Greedy Algorithms

As the name indicates, greedy algorithms are *greedy* in their proceedings. At each step, the algorithm chooses the best available path irrespective of its future implications. Greedy algorithm is said to obtain *optimum* solution, if all its *local optima* leads to a *global optimum* when the program terminates. In the case when it does not, the solution is said to be *sub-optimal*. In what follows, we will see how such myopic approach to certain problems leads to elegant solutions.

Shortest Path in Graphs

Dijkstra algorithm is a greedy algorithm to compute the shortest path for a weighted digraph from a source to any vertex. At each step, the algorithm chooses the closest vertex and relaxes the corresponding edge until all vertices are explored.

Dijkstra algorithm always generates shortest path if edge weights are non-negative.
We can prove above statement using induction. Consider a partition S of nodes, where all nodes have shortest path from the source.

- **Base Case:** If the partition only contains source, shortest path is simply 0.
- **Inductive Hypothesis:** Suppose the partition contains $k-1$ nodes and all have shortest path from the source.
- **Inductive Step:** The path length to the next closest node M is equal to the edge cost plus the distance of the parent node from the source. Since distance of the parent node from the source is the shortest path (from the hypothesis), this path to M must be the shortest. Thus, k nodes have the shortest path from the source.

The performance of Dijkstra algorithm depends heavily on the priority queue implementations used. The table below shows complexities when different data structures are used.

Priority Queue	$deletemin$	$insert$	**Overall complexity**
Array	$O(\|V\|)$	$O(1)$	$O(\|V^2\|)$
Binary Heap	$O(log\|V\|)$	$O(log\|V\|)$	$O((\|V\|+\|E\|)log\|V\|)$
d-array Heap	$O(\frac{dlog\|V\|}{logd})$	$O(\frac{log\|V\|}{logd})$	$O((\|V\|\times d+\|E\|)\frac{log\|V\|}{logd})$
Fibonacci Heap	$O(log\|V\|)$	$O(1)$ (amortized)	$O(\|V\|log\|V\|+\|E\|)$

Dijkstra Algorithm:

```
def shortest_path(G, s):
    distance = dict()
    path = dict()
    known = dict()
    priorityqueue = BinaryHeap()

    for each vertex in G:                                ----
        distance[vertex] = float("inf")                      |
        path[vertex] = None                                  | O(|V|)
        known[vertex] = False                                |
                                                         ----
    distance[s] = 0                                      ----
    priorityqueue.buildheap(distance, vertices)              | O(|V|)
                                                         ----
    while priorityqueue is not empty:                    ----
        _ , v = priorityqueue.deletemin()                    |
        if known[v]: continue                                |
                                                             |
        known[v] = True                                      |
        for each vertex w adjacent to v:                     | O((|E|+|V|)log|V|)
            if distance[v] + cost(v,w) < distance[w]:        |
                distance[w] = distance[v] + cost(v,w)        |
                path[w] = v                                  |
                priorityqueue.insert((distance[w], w))       |
                                                         ----
```

Minimum Spanning Trees

A spanning tree of graph $G = (V, E)$ is a sub-graph G containing all vertices V and a sub-set of edges E $(|V| - 1)$ that touches all vertices without any cycles. It is a sparse subgraph that uncovers many properties of the underlying graph. **Minimum Spanning Tree** is the spanning tree with the least total edge cost. Two algorithms that uses greedy approach to find MST are discussed below.

Prim Algorithm

Prim Algorithm is a greedy algorithm that proceeds pretty much like Dijkstra algorithm. The distinction is in how *distance* is defined and updated. *distance* of a vertex is not its distance from the source like in Dijkstra, but rather its distance from a *known* vertex. Every time a smaller distance from a known vertex is found, *distance* is updated for that vertex. The complexity analysis is identical to that of the Dijkstra algorithm. It runs in $O((|E| + |V|)log|V|)$ when binary heap is used.

Algorithm:

```
- Initialize distance of each node to "inf" and mark it unknown
- Initialize distance of one selected node s to 0, with distance[s] = 0
- While there are unknown nodes left in the graph:
      Select the unknown node v with the lowest cost
      Mark v as known
      For each unknown node w adjacent to v:
          If cost of (v, w) < v's cost:
              v's cost = cost of (v, w)
              distance[w] = v
```

Pseudocode:

```
def mst(G,root):
    distance = dict()
    path = dict()
    known = dict()
    priorityqueue = BinaryHeap()

    for each vertex in G:                              ----
        distance[vertex] = float("inf")                    |
        path[vertex] = None                                | O(|V|)
        known[vertex] = False                              |
                                                       ----
    distance[root] = 0                                 ----
    priorityqueue = buildheap(distance, vertices)          | O(|V|)
                                                       ----
    while priorityqueue is not empty:                  ----
        _ , v = priorityqueue.deletemin()                  |
        if known[v]: continue                              |
                                                           |
        known[v] = True                                    |
        for each vertex w adjacent to v:                   | O((|E|+|V|)log|V|)
            if cost(v,w) < distance[w]:                    |
                distance[w] = cost(v,w)                    |
                path[w] = v                                |
                priorityqueue.insert((distance[w], w))     |
                                                       ----
```

Kruskal Algorithm

Kruskal algorithm is also a greedy algorithm that selects edges in the order of smallest weights and accepts the edge if it does not create a cycle. The algorithm initializes a forest containing single node trees. When an edge is accepted, the corresponding vertices are merged to form a tree. After selecting $|V| - 1$ edges in order, a single tree containing all vertices is obtained.

In order to efficiently merge trees and detect cycles, **Disjoint Set** data structure is used. The equivalence relation that guides elements in different disjoint sets is "connectivity". Two vertices v and w belong to the same disjoint set if there exists an edge (v, w) or (w, v) to connect them.

A cycle is generated in a tree T, when an edge (v, w) such that $v, w \in T$ is added. Thus, checking if an edge generats a cycle in a tree is merely checking if the corresponding two vertices are in the same disjoint set.

The Kruskal algorithm runs in $O(|E| \times log|E|)$. This is because, in the worst case, in order to select $|V| - 1$ edges, the algorithm might have to *deletemin* $|E|$ number of edges each taking $O(log|E|)$ time.

Algorithm:

```
- Put all the vertices into single node trees by themselves
- Put all the edges in a priority queue with key = edge cost
- Repeat until |V|-1 edges have been accepted:
      Extract cheapest edge
      If it forms a cycle:
          ignore it
      else:
          accept the edge – it will join two existing trees and yield a larger tree
- Return the accepted edges (they form the spanning tree)
```

Pseudocode:

```
def minimum_spanning_tree(G):
    mst = []                                                ----
    dsets = DisjointSets(all vertices in G)                     | O(|V|)
    priorityqueue = BinaryHeap()                            ----
                                                            ----
    priorityqueue.buildheap(all edges (c,v,w) in G)             | O(|E|)
                                                            ----
    while len(mst) != |V|-1:                                ----
        c,v,w = priorityqueue.deletemin()                       |
        vset = dsets.find(v)                                    |
        wset = dsets.find(w)                                    |
        if vset != wset:                                        | O(|E|.log|E|)
            dsets.union(vset, wset)                             |
            mst.append((v,w))                                   |
                                                            ----
    return mst
```

Scheduling

Consider we are given certain number of jobs $j_1, j_2, j_3, ..., j_N$ that requires $t_1, t_2, t_3, ..., t_N$ times to complete. How can we arrange jobs such that it minimizes average completion time?

Assume non-preemptive scheduling: once scheduling process is initiated, it cannot be interrupted until the completion of the process. Suppose jobs are arranged as follows: $j_1, j_2, j_3, ..., j_N$ and there is only 1 processor. The total cost C of completing the task is then given by,

$$C = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + ... + (t_1 + t_2 + ... + t_N)$$

$$C = (N + 1 - 1)t_1 + (N + 1 - 2)t_2 + (N + 1 - 3)t_3 + ... + (N + 1 - N)t_N$$

$$C = (N + 1)\sum_{k=1}^{N} t_k - \sum_{k=1}^{N} k \times t_k$$

The first term in the total cost is independent of the *ordering*. Thus, the total cost is minimized when the second term, $\sum_{k=1}^{N} k \times t_k$ is maximized. And it is maximized only when terms of the sum is *monotonically non-decreasing*. This is to say, arrange jobs in ascending order of their completion times. This greedy approach of selecting the shortest job at each step always gives the optimum solution. If there exists two jobs x, y such that $x > y$ and $t_x < t_y$, then clearly by swapping tasks the total cost C can be reduced.

The algorithm can be extended for the case of multiprocessors as well. If there are P number of processors, then P number of jobs with smallest completion times is distributed equally among P processors. The process is repeated until all jobs are completed.

On a different note, it is important to realize that a similar problem: *minimizing final completion time*, is on the other hand an NP-complete problem!

Bin Packing

Given N items of sizes $s_1, s_2, ..., s_N$ such that $0 < s_i \leq 1$, pack all items using fewest number of bins, each of unit size. Bin packing problem takes two forms.

Online Bin Packing

In online bin packing problem, each item must be placed in bin before next item can be processed.

There is no optimal online bin-packing algorithm.

An online algorithm that gives optimal solution for an input must also give optimal solutions for all intermediate input sizes. It will never know how long the input is. Suppose there is an optimal online algorithm A for an input sequence S_1 containing M small items of size $1/2 - \epsilon$ followed by M large items of size $1/2 + \epsilon$. Clearly, the optimum solution is M number of bins because in each bin you can place one small item and one large item. Now consider an intermediate input sequence S_2 containing M small items only. The optimum number of bins for such input is $\lceil M/2 \rceil$. However, the algorithm A will place each small items in separate bins to get optimum solution for sequence S_1. And in doing so, will use M bins instead of optimum $\lceil M/2 \rceil$ bins. Hence, there cannot be an optimal online algorithm for bin packing!

No online bin-packing algorithm can guarantee a sub-optimal solution using less than $4/3$ the optimal number of bins.

The above statement can be proved using the method of contradiction. So let's assume the otherwise and consider the initial sequence S_1 with M being even for simplicity. After processing first M small items, suppose the algorithm uses b bins. Using our assumption,

$$\frac{b}{M} < \frac{2}{3}$$

When all items are processed, new bins that were created after b bins must only contain only one item since all remaining items are large and two large items cannot fit into one bin. Total number of bins used is then $b + M$, which must be at least $2M - b$ when each b bins have two items. Since optimum number of bins is M for S_1,

$$\frac{2M - b}{M} < \frac{4}{3} \implies \frac{b}{M} > \frac{2}{3}$$

Clearly that contradicts with earlier inequality. Hence our assumption that the algorithm uses less than $4/3$ the optimal number of bins, must be incorrect!

Next Fit Algorithm

Check if the item can be placed in the previous bin. If it can, place it there. If not, create a new bin. The algorithm clearly runs in linear time and its sub-optimal solution is not very far from the optimal solution.

Next Fit Algorithm never uses more than twice the optimal number of bins. For certain inputs, the algorithm uses $2M - 2$ bins.

Let M be the number of optimal bins needed to pack N number of items. Consider any two adjacent bins B_j and B_{j+1}. The sum of sizes of two bins must be greater than 1; otherwise they would fit in one bin. Thus, at most half of the space is wasted. Consequently, at most twice the optimal number of bins are used.

In order to construct an example, consider N number of items with sizes $s_1, s_2, ..., s_N$ such that:

$$s_i = \begin{cases} 0.5 & if,\ i\ \%\ 2 = 0 \\ 2/N & if,\ i\ \%\ 2 \neq 0 \end{cases}$$

$$N\ \%\ 4 = 0$$

An optimal packing would be to put $N/2$ items each of size 0.5 into $N/4$ bins, each containing 2 items and the remaining $N/2$ items each of size $2/N$ into 1 bin. Thus,

$$M = \frac{N}{4} + 1$$

However, Next Fit algorithm is going to put each pair of items of weights 0.5 and $2/N$ into one bin. Hence, it will use $N/2$ number of bins, which is $2M - 2$.

First Fit Algorithm

Check if the item can be placed in any of the previous bins. If it can, place it there. If not, create a new bin. A simple implementation of the algorithm will run in $O(N^2)$ since it requires to sequentially scan all bins for each new item. However, if a balanced binary tree such as AVL tree or Red-Black tree is used, the algorithm runs in $O(NlogN)$. It can be shown that First Fit algorithm never uses more than $1.7M + 0.7$ bins.

Best Fit Algorithm

Check if the item can be placed in any of the previous bins. If it can, placed it on the bins with minimum residual capacity. Among all the bins that can accommodate the item, the tightest bin is selected. The algorithm also runs in $O(NlogN)$ using balanced binary search trees. Even though an educated choice is made compared to the First Fit algorithm, in the worst case it still performs roughly 1.7 times as bad as the optimal.

Off-line Bin Packing

In case of off-line bin packing, the algorithm is allowed to view the entire input and extract any relevant information. Below is an example of an off-line bin packing algorithm.

First Fit Decreasing

In this algorithm the input is first sorted in decreasing order by sizes and then First Fit algorithm is run on the sorted input. Large items are packed first. The algorithm does not always give optimal solution; but the sub-optimal solution is closer to the optimal solution. (more so than online algorithms)

First Fit Decreasing algorithm never uses more than $(4M + 1)/3$ bins, where M is the optimal number of bins required to pack N number of items.

Suppose $s_1, s_2, s_3, ..., s_N$ is the sorted (in decreasing order) N items. We begin with first proving two lemmas.

Lemma 1: All items that are placed in extra bins have size at most $1/3$.

We prove it using the method of contradiction. If s_i is the size of the ith item placed on $M + 1$ bin, let's assume $s_i > 1/3$. This implies: $s_1, s_2, ..., s_{i-1} \leq 1/3$, since items are placed in sorted order. So all bins $B_1, B_2, ..., B_M$ contain at most two items.

Consider the state of bins after $i - 1$ items are placed and before ith item is to be placed. Under the assumption $s_i > 1/3$, we can show that first M bins are arranged as follows: there are some j bins with only one item followed by $M - j$ bins with two items. We can show this again using contradiction. Let's say B_x and B_y are two bins such that $1 \leq x < y \leq M$ and that B_x has two items of sizes x_1, x_2 while B_y has one item of size y_1. Since x_1 is placed before y_1 and x_2 is placed before s_i,

$$(x_1 \geq y_1) \wedge (x_2 \geq s_i) \implies x_1 + x_2 \geq y_1 + s_i$$

This tells us that s_i can be placed in B_y bin, which contradicts our assumption. It follows, $s_1, s_2, ..., s_j$ items are placed in first j bins while $s_{j+1}, s_{j+2}, ..., s_{i-1}$ are placed into remaining $M - j$ bins. The optimal algorithm would have also place first j items into separate bins as no two items fit into one bin and neither does any of the $s_{j+1}, s_{j+2}, ..., s_{i-1}$ items into any of j bins (shown by first fit). Finally, we can see that the ith item of size $s_i > 1/3$ cannot be placed into any of these M bins. This means that if the first item placed on extra bin has size greater than $1/3$ then all items cannot possibly be placed into M bins. Hence, our assumption $s_i > 1/3$ contradicts the fact and must be incorrect!

Lemma 2: The number of objects placed in extra bins is at most $M - 1$.

We prove this also using method of contradiction. Let's assume M items are placed into extra bins. Since all items can fit in M bins, we know

$$\sum_{i=1}^{N} s_i \leq M$$

Let S_j be the total size of each bins B_j for $1 \leq j \leq M$. And suppose $x_1, x_2, ..., x_M$ be the sizes of M extra items. It follows,

$$\sum_{i=1}^{N} s_i \geq \sum_{j=1}^{M} S_j + \sum_{j=1}^{M} x_j$$

$$\sum_{i=1}^{N} s_i \geq \sum_{j=1}^{M} (S_j + x_j)$$

$(S_j + x_j) > 1$ for any j, since otherwise x_j would have been placed into bin B_j. It follows,

$$\sum_{i=1}^{N} s_i > \sum_{j=1}^{M} 1 > M$$

However, this is not possible as N items can be placed into M bins. It contradicts the fact and hence the assumption that there are M extra items must be incorrect!

Now we are ready to prove that the algorithm never uses more than $(4M + 1)/3$ bins. Since there are at most $M - 1$ extra items of at most $1/3$ sizes, there can be at most $\lceil (M - 1)/3 \rceil$ extra bins. The total number of bins is then given as,

$$M + \left\lceil \frac{M - 1}{3} \right\rceil = \left\lceil \frac{4M - 1}{3} \right\rceil \leq \frac{4M + 1}{3}$$

With much calculations, a tighter bound of $(11M/9 + 6/9)$ bins can be formulated as well.

Huffman Coding

Videos are captured usually at about 30 frames/sec. A single 512×512 gray scale image that typically uses 8 bits/pixel results in a 0.26 MB data. A color image would be $3 \times 0.26 = 0.78$ MB. This means, a one minute of color video requires $30 \times 60 \times 0.78 = 1.404$ GB. An hour long color video would require ≈ 85 GB of data! This begs a question:
Is there a way to compress data?

An approach to data compression is to exploit its *redundancies*. Such compression technique can be classified into two groups:

- **Lossless Compression:** Original data can be restored from the compressed data. For example, audio sampling is lossless if performed above Nyquist rate.
- **Lossy Compression:** Original data cannot be restored after compression. Quantization of data, in which one value is used to represent group of close values, is lossy.

Suppose a set of M symbols are denoted by $s_1, s_2, ..., s_M$. For images, $M = 256$ represents 256 gray scale values. For text file, $M = 128$ represents 128 ASCII characters including lower and upper cases. And so on... In order to uniquely identify each M symbols, it requires $log_2 M$ bits.

Consider a text file that contains only four characters n, e, w, s. Each symbol can then be identified by 2 bits: $00, 01, 10, 11$. Such method of coding in which symbols have fixed length is known as **Fixed-Length Coding**. It maintains unique correspondence. Each symbol takes 2 bits on average.
Now, suppose we extracted the probability distribution for each character from the file.

$$P(char = n) = 1/2$$

$$P(char = e) = 1/4$$

$$P(char = w) = 1/8$$

$$P(char = s) = 1/8$$

Based on the probability distribution, we assigned each symbols the following codewords/bits: $0, 10, 110, 111$. **The character that appears frequently is designated with a shorter codeword**. The average number of bits per symbol is then,

$$1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3 \times \frac{1}{8} + 3 \times \frac{1}{8} = 1.75$$

Thus, if a file contains 1 million symbols, we have reduced its size from 2 MB to 1.75 MB!!! Such method of coding symbols with variable lengths is known as **Variable-Length Coding**. It comes with two subtleties that are discussed below.

- **Unique Decodability:** A code is said to be *uniquely decodable* if two distinct strings never give rise to the same encoded bit sequence. If characters n, e, w, s are given the codewords $0, 10, 01, 11$, then string we and nsn gives the same encoded bit sequence 0110. It is a must property for a useful compression.
- **Instantaneous/ Prefix-Free Codes:** A code is said to be *instantaneous* if each symbol can be decoded as soon as its corresponding bit string is read. If characters n, e, w, s are given the codewords $0, 01, 011, 111$, then for a bit stream $011111...1$, there is no way to tell if the first symbol is n, e, or w (until we read the last bit 1). It is however, *uniquely decodable*. Thus, it can be observed that a code is instantaneous if no codeword is the *prefix* of another codewords. In the above example, the codeword for n is a prefix for the codewords e and w.

You can see our earlier example in which each character was assigned codewords $0, 10, 110, 111$ is *uniquely decodable* and *instantaneous/prefix-free*.

Information and Entropy

Once you make something better, there is always this question: Is there a limit to how much better can you make something? We will answer this question for the case of compression. Specifically, what is the minimum value of average number of bits per symbol?

Information is formally defined in relation to its probability of occurrence. If an event has a unit probability, then there is no information obtained upon its occurrence. The event was certain to happen. On contrary, if an event has 0 probability and it occurred, then there is great deal of information that we learn. In the intermediate case when an event has $1/2$ probability of occurrence, we received 1 bit of information upon learning that the event occurred. These properties can be formulated with the following definition of Information:

$$Information\ gained\ upon\ learning\ an\ event\ of\ probability\ p\ occurred = log_2\left(\frac{1}{p}\right)\ bits$$

Consider a stream of M symbols $s_1, s_2, ..., s_M$ with certain probability distribution $p_1, p_2, ..., p_M$ coming from a text file or a video. Consider a simplest model in which each symbol is drawn independently according to these probabilities. We can conclude,

$$0 \leq p_i \leq 1$$

$$p_1 + p_2 + ... + p_M = 1$$

Suppose we get a symbol s_i. The amount of information we obtain is then $log_2\left(\frac{1}{p_i}\right)$. The expected amount of information upon observing a symbol, known as the **entropy** H, is:

$$H = p_1 \times log_2\left(\frac{1}{p_1}\right) + p_2 \times log_2\left(\frac{1}{p_2}\right) + ... + p_M \times log_2\left(\frac{1}{p_M}\right)$$

$$H = \sum_{i=1}^{M} p_i \times log_2\left(\frac{1}{p_i}\right)$$

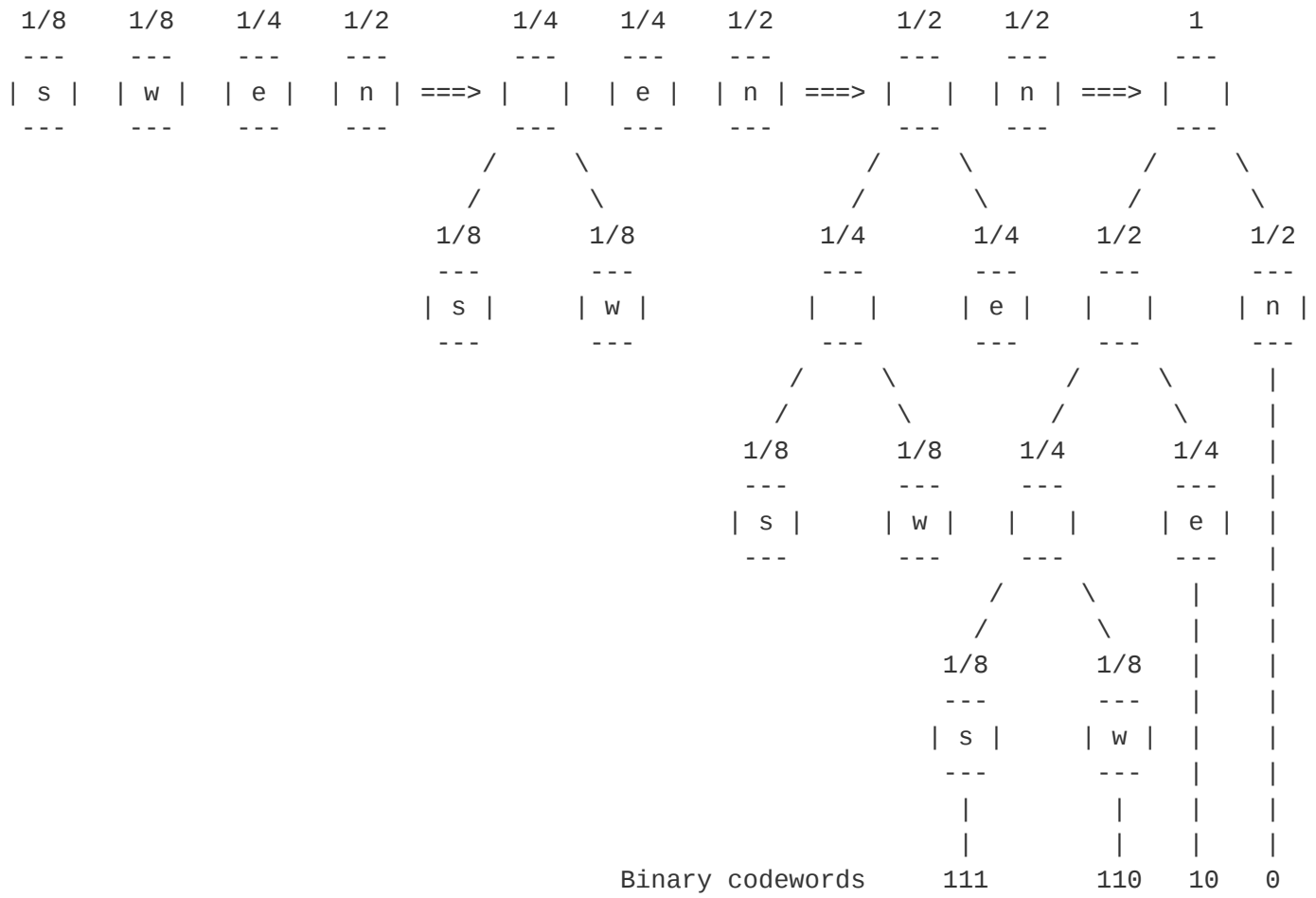
Thus entropy is the average number of bits of information per symbol from the source. In fact, as one might expect, H is also the minimum bits per symbol needed to represent a uniquely decodable source. Furthermore, according to the **Source Coding Theorem**:

- The average number of bits per symbol of any uniquely decodable source must be greater than or equal to the entropy H of the source.
- If the string of symbols is sufficiently large, there exists a uniquely decodable code for the source such that the average number of bits/symbol of the code is as close to H as desired.

Huffman coding is a simple and systematic way to design good variable-length codes given the probabilities of the symbols. Huffman coding algorithm can be described as follows:

- Maintain a forest of trees. Initialize forest with M single node trees - one for each symbol. The *weight* of a tree is equal to the sum of the probabilities of its leaves.
- $M - 1$ times, select two trees T_1 and T_2 of smallest weight (breaking ties arbitrarily) and form a new tree with sub-trees T_1 and T_2.
- After $M - 1$ selection, a single tree remains in the forest. This is the optimal Huffman coding tree.

Huffman algorithm is greedy because at each step, two trees with minimum weights are merged without regard to global considerations. Consider the example:




```
Right child move = 0
Left child move = 1
```

If trees are maintained in a priority queue, ordered by weight, then the running time is $O(MlogM)$. This is because, it would require 1 *buildheap* that takes $O(M)$, $2M-2$ *deletemin* that takes $O(MlogM)$ and $M-2$ *insert* that takes $O(MlogM)$. In case lists are used, the running time would be $O(M^2)$. For the above example, Huffman coding gives the optimal average number of bits per symbol i.e $H=1.75$. However, in general, it can be shown that:

$$H \leq Average\ length\ of\ Huffman\ code \leq H+1$$

Logical Programming / Horn Formula

Seeking a human-level intelligence in a machine would require some form of logical reasoning skill. Horn formulas are a framework for expressing local facts and deriving conclusions. Consider a puzzle:

- *If Mr. Smith has a dog, then Mrs. Brown has a cat.*
- *If Mr. Jones has a dog, then he has a cat, too.*
- *If Mr. Smith has a dog and Mr. Jones has a cat, then Mrs. Peacock has a dog.*
- *If Mrs. Brown and Mr. Jones share a pet of the same species, then Mr. Smith has a cat.*
- *All the men have dogs*

What can we say about who has what kind of pet?

The above puzzle can be translated a logical formula. Let s, j, b, p be the Boolean variables that are True if Smith, Jones, Brown, or Peacock (respectively) have a dog, and let S, J, B, P be the Boolean variables that are True if they have a cat.

$$\begin{aligned} (s &\implies B) \quad \wedge \\ (j &\implies J) \quad \wedge \\ ((s \wedge J) &\implies p) \quad \wedge \\ ((b \wedge j) &\implies S) \quad \wedge \\ ((B \wedge J) &\implies S) \quad \wedge \\ (s) \quad &\wedge \\ (j) \quad & \end{aligned}$$

Evidently, s, j, p, S, B, J must be True while others can be False. This corresponds to the conclusion that Mr. Smith and Mr. Jones must own both a cat and a dog, Mrs. Peacock must own a dog (though we don't know whether she owns a cat), and Mrs. Brown must own a cat (though we don't know whether she owns a dog).

Nomenclature:

- **Literal** is a variable x or its negation $\neg x$.
- **Clause** is a disjunction (" $\vee$ ") of literals.

$$\neg x \ \vee \ \neg y \ \vee \ z \ \vee \ k$$

- **Horn clause** is a clause with at most one positive literal.

$$Implication: \quad \neg x \ \vee \ \neg y \ \vee \ z$$

$$Pure\ negative: \quad \neg x \ \vee \ \neg y \ \vee \ \neg z$$

- **Horn formula** is a conjunction (" $\wedge$ ") of Horn clauses.

$$\begin{aligned} (\neg x \ \vee \ y) \quad & \wedge \\ (\neg z \ \vee \ \neg x \ \vee \ p) \quad & \wedge \\ (j) \quad & \end{aligned}$$

  Above clauses can also be written in the equivalent form of implication

$$\begin{aligned} (x \implies y) \quad & \wedge \\ ((z \wedge x) \implies p) \quad & \wedge \\ (j) \quad & \end{aligned}$$

We can use a greedy approach to derive a conclusion (if there is) from a given Horn Formula. We begin with initializing all variables to False. In doing so, all pure negative clauses are satisfied. The algorithm changes a variable only if an implication forces it to. The algorithm is greedy in a sense that it only cares about satisfying pure negative clauses.

On the example above, the algorithm starts by setting s to true, due to the empty implication (s). Then, B gets forced to true, due to $s \implies B$. Then j becomes true, thanks to the other empty implication. The algorithm continues by setting J, then p, and finally S. At this point all the implications are satisfied, and we conclude that s, B, j, J, p, and S must all be true.

If Horn formula has length N, then the algorithm runs in $O(N^2)$. It is the case because for each $O(N)$ iterations, it has to scan for $O(N)$ times to check for unsatisfied implications. A clever implementation, involving the use of doubly linked list for left hand side of each clause, runs in $O(N)$.

Set Cover

A newly built city is deciding on where and how many schools to build. The two constraints are: each school should be in a town and no one should have to walk for more than 10 miles to find nearest school. What is the minimum number of schools needed? Problems as such are a typical set cover problem! We begin with formalizing the set cover problem.

Suppose we are give a universe U that contains n items. Furthermore, there are k number of sets defined over n items such that $S_1, S_2, ..., S_k \subseteq U$. A set cover is a collection C of some of the sets in $S_1, S_2, ..., S_k$, whose union is the entire universe. Thus, if $\bigcup_{S_i \in C} S_i = U$, then C is a set cover. We would like to minimize $|C|$.

A greedy approach to the set cover problem is to choose, at each step, the set with most number of uncovered elements and repeat until all elements are covered. Consider an example universe where the following sets are defined:
$S_1 = \{1, 2, 3, 8, 9, 10\}, S_2 = \{1, 2, 3, 4, 5\}, S_3 = \{4, 5, 7\}, S_4 = \{5, 6, 7\}, S_5 = \{6, 7, 8, 9, 10\}$
. The greedy algorithm begins with choosing the largest set $S_1 = \{1, 2, 3, 8, 9, 10\}$. The remaining sets are update to only contain uncovered elements. Thus, we are left with:
$S_2 = \{4, 5\}, S_3 = \{4, 5, 7\}, S_4 = \{5, 6, 7\}, S_5 = \{6, 7\}$. The algorithm then could choose $\{4, 5, 7\}$ followed by $\{6\}$.

The greedy algorithm chose 3 sets, which clearly is not optimal. The optimal solution is choosing 2 sets: $S_2 = \{1, 2, 3, 4, 5\}$ and $S_5 = \{6, 7, 8, 9, 10\}$. Thus greedy algorithm for the set cover problem does not always give the optimal solution. However, the sub-optimal solution is not too far from the optimal solution.

If the optimal number of sets for n items is m, then the greedy algorithm finds a set cover with at most $m \times log_e n$ sets.
The first set that the greedy algorithm chooses has size at least n/m. The number of remaining elements still uncovered is then,

$$n_1 \leq n - n/m = n(1 - 1/m)$$

Since m is the optimal number of sets, at least one of the remaining sets must contain $n_1/(m-1)$ items. Thus after second pick, the number of remaining elements still uncovered is

$$n_2 \leq n_1 - n_1/(m-1) = n_1(1 - 1/(m-1))$$

$$n_2 \leq n_1(1 - 1/m) \leq n(1 - 1/m)^2$$

Thus after ith pick, the remaining elements still uncovered is,

$$n_i \leq n_{i-1}(1-1/m) \leq n(1-1/m)^i$$

Suppose, the greedy algorithm picked k number of sets to cover all the elements. The remaining elements after kth pick, $n(1-1/m)^k$, must then be less than 1.

$$n(1-1/m)^k < 1$$

$$(1-1/m)^{m \times \frac{k}{m}} < 1/n$$

Using the relation, $(1-x)^{\frac{1}{x}} \approx 1/e$.

$$e^{-\frac{k}{m}} < 1/n$$

$$k < m \times log_e n$$

Thus, the size of the set cover obtained by the greedy algorithm is bounded by $mlog_e n$.

An approximation factor of a greedy algorithm is defined as the maximum ratio between the greedy algorithm solution and the optimal solution. The approximation factor of a greedy algorithm for the set cover problem is $log_e n$. Furthermore, assuming a widely held view that $P \neq NP$, there are provably no polynomial-time algorithms with smaller approximation factor. So, unless $P = NP$, there is no hope for better algorithm!