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

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

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**

1. Input: An array of frequencies $f[1 \cdots n]$ such that $f_1 \leq f_2 \leq \cdots \leq f_n$
2. Output: An encoding tree $T$ with $n$ leaves
3. Let $H$ be a priority queue of integers, ordered by $f$
4. for $i$ in 1 to $n$
   1. insert($H$, $i$)
5. for $k = n + 1$ to $2n - 1$
   1. $i = \text{deletemin}(H)$
   2. $j = \text{deletemin}(H)$
   3. create a node numbered $k$ with children $i$ and $j$
   4. $f[k] = f[i] + f[j]$
6. insert($H$, $k$)

**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:
  
  \[
  \begin{array}{c}
  \text{|-1-|} \\
  \text{|-1-|} \\
  \text{|-------3-------|}
  \end{array}
  \]

- Greedy by max profit doesn’t work:
  
  \[
  \begin{array}{c}
  \text{|---------2--------|} \\
  \text{|-1-| |-1-| \ldots |-1-|}
  \end{array}
  \]

- Greedy by max unit profit doesn’t work:
  
  \[
  \begin{array}{c}
  \text{|-1-|} \\
  \text{|-1-|} \\
  \text{|-------3-------|}
  \end{array}
  \]

The DP approach:
Sort intervals by finish time, as before ($f_1 \leq \cdots \leq f_n$). 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 $I_1, \ldots, I_k$, where $k$ is largest index of intervals that do not overlap with $I_n$ (i.e., $I_{k+1}, \ldots, I_{n-1}$ all overlap with $I_n$).

- If $I_n \notin S$, $S$ must consist of optimal way to schedule intervals $I_1, \ldots, I_{n-1}$.

In other words, problem has recursive structure: optimal solutions for $I_1, \ldots, I_n =$ optimal solution for $I_1, \ldots, I_{n-1}$ or optimal solution for $I_1, \ldots, I_k$ 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 $\text{pred}[i] = \text{the largest index } j \text{ such that } f_j \leq s_i$ (so $I_{\text{pred}[i]}$ does not overlap with $I_i$ but $I_{\text{pred}[i]+1}, \ldots, I_{i-1}$ all overlap with $I_i$). Assume values of $\text{pred}[]$ computed once and stored in an array.

Computing $V$:

\[
\begin{align*}
V[0] &= 0 \\
V[i] &= \max(V[i-1], V[\text{pred}(i)] + w_i)
\end{align*}
\]

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 $i = 1, \ldots, n-1$, 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 $\text{pred}[i]$ values + $\Theta(n)$ for computing $V$ values. Thus $\Theta(n \log n)$ in total.

**Computing optimal answer:**

1. $S = \{\}$
2. $i = n$
3. while $i > 0$ do
   4. if $V[i] = V[i-1]$ // don’t schedule interval $I_i$
      5. then
      6. $i = i - 1$
   7. else // schedule interval $I_i$
      8. $S = S \cup \{I_i\}$
      9. $i = \text{pred}[i]$
4. 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 subproblems 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
1 forall the $u$ in $V$ do
2 \quad $dist(u) = \infty$ // minimum distance from $s$ to $u$ so far
3 \quad $prev(u) = nil$ // predecessor of $u$ on the shortest path $s$-$u$ so far
4 $dist(s) = 0$
5 $H = makequeue(V)$ // using dist values as keys
// Main loop
6 while $H$ is not empty do
7 \quad $u = deletemin(H)$
8 \quad forall the edges $(u, v) \in E$ do
9 \quad \quad if $dist(v) > dist(u) + w(u, v)$ then
10 \quad \quad \quad $dist(v) = dist(u) + w(u, v)$
11 \quad \quad \quad $prev(v) = u$
12 \quad \quad \quad 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.

**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.

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

**Step 2:** Write a recurrence.

\[
A[0, s] = 0
\]

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

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

**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:**

```plaintext
1 for all the \( v \) in \( V \) do
2    \( A[0, v] := \infty \)
3    \( p[v] := \text{NIL} \)
4 \( A[0, s] := 0 \)
5 for \( k = 1 \) to \( n - 1 \) do
6    for all the \( v \) in \( V \) do
8        for all the edges \( (u, v) \) in \( E \) do
9            if \( A[k - 1, u] + w(u, v) < A[k, v] \) then
10                \( A[k, v] := A[k - 1, u] + w(u, v) \)
11                \( p[v] := u \)
```
To obtain the shortest $s$-$v$ path (as list of edges):

Function $\text{Path}(s, v)$

1. if $v = s$ then
2. return []
3. return [Path($s, p[v]$), ($p[v], v$)]