Huffman encoding:
Assume a context is available (a document, a signal, etc.). These contexts are formed by some symbols (words in a document, discrete samples from a signal, etc). Each symbols s_i is occurred f_i times in the context. We aim to encode each symbol s_i as a binary string such that the size of the encoded context is minimized (zipped file). Clearly if a symbol say s occurs very often (resp. infrequently), we want to use a relatively short (resp. long) string to represent it.
In order to simplify decoding, a nice property is that the encodings satisfy the *prefix-free* property that no codeword is the prefix of another code word.
Such an encoding is equivalent to a full ordered binary tree T; that is, a rooted binary tree where

- Every non leaf has exactly two children
- With the left edge say labeled 0 and the right edge labeled 1
- With every leaf labeled by a symbol

Then the labels along the path to a leaf define the string encoding the symbol at that leaf. The goal is to create such a tree T so as to minimize

$$cost(T) = \sum_i f_i \times (\text{depth of } s_i \text{ in } T)$$

The intuitive idea is to greedily combine the two lowest frequency symbols s_1 and s_2 to create a new symbol with frequency $f_1 + f_2$.

Example (the DPV textbook): Symbols $\{A, B, C, D\}$ with frequencies $70, 3, 20, 37$.
Obvious choice: 2 bits per symbol ($A : 00, B : 01, C : 10, D : 11$).
Total document size $= (70 + 3 + 20 + 37) \times 2 = 260$.
Pseudo-code:

Algorithm 1: Huffman algorithm

Input: An array of frequencies $f[1 \cdots n]$ such that $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$
Output: An encoding tree T with n leaves

```
Let H be a priority queue of integers, ordered by f
for i in 1 to n do
    insert(H, i)
for k = n + 1 to 2n − 1 do
    i = deletemin(H)
    j = deletemin(H)
    create a node numbered k with children i and j
    f[k] = f[i] + f[j]
    insert(H, k)
```

Symbol	Codeword
A	0
B	100
C	101
D	11

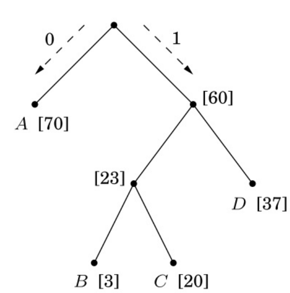

Figure 1: The best encoding tree for the aforementioned example (from DPV).

Huffman tree: (Total document size $= 70 \times 1 + 3 \times 3 + 20 \times 3 + 37 \times 2 = 213$.

Dynamic programming:

The weighted interval scheduling problem (WISP):

Interval I_i starts at s_i, finishes at f_i, and has weight (or profit) w_i. We want to identify a non-overlapping subset S ($S \subseteq \{I_1, \cdots, I_n\}$) with maximum sum of interval weights in the chosen subset.

Does greedy approaches work?

- Greedy by finish time doesn't work:

```
|-1-|           |-1-|
 |-------3-------|
```

- Greedy by max profit doesn't work:

```
|----------2----------|
 |-1-| |-1-| ... |-1-|
```

- Greedy by max unit profit doesn't work:

```
|-1-|           |-1-|
 |-------3-------|
```

The DP approach:

Sort intervals by finish time, as before ($f_1 \leqslant \cdots \leqslant f_n$). Consider optimal schedule S. There are two possibilities: $I_n \in S$ or $I_n \notin S$.

- If $I_n \in S$, rest of S must consist of optimal way to schedule intervals $I_1, ..., I_k$, where k is largest index of intervals that do not overlap with I_n (i.e., $I_{k+1}, ..., I_{n-1}$ all overlap with I_n).
- If $I_n \notin S$, S must consist of optimal way to schedule intervals $I_1, ..., I_{n-1}$.

In other words, problem has recursive structure: optimal solutions for $I_1, ..., I_n$ = optimal solution for $I_1, ..., I_{n-1}$ or optimal solution for $I_1, ..., I_k$ together with I_n, where k is largest index of interval that does not overlap with I_n.

Recursive solution:

Consider the following "semantic array":

$$V[i] = \text{max profit obtainable by a set of intervals which are a subset of the first } i \text{ intervals } \{I_1, \cdots, I_i\}$$

We can define $V[0] = 0$. Clearly $V[n]$ is the optimal value.

How to compute the entries in V? Define $pred[i]$ = the largest index j such that $f_j \leqslant s_i$ (so $I_{pred[i]}$ does not overlap with I_i but $I_{pred[i]+1}, ..., I_{i-1}$ all overlap with I_i). Assume values of $pred[]$ computed once and stored in an array.

Computing V:

$$V[0] = 0$$

$$V[i] = \max(V[i-1], V[pred(i)] + w_i)$$

Correctness: it is immediate from reasoning above: either interval I_n or don't, and since we don't know which choice leads to best schedule, just try both.

Runtime:

- Recursive: Suppose for all $i = 1, ..., n-1$, interval I_i overlaps I_{i+1} and no other I_j for $j > i+1$. Thus, the complexity would be $T[n] = T[n-1] + T[n-2]$. The solution is exponential to n (recall Fibonacci sequences).

- Iterative: There are only $n+1$ values that should be computed: $V[0], \cdots, V[n]$. Exponential runtime of recursive algorithm is due to wasted time recomputing values.
  Idea: store values in an array and compute each only once, looking it up afterwards.
  Time: $\Theta(n \log n)$ for sorting and computing $pred[i]$ values + $\Theta(n)$ for computing V values. Thus $\Theta(n \log n)$ in total.

Computing optimal answer:

```
S = {}
i = n
while i > 0 do
    if V[i] == V[i-1] // don't schedule interval I_i
    then
        i = i - 1
    else
        // schedule interval I_i
        S = S ∪ {I_i}
        i = pred[i]
return S
```

Dynamic Programming Paradigm:

- For optimization problems that satisfy the following properties:
  - *subproblem optimality*: an optimal solution to the problem can always be obtained from optimal solutions to subproblems;
  - *simple subproblems*: subproblems can be characterized precisely using a constant number of parameters (usually numerical indices);
  - *subproblem overlap*: smaller subproblems are repeated many times as part of larger problems (for efficiency).

- Step 0: Describe recursive structure of problem: how problem can be decomposed into simple subproblems and how global optimal solution relates to optimal solutions to these subproblems.

- Step 1: Define an array ("semantic array") indexed by the parameters that define subproblems, to store the optimal value for each subproblem (make sure one of the *sub*problems actually equals the whole problem).

- Step 2: Based on the recursive structure of the problem, describe a recurrence relation satisfied by the array values from step 1.

- Step 3: Write iterative algorithm to compute values in the array, in a bottom-up fashion, following recurrence from step 2.

- Step 4: Use computed array values to figure out actual solution that achieves best value (generally, describe how to modify algorithm from step 3 to be able to find answer; can require storing additional information about choices made while filling up array in Step 3).

Single-Source Shortest Paths with non-negative weights (Greedy approach):
Input: connected graph $G = (V, E)$ with **non-negative** edge weights (costs) $w(e)$ for all $e \in E$; Source $s \in V$.
Output: a path from s to all $v \in V$ with minimum total cost (*shortest* path).

Special case: if $w(e) = 1$ for all edges e: BFS!

In general: Dijkstra Algorithm (an "adjusted" BFS): use a priority queue instead of a queue to collect unvisited vertices; set priority = shortest distance so far.

Pseudo-code:

Algorithm 2: Dijkstra's shortest-path algorithm (similar to Figure 4.8 in DPV textbook)

```
Result: dist(u): The distance from s to u
// Initialization
forall the u in V do
    dist(u) = ∞          // minimum distance from s to u so far
    prev(u) = nil         // predecessor of u on the shortest path s-u so far
dist(s) = 0
H = makequeue(V)          // using dist values as keys
// Main loop
while H is not empty do
    u = deletemin(H)
    forall the edges (u, v) ∈ E do
        if dist(v) > dist(u) + w(u, v) then
            dist(v) = dist(u) + w(u, v)
            prev(v) = u
            decreasekey(H, v)
```

Runtime:

- $O(n)$ for initialization.
- n insert operations for *makequeue*
- n operations for *deletemin* (each iteration removes one vertex from the queue).
- Each iteration examines a subset of edges and updates priorities. Over all iterations, each edge generates at most one queue update.
- The time needed for queue operations depends on implementation.
- Total (using a binary heap): $O((m + n) \log n) = O(m \log n)$.

Correctness: Using the exchange technique with induction.

Single-Source Shortest Paths with real weights (Dynamic programming): Bellman-Ford Algorithm.
Input: connected graph $G = (V, E)$ with edge weights $w(e)$ for all $e \in E$ (weights can be negative but there is no negative cycle); Vertex $s \in V$.
Output: For each $v \in V$, a shortest path (i.e., minimum weight) from s to v.

Two natural ways to characterize subproblems: restrict number of edges in a path, or restrict possible vertices allowed on a path. Consider restricting edges.

Step 0: Consider a shortest path P from s to v. Since G contains no negative weight cycle, P must be simple (no cycles) so it contains at most $n-1$ edges. If P contains more than 1 edge, let u be the last vertex on P before v.

Claim: The part of P from s to u must be a shortest path in G.

Otherwise, there would be a shorter path from s to v.

Step 1: Define an array, using one index to restrict number of edges.

$A[k, v]$ = smallest weight of paths from s to v with at most k edges where $0 \leqslant k \leqslant n-1$, $v \in V$.

Step 2: Write a recurrence.

$A[0, s] = 0$

$A[0, v] = \infty$ for all $v \neq s$ (only node reachable from s with no edge is s itself)

$A[k, v] = \min(A[k-1, v], A[k-1, u] + w(u, v) : (u, v) \in E)$, for $k \in [0, ..., n-1]$ and $v \in V$ (shortest path with at most k edges either has at most $k-1$ edges or it consists of a shortest path with at most $k-1$ edges followed by one edge (u, v); examine all possible last edges (u, v) to find the best)

Step 3: Compute values bottom-up. [*will be explained next week*]

Create a two dimensional array. Each column corresponds to a vertex $v \in V$. Each row corresponds to a number from 0 to $n-1$. Calculate the values row by row. The last row contains the weight of the shortest path from s to any vertex $v \in V$.

Step 4: Find the optimal answer.

Work backwards from v. The amount of additional work required can be decreased by using a simple *trick* and storing additional information as the values are computed in Step 3. Use second array $p[v]$ to store predecessor of v on shortest path.

[Intuition: algorithm examines many possibilities to compute best value of $A[k, v]$ – remember the possibility that gave the best answer.]

Modified algorithm:

```
forall the v in V do
    A[0, v] := ∞
    p[v] := NIL
A[0, s] := 0
for k = 1 to n − 1 do
    forall the v in V do
        A[k, v] := A[k − 1, v]
        forall the edges (u, v) in E do
            if A[k − 1, u] + w(u, v) < A[k, v] then
                A[k, v] := A[k − 1, u] + w(u, v)
                p[v] := u
```

To obtain the shortest s-v path (as list of edges):

Function Path(s, v)

```
if v = s then
    return []
return [Path(s, p[v]) , (p[v], v)]
```