make the one that looks best right now. Make a locally optimal choice in hope of getting a globally optimal solution.

Let us consider an example: **Fractional Knapsack**, i.e. allow fractional items to be included in the solution.

Bellman Equation: For all $1 \leq i \leq n$ and $0 \leq w \leq W$,

$$\begin{aligned} V[i,w] = \quad & \underset{x_i \in [0,1]}{\text{maximize}} \quad x_i v_i + V[i-1, w - x_i w_i] \\ & \text{subject to} \quad x_i w_i \leq w. \end{aligned}$$

Example 4: Fractional Knapsack Problem



Assuming the files are sorted in a monotonically decreasing order of normalized recomputing time (recomputing time per byte), for all $1 \leq i \leq n$, define

$$V[i] = V[i-1] + \left(\begin{array}{ll} \underset{x_i \in [0,1]}{\text{maximize}} & x_i v_i \\ \\ \text{subject to} & x_i w_i \leq W - \displaystyle\sum_{k=1}^{i-1} x_k w_k. \end{array} \right).$$

Example 5: Lossless Data Compression

Problem:

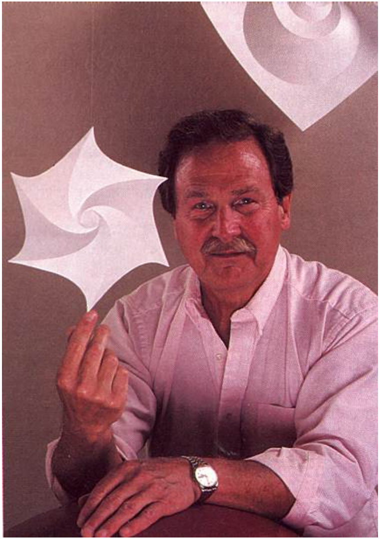
Consider a random data file which comprises of a long sequence of symbols $\{D_1, \cdots, D_n\}$, where each symbol D_i is derived from the set $S = \{s_1, \cdots, s_k\}$ for all $i = 1, \cdots, n$.
Our goal is to assign a unique binary sequence to each symbol in S such that the overall file can be represented using a minimum-length binary sequence.
Let f_i denote the frequency (probability) of occurrence of $s_i \in S$.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Solution: Huffman Coding

History of Huffman Coding

Robert M. Fano

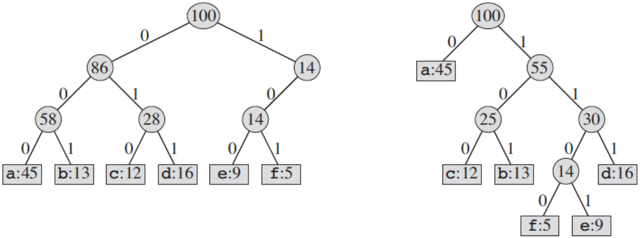


David A. Huffman

Variable-length Prefix Codes

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

For the sake of decodability, we need **prefix codes**.



(a) (b)

Example: A string 001011101 can be uniquely parsed into 0.0.101.1101, which is aabe.

Variable-length Prefix Codes (cont.)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

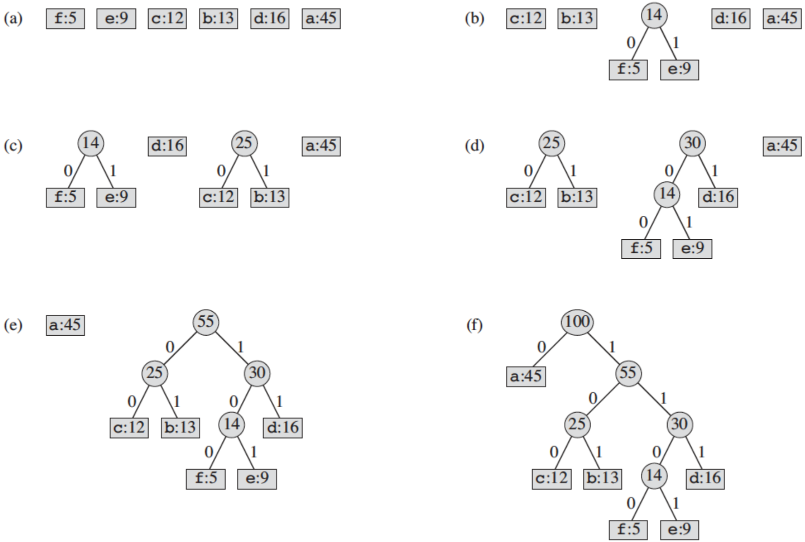
Average length of binary code per symbol for the above example:

- Fixed-length sequences $\Rightarrow$ 3
- Variable-length sequences $\Rightarrow$ 2.24 bits.

$$= 0.45 \times 1 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 4 + 0.05 \times 4.$$

Idea: Lower the frequency, larger the code (greater the depth in the prefix tree).

Huffman Coding



Huffman Coding (cont.)

$\textsc{Huffman}(S)$

$n = |S|$
$Q = S$
for $i = 1$ **to** $n - 1$
Allocate a new node z
$z.\mathit{left} = x = \mathrm{EXTRACT\text{-}MIN}(Q)$
$z.\mathit{right} = y = \mathrm{EXTRACT\text{-}MIN}(Q)$
$z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$
$\mathrm{INSERT}(Q, z)$
return $\mathrm{EXTRACT\text{-}MIN}(Q)$ // The root of the prefix tree

where $\mathrm{EXTRACT\text{-}MIN}(Q)$ removes and returns the element of Q with the smallest frequency, and $\mathrm{INSERT}(Q, z)$ inserts a node z in Q.

More formally, a greedy choice is...

$$s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-1}} s_n$$

$$\textbf{Goal:} \underset{\{x_1, \cdots, x_{n-1}\}}{\text{minimize}} \; g_n(s_n) + \sum_{i=1}^{n-1} g_i(s_i, x_i)$$

If $U(s_1)$ denotes the value obtained using the greedy algorithm, then

$$U(s_k, x_k, \cdots, x_n) = \left[\min_{x_k} \tilde{g}_k(s_k, x_k)\right] + U(s_{k+1}, x_{k+1}, \cdots, x_n)$$

for all $k = 1, 2, \cdots, n-1$ and $U(s_n) = \tilde{g}_n(s_n)$.

$$\textbf{Note:} \; U(s_1) \geq \underset{\{x_1, \cdots, x_{n-1}\}}{\text{minimize}} \; g_n(s_n) + \sum_{i=1}^{n-1} g_i(s_i, x_i)$$

Example 6: Coin change problem

If we want to make change for K cents, and we have infinite supply of each of $S = \{c_1, c_2, .., c_n\}$ valued coins, find the minimum number of coins needed to make the change?

Goal: $\underset{\boldsymbol{x}}{\text{minimize}} \displaystyle\sum_{i=1}^{n} x_i$ such that $\displaystyle\sum_{i=1}^{n} x_i c_i = K$.

Bellman Equation: Let $V[i,k]$ represent the minimum number of coins needed to make change for k cents using the coins $\{c_1, \cdots, c_i\}$. Then,

$$\begin{aligned} V[i,k] = \quad & \underset{x \in \{0,1,\cdots\}}{\text{minimize}} \quad x + V[i-1, k - c_i x] \\ & \text{subject to} \quad 1.\ c_i x \leq k \end{aligned}$$

for all $i = 1, \cdots, n$ and $k = 0, \cdots, K$.

Example 6: Coin change problem (cont.)

Greedy approach: Choose the coin with the largest denomination at each stage.

$$U[i] = 1 + U[i - c_{j^*(i)}]$$

where $c_{j^*(i)} = \{\max_j c_j$ such that $c_j \leq i\}$.

1. Let the value U be initialized to K cents, and $m = 0$.
2. Choose the largest denomination (say x) that is smaller than U, and increment m by 1.
3. Subtract x from U to find the remainder value as $U - x$.
4. If $U = 0$, print m. Else, repeat steps 2 and 3.

Optimality of Greedy Algorithms

$$s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-1}} s_n$$

Optimal Substructure: Let $V(s_1)$ denote the optimal value of the multi-stage decision problem, which is characterized using Bellman equation. In such a case,

$$V(s_k) = \min_{x_k} \left[g_k(s_k, x_k) + V(f(s_k, x_k)) \right].$$

Greedy Choice Property: *Locally-optimal* decisions result in a globally optimal sequence of decisions:

$$U(s_k) = V(s_k)$$

for all $k = 1, 2, \cdots, n-1$.

Example: Fractional Knapsack

- Let the optimal solution to the fractional knapsack problem is $\{x_1^*, \cdots, x_n^*\}$.
- Similarly, let the greedy solution to fractional knapsack problem is $\{x_1, \cdots, x_n\}$.

Case 1: Note, if $x_i = 1$ for all $i = 1, \cdots, n$, then we also have $x_i^* = 1$. This case occurs when $\sum_{i=1}^{n} w_i \leq W$.

Example: Fractional Knapsack

Case 2: Let j be the first index where $x_j \neq 1$, i.e.,

$$x_i = \begin{cases} 1, & \text{if } i = 1, \cdots, j-1 \\ x_j, & \text{if } i = j, \text{ and } x_j \in [0, 1) \\ 0, & \text{otherwise.} \end{cases}$$

Since this occurs when $\sum_{i=1}^{n} w_i > W$, any optimal solution always fills the knapsack, i.e., $\sum_{i=1}^{n} w_i x_i^* = W$.

Example: Fractional Knapsack

Let k denote the least index such that $x_k^* \neq x_k$.

Then, we have $x_k^* < x_k$ because

- If $k < j$, then $x_k = 1$. Then $x_k^* < x_k$, since $x_k^* \leq x_k = 1$.
- If $k = j$, then since $\sum_{i=1}^{j} w_i x_i = W$, and since $x_i^* < x_i$ for all $i = 1, \cdots, j$, we obtain $x_k^* = x_k$ (contradiction, since we would have $\sum_{i=1}^{n} w_i x_i^* \neq W$).
- If $k > j$, then $x_k^* = 0 = x_k$ (contradiction, otherwise we would have $\sum_{i=1}^{n} w_i x_i^* > W$).

Example: Fractional Knapsack

Suppose we increase x_k^* to x_k, and decrease as many $x_{k+1}^*, \cdots, x_n^*$ as needed. This results in a new solution: $\{y_1, \cdots, y_n\}$, where $y_i = x_i$ for all $i = 1, \cdots, k$, and

$$\sum_{k<i\leq n} w_i(x_i^* - y_i) = w_k(y_k - x_k^*).$$

Then, the total value of the solution $\{y_1, \cdots, y_n\}$ is

$$\begin{aligned}
\sum_{i=1}^{n} v_i y_i &= \sum_{i=1}^{n} v_i x_i^* + \frac{v_k}{w_k} w_k(y_k - x_k^*) - \sum_{i=k+1}^{n} \frac{v_i}{w_i} w_i(x_i^* - y_i) \\
&\geq \sum_{i=1}^{n} v_i x_i^* + \frac{v_k}{w_k}\left[w_k(y_k - x_k^*) - \sum_{i=k+1}^{n} w_i(x_i^* - y_i)\right] \\
&= \sum_{i=1}^{n} v_i x_i^*.
\end{aligned}$$

Example: Fractional Knapsack

So far, we have $\sum_{i=1}^{n} v_i y_i \geq \sum_{i=1}^{n} v_i x_i^*$.

But, if $\sum_{i=1}^{n} v_i y_i > \sum_{i=1}^{n} v_i x_i^*$, $\{x_1^*, \cdots, x_n^*\}$ cannot be optimal.

Therefore, we have $\sum_{i=1}^{n} v_i y_i = \sum_{i=1}^{n} v_i x_i^*$.

Can iteratively repeat this procedure for $k+1, k+2, \cdots$ and show that the value for $\boldsymbol{z}$ is the same as that of $\boldsymbol{x}$.
In other words, the greedy solution to fractional knapsack is optimal.