Greedy Algorithms

Non-optimization problem

One correct solution or at least any correct solution is good enough

- Sorting
- Connected components
- Stable marriages
- ...

Optimization problem

Multiple correct (feasible) solutions, try to find the best one

- Minimum spanning tree
- Shortest paths
- Minimum-length codes
- Minimum vertex coves
- ...

Greedy algorithms solve optimization problems by making locally beneficial (greedy) choices. These choices are often natural. Proving that the algorithm finds an optimal solution is usually more challenging.

Not every optimization problem can be solved optimally using a greedy algorithm. For problems that can't, greedy algorithms, if designed carefully, often yield very good solutions.

Introductory Example: Interval Scheduling

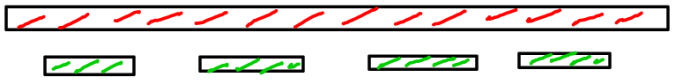
Given a set of activities, each with a starting time s_i and an ending time e_i, and all of them competing for the same resource, schedule as many as possible of them without conflicts.

More precisely, any two scheduled activities cannot overlap: their intervals $[s_i, e_i]$ and $[s_j, e_j]$ need to be disjoint.

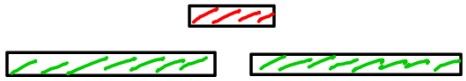
Possible application: Resource = lecture hall, activities = classes to be held in the lecture hall.

Natural strategies:

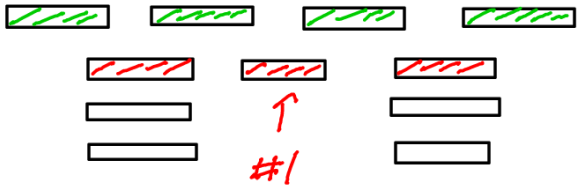
- Choose the interval that starts first



- Choose the shortest interval



- Choose the interval with the fewest conflicts



Basic idea: Always choose the interval that has the least chance of conflicting with other intervals.

The one that works: Choose the interval that ends first

Interval Schedules (I)

Sort the intervals in I by their ending times
$e = -\infty$ // ending time of last scheduled interval
$S = \emptyset$ // set of scheduled intervals
for every interval $(s_i, e_i) \in I$ do
  if $s_i > e$ then
    $S = S \cup \{(s_i, e_i)\}$
    $e = e_i$
return S

Obs: An element (s_i, e_i) in S conflicts with an already scheduled element iff $s_i \leq e$. Thus, the above algorithm implements the strategy of scheduling the feasible interval that ends first in each step.

Proof: We have $e = \max\{e_j \mid (s_j, e_j) \in S\}$. Thus, $e_i \geq e \geq e_j$ for all $(s_j, e_j) \in S$ and there exists an interval $(s_k, e_k) \in S$ with $e_k = e$. If $[s_i, e_i] \cap [s_j, e_j] \neq \emptyset$, this implies that $s_i \leq e_j \leq e$. If $s_i \leq e = e_k$, then, since $e_i \geq e$, $[s_i, e_i] \cap [s_k, e_k] \neq \emptyset$. □

We need to prove that picking the next interval to add to S as the interval in I that ends first among all intervals that do not conflict with S maximizes the number of intervals we end up adding to S.

We employ a standard device used in the analysis of greedy algorithms: We consider an optimal solution and prove that the solution constructed by our algorithm after each step is no worse than

the optimal solution in a sense that depends on the problem.

Lemma: Let $(s_1, e_1), (s_2, e_2), \ldots, (s_k, e_k)$ be the solution constructed by the above algorithm, and let $(s_1^*, e_1^*), (s_2^*, e_2^*), \ldots, (s_\ell^*, e_\ell^*)$ be an optimal solution, that is, one that maximizes ℓ. Let the intervals in the optimal solution be sorted by increasing ending times. Then $k = \ell$ and, for $1 \le i \le k$, $(s_1, e_1), \ldots, (s_i, e_i)$ do not conflict with (s_{i+1}^*, e_{i+1}^*).

Proof: It is obvious that $k \le \ell$ because $(s_1, e_1), \ldots, (s_k, e_k)$ is a feasible solution and $(s_1^*, e_1^*), \ldots, (s_\ell^*, e_\ell^*)$ is an optimal solution. The claim that $k \ge \ell$ follows from the second part of the lemma. Indeed, if $k < \ell$, then (s_{k+1}^*, e_{k+1}^*) does not conflict with $(s_1, e_1), \ldots, (s_k, e_k)$, so the algorithm would be able to add a $(k+1)$st interval to the solution.

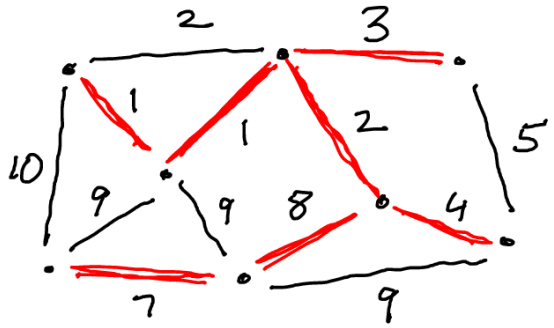
Now let us prove that $(s_1, e_1), \ldots, (s_i, e_i)$ do not conflict with (s_{i+1}^*, e_{i+1}^*). This is obvious for $i = 0$. For $i > 0$, assume the claim holds for $i-1$. Then (s_i^*, e_i^*) is an interval the algorithm can choose in the ith step. Since it chooses the interval that ends first, we have $e_i \le e_i^*$. Since (s_i^*, e_i^*) and (s_{i+1}^*, e_{i+1}^*) do not conflict, we have $s_{i+1}^* > e_i^* \ge e_i$. Since (s_i, e_i) is chosen in step i, it was also a candidate in steps $1, \ldots, i-1$. Since it was not chosen, we have $e_j \le e_i \ \forall\, 1 \le j \le i$. Thus, $e_j < s_{i+1}^* \ \forall\, 1 \le j \le i$, that is, (s_{i+1}^*, e_{i+1}^*) does not conflict with $(s_1, e_1), \ldots, (s_i, e_i)$. □

The key in the proof was to show that $e_i \leq e_i^*$ $\forall i$, which implies that the greedy algorithm has no more conflicts with subsequent intervals than the optimal solution, so the greedy algorithm always has at least as many choices left in each step as the optimal algorithm.

Exercise: Prove that the algorithm takes $O(n \lg n)$ time.

Minimum spanning tree

Given a graph G whose edges have weights a *minimum spanning tree (MST)* is a spanning tree whose edges have the smallest possible total weight among all spanning trees of G.



What's a natural greedy choice?

We want a graph that's connected and acyclic (a tree). So we start with the empty edge set and repeatedly pick an edge whose endpoints are currently in different connected components. Since we want an MST, we always pick the cheapest such edge.

This gives us Kruskal's algorithm:

```
Kruskal(G)
Sort the edges by increasing weight
T = ∅
for every edge (x,y) in the sorted list do
    if x and y are disconnected in T then
        add (x,y) to T
```

Lemma: *Kruskal's algorithm computes an MST of G.*

To prove this, we need the following theorem.

Cut Theorem: *Let (U,W) be a partition of the vertex set of G into two disjoint subsets, and let e be the lightest edge with one endpoint in each set. Then there exists an MST of G that includes e.*

Proof: Let T be an MST of G, let $e=(u,v)$, and let P be the path from u to v in T. Since P has one endpoint in U and one in W, it has an edge $e'=(u',v')$ with $u' \in U$ and $v' \in W$. By the choice of e, $w(e) \leq w(e')$. Let $T' = T - \{e'\} \cup \{e\}$. Then $w(T') = w(T) - w(e') + w(e) \leq w(T)$. Thus, T' is an MST (and includes e) if it is in fact a tree. This, however, is easy to see: $T \cup \{e\}$ contains exactly one cycle formed by P and e and this cycle is broken in T' by removing e'. Thus, T' is acyclic. T' is connected because T is, so $T \cup \{e\}$ is, and removing an edge from a cycle does not disconnect the graph. □

The proof of the cut theorem actually shows a stronger claim, which we need to prove that Kruskal's algorithm is correct:

Theorem: Let (U, W) be a partition of the vertex set of G into two disjoint subsets, let e be the lightest edge with one endpoint in each set, let T be an MST of G, and let E_0 be the set of edges in T that have both endpoints in U or both endpoints in W. Then there exists an MST T' of G that includes the edges in $E_0 \cup \{e\}$.

Correctness proof of Kruskal: First we prove that Kruskal computes a spanning tree. Assume the graph T it produces contains a cycle C, and let e be the last edge in C added to T. Then the endpoints of e are connected in T by the time we add e to T. Similarly, if T is disconnected, then there exists an edge e in G whose endpoints are disconnected in T. Since they are disconnected in T when the algorithm finishes, they are disconnected when the algorithm inspects e. Then, we could have added e to T too. This proves that T is connected and contains no cycles, that is, T is a tree. Since T contains all vertices of G, T is a spanning tree.

Now let $\langle e_1, e_2, \ldots, e_{n-1} \rangle$ be the sequence of edges added to T. We prove by induction on i that there exists an MST of G that includes the edges in $E_i := \{e_1, e_2, \ldots, e_i\}$. In particular, there exists

an MST that includes the edges $e_1, e_2, \ldots, e_{i-1}$, that is, T is an MST of G.

For $i=0$, the claim holds trivially. For $i>0$, let T_{i-1} be the subgraph of T with edge set $\{e_1, e_2, \ldots, e_{i-1}\}$, and let U be the vertex set of the connected component of T_{i-1} containing one endpoint of e. Let $W := V \setminus U$. By the inductive hypothesis, there exists an MST of G that includes the edges in E_{i-1}. Every edge in E_{i-1} has both endpoints in U or both endpoints in W. Thus, for the lightest edge e with one endpoint in U and one endpoint in W, there exists an MST of G that includes the edges in $E_{i-1} \cup \{e\}$. It remains to prove that e_i is such a lightest edge. Indeed, all vertices in U are disconnected from all vertices in W in T_{i-1}. Thus, any edge e_j with $j<i$ and with one endpoint in U and one endpoint in W would have been added to T_{i-1} before inspecting e_i. Thus, every edge e_j with one endpoint in U and one endpoint in W satisfies $j \geq i$ and is thus no lighter than e_i. □

The running time of Kruskal's algorithm depends on the data structure we use to represent the connected components of T. Initially, every vertex is in its own component (because T has no edges). The data structure needs to support two operations: test whether two vertices belong to the same component and merge two components when adding an edge

between them. This is an application of the classical union-find problem:

Union-find: Maintain a partition of a set S into subsets $S_1, S_2, \ldots, S_k$. Initially, every element of S is in its own set. We need to support two operations:

- Find(x) identifies the set S_i that contains x
- Union(x,y) replaces the set S_i and S_j containing x and y with a new set $S_i' = S_i \cup S_j$, that is, it merges S_i and S_j.

In Kruskal's algorithm, S is the vertex set of G and $S_1, S_2, \ldots, S_k$ are the vertex sets of the connected components of T. This gives the following concrete implementation of Kruskal's algorithm:

```
Kruskal(G)
Sort the edges by increasing weight
T = ∅
for every edge (x,y) in the sorted list do
    if Find(x) ≠ Find(y) then
        add (x,y) to T
        Union(x,y)
```

Lemma: The running time of Kruskal's algorithm is $O(m \lg n + U)$, where U is the cost of all operations it performs on the union-find data structure.

The number of operations we perform on the union-

find data structure is at most $3m$ ($2m+n-1$ to be precise, but $3m$ is trivial to see). The textbook describes a sophisticated data structure that ensures the cost of these operations is at most $O(m + n\alpha(n))$, where $\alpha(\cdot)$ is the inverse of Ackerman's function. Even for truly insanely large inputs, $\alpha(n)$ is a very small constant, so in practice, the cost is $O(n+m)$. Since we already have an $O(m \lg n)$ cost in Kruskal's algorithm because we need to sort the edges by their weights, a cost of $O(m + n \lg n)$ for the operations on the union-find data structure is more than good enough. We discuss such a data structure next, which gives

Lemma: *Kruskal's algorithm takes $O(m \lg n)$ time.*

Union-Find

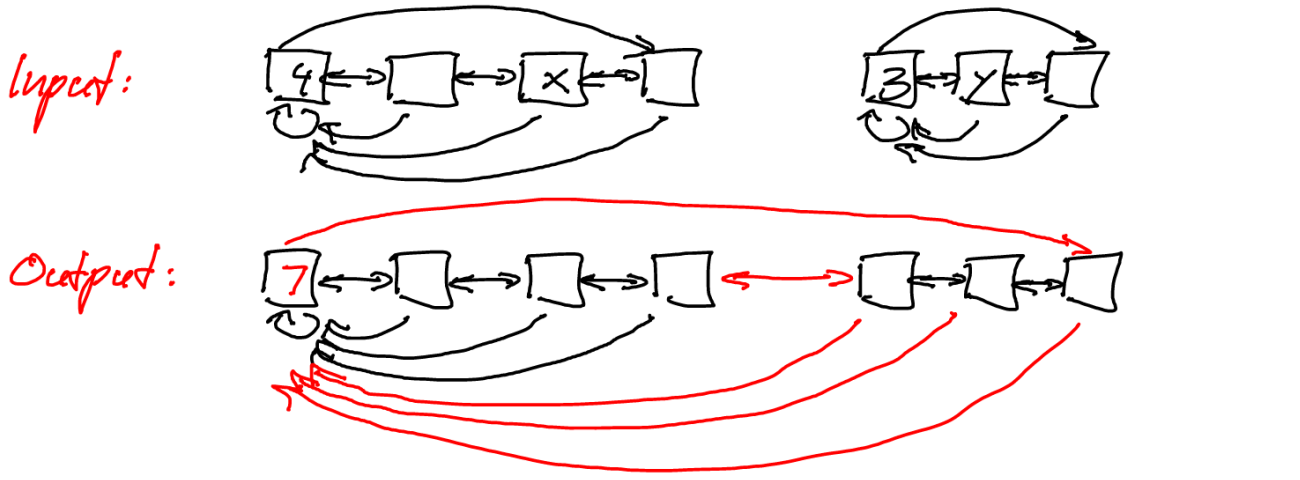
We represent each set S_i as a doubly-linked list. In addition, the head of each list stores a pointer to the tail and every node in the list stores a pointer to the head. The head also stores the size of the list


```
Find(x)
return head(x)
```

Clearly, Find(x) = Find(y) if and only if x and y belong to the same list, so this operation is correct. It also obviously takes constant time.

A Union(x,y) operation concatenates the shorter of the two lists onto the longer one. This takes $O(1+\ell)$ time, where ℓ is the length of the shorter list:

```
Union(x, y)
 if Find(x) = Find(y) then return
 if size(head(x)) < size(head(y)) then
     swap x and y
 Concatenate x and y's lists
 size(head(x)) = size(head(x)) + size(head(y))
 tail(head(x)) = tail(head(y))
 for every node z in y's list do
     head(z) = head(x)
```



(Red values are changed by the union procedure)

Both Union and Find take constant time if we ignore the cost of concatenating x and y's list. Since

we perform $O(m)$ such operations, this contributes $O(m)$ time to the cost of Kruskal's algorithm. The cost of concatenating two lists is $O(\ell)$, where ℓ is the length of the shorter list. By the next lemma, the total cost of concatenating lists is $O(n \lg n)$ because there are n nodes in the lists in total.

Lemma: *Every node in the union-find data structure participates in at most $\lg n$ union operations as a member of the shorter list.*

Proof: We prove by induction on i that, if a node has participated in i union operations as a member of the shorter list, then the list it belongs to has size at least 2^i. Since a node cannot belong to a list with more than n nodes (there are only n nodes in total), no node is involved in more than $\lg n$ union operations.

For $i = 0$, the claim holds trivially because the list contains the node itself and thus has size $\geq 1 = 2^0$.

For $i > 0$, the list containing the node before the ith union operation has size $\geq 2^{i-1}$, by the inductive hypothesis. Since it is the shorter of the two lists being merged, the other list also has at least 2^{i-1} elements. Merging these two lists thus produces a list of size at least 2^i. □

Back to MST - Prim's algorithm

Kruskal's algorithm grows the MST one edge at a time while maintaining the invariant that the current graph T is a spanning graph of G (contains all vertices of G) and is a forest.

A different strategy is to maintain that T is a tree and to add vertices and edges to it until it is a spanning tree of G. The greedy choice is once again to pick the cheapest edge to add in each step:

Greedy choice: *In each step, choose the cheapest edge e connecting a vertex $u \in T$ to a vertex $w \notin T$ and add w and e to T.*

This gives us Prim's algorithm, another graph exploration variant:

```
Prim(G)
  Pick a root r
  T = ({r}, ∅)
  Q = an empty priority queue
Mark r as explored and all other vertices as unexplored
for every edge (r,w) ∈ Adj(r) do
    Insert(Q, (r,w), w((r,w)))
while Q is not empty do
    (u,w) = DeleteMin(Q)
    if w is unexplored then
        Mark w as explored
        Add w and the edge (u,w) to T
```

```
for every edge (w,x) ∈ Adj(w) do
    Insert(Q, (w,x), ω((w,x)))
```

The algorithm uses a priority queue, which is an ADT that supports the following operations:

- IsEmpty(Q): Test whether Q is empty
- Insert(Q,x,p): Insert x into Q with priority p. (Assumes $x \notin Q$.)
- DecreaseKey(Q,x,p): Requires that $x \in Q$ and x has priority p'. Replaces x's priority with $\min(p,p')$.
- DeleteMin(Q): Deletes and returns the element with minimum priority in Q.

Since Prim's algorithm is a variant of graph exploration, we know it computes a spanning tree T of G. We need to prove that it is an MST.

Lemma: *Prim's algorithm computes an MST T of G.*

Proof: It is easy to verify the following two statements:

(i) At all times, every edge in Q has at least one endpoint in T. Q contains all edges with exactly one endpoint in T (and possibly additional edges with both endpoints in T).

(ii) When adding an edge e to T, e has minimum weight among all edges with exactly one endpoint in T (because e has minimum weight among the edges in Q and, by (i), Q contains all edges with exactly one endpoint in T and no edges without an endpoint in T).

Now let $T_0 \subset T_1 \subset \dots \subset T_{n-1} = T$ be the sequence of trees produced by the algorithm, where T_i is the tree obtained after adding i edges. Using the cut theorem, we can once again prove that there exists an MST $T' \supseteq T_i$ $\forall i$. Since T is a spanning tree of G it is thus an MST of G.

Clearly, there exists an MST $T' \supseteq T_0$ because T_0 has no edges. For $i > 0$, assume there exists an MST $T' \supseteq T_{i-1}$. Let U be the vertex set of T_{i-1} and $W := V \setminus U$. Edge e_i is the cheapest edge with one endpoint in U and one in W. Moreover, all edges in T_{i-1} have both endpoints in U. Thus, there exists an MST T'' that includes e_i and all edges in T_{i-1}, that is, $T'' \supseteq T_i$. □

Again, an efficient implementation of Prim's algorithm depends on an efficient priority queue. Using a binary heap, for example, all priority queue operations take $O(\lg n)$ time, so Prim's algorithm takes $O(m \lg n)$ time. We can do better, however. The key is to change the algorithm and to use a better priority queue.

Change 1: Use a priority queue of vertices instead of a priority queue of edges

Obs: If a vertex $w \notin T$ is connected to vertices in T using edges $e_1, e_2, \ldots, e_k$, Prim's algorithm adds at most one of these edges to T, namely the cheapest one.

Thus, we can keep the vertices not in T in the priority queue. The priority of each vertex is the weight of the cheapest edge connecting it to T or ∞ if no such edge exists. This leads to the following algorithm.

```
Prim(G)
Choose some root r
Mark r as explored and all other vertices as
unexplored
T = ({r}, ∅)
Q = an empty priority queue
for every neighbour w of r do
    Insert(Q, w, ω((r,w)))
    p(w) = r
while Q ≠ ∅ do
    v = DeleteMin(Q)
    Add v and edge (p(v), v) to T
    Mark v as explored
    for every neighbour w of v do
        if w is unexplored then
            if w ∉ Q then
                Insert(Q, w, ω((v,w)))
```

```
        p(w) = v
    else if w((v,w)) < w((p(w),w)) then
        DecreaseKey (Q, w, w((v,w)))
        p(w) = v
```

The change seems cosmetic because both versions of Prim's algorithm perform $\Theta(m)$ priority queue operations and otherwise take $O(n+m)$ time. The important difference is that the previous version performed $\Theta(m)$ insertions and $\Theta(m)$ DeleteMin operations, while the new version performs n Insert, n DeleteMin, and $\Theta(m)$ DecreaseKey operations. Using a binary heap as a priority queue, this change is not significant because all operations take $O(\lg n)$ time. Next we discuss a priority queue that takes $O(n \lg n + m)$ time to perform a sequence of m operations of which n are DeleteMin operations. Thus, Prim's algorithm takes $O(n \lg n + m)$ time using this priority queue, which is $o(m \lg n)$ as long as $m \in \omega(n)$.

Thin heaps

A thin heap is a circular list of heap-ordered thin trees.

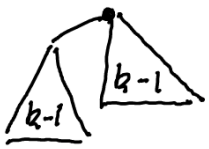
A tree storing elements at its nodes is **heap-ordered** if the element at each node is no less than the element at its parent. Thus, the root stores the smallest element, very useful for DeleteMin operations.

A thin tree is a relaxed variant of a binomial tree. So let's define the latter first. Binomial trees are defined inductively. A **binomial tree of rank 0** has a single node. A **binomial tree of rank k**, for $k > 0$, is obtained by taking two binomial trees of rank $k-1$ and making the root of one tree leftmost child of the root of the other.

Rank 0

•

Rank k



Observation: A binomial tree of rank k has size 2^k. The children of the root are roots of binomial trees of rank $k-1, k-2, \ldots, 0$.

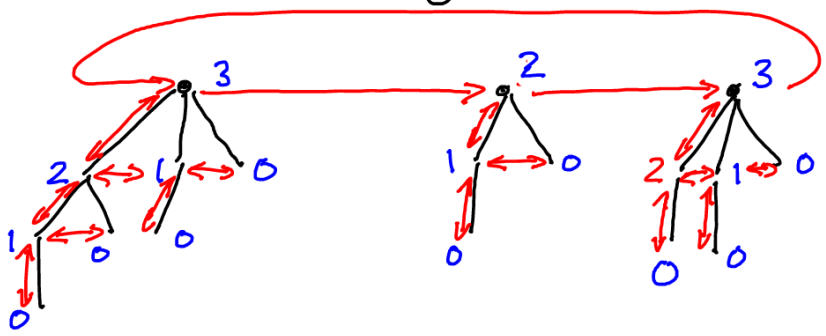
If we require that each tree in the heap is a binomial tree, we obtain a binomial heap. The textbook discusses them in more detail and shows that each priority queue operation takes $O(\lg n)$ time, just as on binary heaps, but we want faster operations. The textbook also discusses Fibonacci heaps, a different relaxation of binomial heaps that, up to constant factors, achieve the same performance as thin heaps. Fibonacci heaps were invented first, but thin heaps are simpler and faster in practice.

A thin tree is similar to a binomial tree, but its nodes may be **thin** or **thick**. A thick node of rank k has k children of ranks $k-1, k-2, \ldots, 0$.

A thin node of rank k has k-1 children of ranks k-2, k-3, ..., 0, that is, it is a thick node that has lost its leftmost child. A binomial tree is thus a thin tree with only thick nodes. The root of a thin node is always thick.

To represent each node in a thin heap, we need the following fields:

- the element it stores
- the rank
- a pointer to its leftmost child
- a pointer to its right sibling
- a pointer to its parent or left sibling. This pointer is null for every root.



Black edges are how we visualize the tree. Red edges are the pointers nodes store. The ranks of thick nodes are shown in blue, those of thin nodes are red.

In a thin heap, the minimum element is stored at one of the roots (because the trees are heap-ordered). We store a pointer to this root and its successor in the root list. Call these pointers min and succ.

Next we discuss the implementation of priority queue operations:

IsEmpty (Q): Test whether the root list is empty, i.e., whether min is null.

Insert (Q,x): If Q is empty, create a new node storing x, make it its own successor in the root list and make min and succ point to it. The new node has rank 0.

If Q is not empty, create a new root node of rank 0 and insert it between min and succ. If $x < min$, then make min point to x. Otherwise, make succ point to x.

Delete (Q,x): DecreaseKey (Q, x, $-\infty$), then DeleteMin (Q)

DeleteMin (Q): This operation returns the element stored at min. Before doing this, it needs to update the root list, and the min and succ pointers. We collect all roots other than min. We collect the children of min and make them thick if necessary by decreasing their ranks by one. These nodes become the new roots of Q. There may be too many of them. Thus, we repeat the following process: While there are two roots of equal rank k, make the one storing the greater element the leftmost child of the one storing the smaller element and increase the latter's rank to k+1. Collect the remaining roots, link them to form a circular list and make min and succ once again point to the root storing the smallest element and to its successor, respectively.

By the following lemma, all nodes involved in the merging process have rank less than $2 \lg n$, so the merging process can be implemented in constant time per involved root by storing an array of length $2 \lg n$ whose kth entry points to null or to a root of rank k.

Lemma: A thin tree whose root has rank k has size at least φ^{k-1}, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Proof: We prove that the tree has at least F_k nodes, where F_k is the kth Fibonacci number. These numbers are defined as

$$F_k = \begin{cases} 1 & k = 0 \text{ or } k = 1 \\ F_{k-1} + F_{k-2} & k \geq 2 \end{cases}$$

This implies the lemma because it is easy to show by induction that $F_k \geq \varphi^{k-1}$. For $k = 0$ and $k = 1$, $F_k = 1$ and $\varphi^{k-1} \leq 1$. For $k \geq 2$, we obtain

$$\begin{aligned} F_k = F_{k-1} + F_{k-2} &\geq \varphi^{k-2} + \varphi^{k-3} \\ &= \left(\frac{1+\sqrt{5}}{2} + 1\right) \varphi^{k-3} \\ &= \left(\frac{3+\sqrt{5}}{2}\right) \varphi^{k-3} \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^2 \varphi^{k-3} \\ &= \varphi^{k-1} \end{aligned}$$

A thin tree of rank 0 or 1 has at least one node, so it has at least F_k nodes.

For $k \geq 2$, the root has at least $k-1$ children of ranks $k-2, k-3, \ldots, 0$. The root together with the last $k-2$ children form a thin tree of rank $k-1$ (whose root is thin). Thus, there are at least F_{k-1} nodes in this portion of the tree. The leftmost subtree has at least F_{k-2} nodes. Thus, we have at least $F_{k-1} + F_{k-2} = F_k$ nodes in total. □

DecreaseKey (Q, x, p): We start by updating x's priority. If x is a root, we update min and succ if x now has the smallest priority. If x is not a root, we remove it from the child list of its parent, make it a root, and then proceed as above. When making x a root, we decrease its rank by one if necessary to make x thick. Removing x from the child list of its parent may result in a violation of the rank conditions of a thin heap. Let y be x's left sibling or its parent if it has no left sibling. We distinguish two types of violations.

Sibling violation: If x has a left sibling y and x's rank is k, then y has rank $k+1$ and its new right sibling after removing x has rank $k-1$ or, if $k=0$, does not exist.

We fix this violation depending on whether y is thin or thick. If y is thick, its leftmost child z has rank k. We remove z from y's child list and make it y's right sibling. This makes y thin but restores the rank conditions. So the operation terminates.

If y is thin, we decrease y's rank by 1, thereby making it thick and moving the violation one position to the left (between y, which has rank k now, and its left sibling, which has rank $k+2$, or between y and its parent). We update y to point to y's left sibling if it exists or to y's parent otherwise and repeat the process.

Parent violation: If x is the leftmost child of y y is thin, then $\text{rank}(x) = k$ and $\text{rank}(y) = k+2$. After removing x, y has only children of rank $k-1, k-2, \ldots, 0$, so the rank condition is violated again. In this case, we remove y from its parent's child list, insert it into the root list between min and succ, and update succ. We also decrease y's rank by 2, thereby making it thick again. Removing y may create a rank violation at y's left sibling or parent, so we update y so it points to this sibling or parent and repeat this process.

The fun part – analysis

To analyze the cost of a sequence of operations on a thin heap, we need to introduce the concept of amortized analysis. The idea is to assign an amortized cost to each operation that has no direct relation to the actual cost of the operation but satisfies the following crucial condition:

For any sequence of operations $\langle o_1, o_2, \ldots, o_m \rangle$ with actual costs $c_1, c_2, \ldots, c_m$ and amortized costs $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_m$, we have

$$\sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} \hat{c}_i$$

This allows us to bound the actual cost (!) of any sequence of operations by summing the amortized costs of the individual operations.

For the thin heap, we prove that the amortized cost of each operation is in $O(1)$, except that DeleteMin operations cost $O(\lg n)$ time amortized. This proves the desired result that a sequence of m operations n of which are DeleteMin operations takes $O(m + n \lg n)$ time.

There are many ways to define amortized costs with the above property (see the textbook chapter on amortized analysis). Here we introduce one of these tools: a potential function.

A potential function captures the structure of the data structure. As we perform operations on the data structure, its structure and hence its potential changes. We define the amortized cost of an operation as its actual cost plus the change in potential. So, let Φ_0 be the initial potential of the data structure, and let Φ_i be the potential after i operations $o_1, o_2, \ldots, o_i$. Then $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$.

This gives

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{m} c_i + \Phi_m - \Phi_0.$$

If $\Phi_m \geq 0$ and $\Phi_0 = 0$, we thus obtain the desired property that $\sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$.

For the thin heap, we define the potential as the number of roots plus twice the number of thin nodes. This clearly satisfies the condition that $\Phi_0 = 0$ and $\Phi_m \geq 0$ for every operation sequence, so $\sum \hat{c}_i \geq \sum c_i$ for every operation sequence $\langle o_1, o_2, \ldots, o_m \rangle$. Now consider the individual operations.

IsEmpty(Q) does not change the potential and has $O(1)$ cost. So its amortized cost is $O(1)$.

Insert(Q) has cost $O(1)$ and increases the potential by 1 because it creates a new root, so its amortized cost is $O(1)$.

DeleteMin(Q): Assume the minimum root has rank k and there are r roots. Then the cost of this operation is $O(k + r + \log n)$. The operation does not create any thin nodes and ensures we have less than $2 \log n$ roots left at the end because every root in the final root list has a unique rank and the maximum rank is less than $2 \log n$. Thus, the change in potential is at most $2 \log n - r$. Since $k \leq 2 \log n$, the real cost of the operation is $O(\log n + r)$. If $r \leq 2 \log n$, this is in $O(\log n)$.

If $r > 2\lg n$, the potential decrease by $r - 2\lg n$ pays for $O(r - 2\lg n)$ of the actual cost, leaving $O(\lg n + 2\lg n) = O(\lg n)$ as the amortized cost.

DecreaseKey: Making x a root takes constant time and increases the potential by at most 3, so the amortized cost of this part of the operation is $O(1)$. Now consider the cost of restoring violations.

Sibling violation: If y is thick, it takes constant time to make its leftmost child y's right sibling. Since this restores the rank condition, this happens only once and thus adds $O(1)$ to the cost of the operation.

If y is thin, decreasing its rank takes constant time and makes y thick. Thus, the potential drops by 2 and the amortized cost of this step is 0!

Parent violation: Removing y from its parent's child list and making it a root takes constant time. If y has a left sibling, this eliminates a thin node (y) and adds a new root (y), so the potential drops by 1. The amortized cost is 0.

If y has no left sibling and parent(y) is thick, the potential increases by 1 (y becomes thick, parent(y) becomes thin, and y becomes a new root). So the amortized cost is $O(1)$, but the rank condition is now restored, so this happens only once.

Finally, if parent(y) is thin, the potential decreases by 1 (y becomes thick, parent(x) was already thin, and y becomes a new root), so the amortized cost is 0.

Delete: The amortized cost is $O(\lg n)$ because DeleteMin has cost $O(\lg n)$ and DecreaseKey has cost $O(1)$.

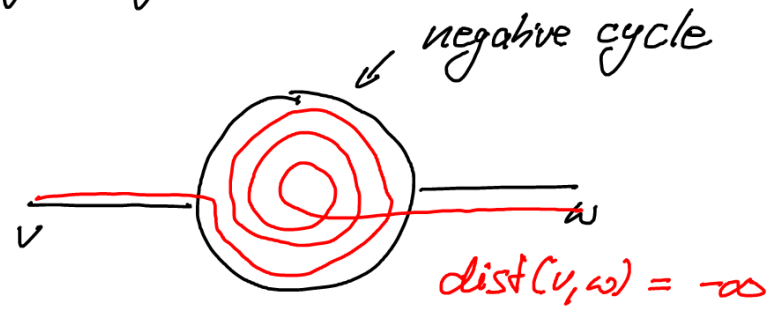
Theorem: A thin heap supports Insert, DecreaseKey, and IsEmpty operations in constant amortized time. Delete and DeleteMin take $O(\lg n)$ time amortized.

Exercise: Use a potential function to prove that a sequence of m operations on the union-find structure we used in Kruskal's algorithm is $O(m + n \lg n)$. (Hint: Define the potential of a node v to be $\varphi_v := \lceil \lg \ell_v \rceil$, where ℓ_v is the length of the list that contains v. The potential of the data structure is $\Phi = \sum_v (\lceil \lg n \rceil - \varphi_v)$. You should verify that creating the data structure has amortized cost $O(n \lg n)$ and the amortized cost per operation is $O(1)$ using this potential function.)

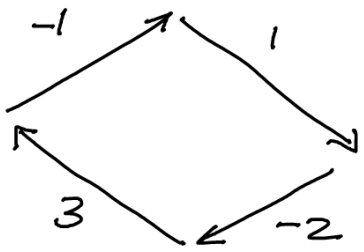
Dijkstra's algorithm

Given a graph $G=(V,E)$ with weights on its edges, a shortest path from a vertex v to a vertex w is a path from v to w with minimum total weight. The total edge weight of this path is the distance from v to w.

In general, the distance is well-defined only if G contains no negative cycle (cycle of negative total edge weight):



A directed graph may contain negative-weight edges and still not contain a negative-weight cycle:



For such graphs, shortest paths are well-defined but computing them is harder than when all edges have non-negative weights. We focus on non-negative weights here and discuss an algorithm for arbitrary edge weights when we cover dynamic programming.

The single source shortest paths (SSSP) problem is to compute the distances from a source vertex s to all other vertices of G.

What's the right greedy strategy for solving this problem? Some structural lemmas help:

Lemma: For $v \neq s$, let P_v be a shortest path from s to v. There exists such a set of shortest paths $\{P_v \mid v \in V \setminus \{s\}\}$ such that the union of these paths is a tree T. We call T a shortest path tree with root s.

Proof: Arrange the vertices of G in an arbitrary order $v_1, v_2, \ldots, v_n$ and let i be the largest index such that $P_{v_1} \cup P_{v_2} \cup \ldots \cup P_{v_i}$ is a tree. Since P_{v_1} is a path, we have $i \geq 1$. If $i = n$, then $T = P_{v_1} \cup P_{v_2} \cup \ldots \cup P_{v_n}$ is a tree and the lemma holds. So assume $i < n$. Let $T' = P_{v_1} \cup P_{v_2} \cup \ldots \cup P_{v_i}$ and let w be the last vertex of $P_{v_{i+1}}$ that belongs to T'. w divides $P_{v_{i+1}}$ into two (possibly empty) subpaths P' and P'' from s to w and from w to v_{i+1}. Let P''' be the path from s to w in T' and let $P'_{v_{i+1}}$ be the concatenation of P''' and P''. Then $P_{v_1} \cup P_{v_2} \cup \ldots \cup P_{v_i} \cup P'_{v_{i+1}}$ is a tree because $P''' \subseteq T'$ and P'' has exactly one vertex in T'. We prove that $P'_{v_{i+1}}$ is a shortest path from s to v_{i+1}, so we can replace $P_{v_{i+1}}$ with $P'_{v_{i+1}}$ and thereby increase i by one.

The key is to observe that P''' is no longer than P', so P'_{i+1} is no longer than P_{i+1}. To see this, consider a vertex v_j, $1 \le j \le i$, such that P_{v_j} includes ω and then P''' (because T' is a tree). Since $\omega \in T'$ and $T' = P_{v_1} \cup P_{v_2} \cup \ldots \cup P_{v_i}$, such a vertex v_j exists. Let P'_{v_j} be the path from s to v_j that follows P' from s to ω and then P_{v_j} from ω to v_j. If P' were shorter than P''', then P'_{v_j} would be shorter than P_{v_j}, a contradiction because P_{v_j} is a shortest path from s to v_j. □

Lemma: For a spanning tree T of G, let $d_T(v)$ be the length of the path from s to v in T, and let $D(T) = \sum_{v \in G} d_T(v)$. Then T is a shortest path tree of G iff there is no spanning tree T' with $D(T') \le D(T)$.

Proof: Assume T is a shortest path tree but there exists a spanning tree T' with $D(T') < D(T)$. Then $d_{T'}(v) < d_T(v)$, for some $v \in G$. Since $d_{T'}(v)$ is the length of the path from s to v in $T' \le G$ and $d_T(v)$ is the length of the path from s to v in T, the path from s to v in T cannot be a shortest path from s to v in G, a contradiction.

Now assume T is not a shortest path tree, and let T' be a shortest path tree. Then $d_T(v) \ge d_{T'}(v)$ for all $v \in G$ and there exists at least one vertex v with $d_T(v) > d_{T'}(v)$. Thus,

$D(T) > D(T')$. □

Since we want to minimize $D(T)$, a natural strategy is to start with $T = (\{s\}, \emptyset)$ (and thus $D(T) = 0$) and add vertices to T one by one while minimizing the increase of $D(T)$. This gives us Dijkstra's algorithm, which maintains the vertices not in T in a priority queue, sorted by the cost (increase of $D(T)$) of attaching them to T:

```
Dijkstra(G, s):
Mark all vertices of G as unexplored
Mark s as explored
d(s) = 0
Q = an empty priority queue
for every out-neighbour w of s do
    Insert(Q, w, w((s, w)))
    parent(w) = s
    d(w) = w((s, w))
while Q ≠ ∅ do
    v = DeleteMin(Q)
    Mark v as explored
    for every out-neighbour w of v do
        if w is unexplored then
            if w ∉ Q then
                Insert(Q, w, d(v) + w((v, w)))
                parent(w) = v
                d(w) = d(v) + w((v, w))
            else if d(v) + w((v, w)) < d(w) then
                DecreaseKey(Q, w, d(v) + w((v, w)))
```

$parent(w) = v$
$d(w) = d(v) + w((v,w))$

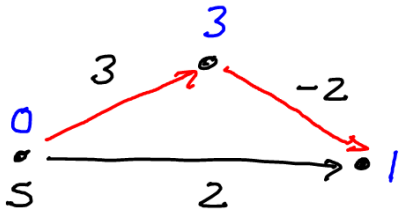
The running time of this algorithm is $O(n \log n + m)$ because it is the same as Prim's algorithm except that it uses different priorities. The next lemma shows that when the algorithm finishes $d(v) = dist_G(s,v)$ for all $v \in G$.

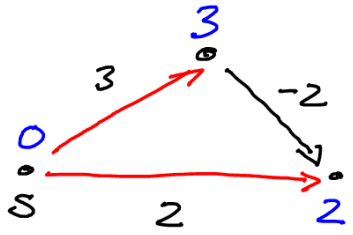
Lemma: When Dijkstra's algorithm finishes, $d(v)$ is the distance from s to v in G for all $v \in G$, provided all edges in G have non-negative weights.

Proof: Once a vertex is marked as explored, $d(v)$ does not change any more. Thus, $d(s) = 0$ at the end of the algorithm, which is the distance from s to s in G. Now assume there exists a vertex $v \neq s$ such that $d(v) \neq dist_G(s,v)$ when the algorithm finishes. Choose v such that every vertex u explored before v satisfies $d(u) = dist_G(s,u)$ at the end of the algorithm (and hence when u is marked as explored). Since Dijkstra's algorithm is a variant of graph traversal, it visits every vertex of G, including v. For v to be visited, it needs to be removed from Q, which in turn requires v to be inserted into Q. Let u be the last vertex that performs an $Insert(Q, v, d(v))$ or $DecreaseKey(Q, v, d(v))$ operation. Then u is visited before v, so $d(u) = dist_G(s,u)$ and, hence, $d(v) = d(u) + w((u,v)) = dist_G(s,u) + w((u,v)) \geq dist_G(s,v)$.

Now assume $d(v) > dist_G(s,v)$, let u be the vertex in P_v closest to v that is visited before v, and let w be u's successor in P_v. If $w = v$, then $d(v) \le d(u) + \omega((u,v)) = dist_G(s,u) + \omega((u,v)) = dist_G(s,v)$ when v is visited, a contradiction. If $w \neq v$, then $d(w) \le d(u) + \omega((u,w)) = dist_G(s,w) + \omega((u,w)) = dist_G(s,w) \le dist_G(s,v) < d(v)$ when v is visited. Since w is visited after v, this is a contradiction because v is not the minimum entry in Q when it is visited. Hence, $d(v) \le dist_G(s,v)$. □

Try to figure out where the previous proof uses that all edge weights are non-negative. Does Dijkstra's algorithm work also for negative edge weights? No. It is easy to verify that, in the following example, the tree on the left is a shortest path tree and the tree on the right is the one computed by Dijkstra's algorithm.





Minimum-length codes

Given a text T over an alphabet A, we would like to find a code for A that minimizes the number of bits needed to encode T. For example, consider the 4-letter alphabet $\{A, C, G, T\}$ encoded as

$$A = 00 \quad C = 01 \quad G = 10 \quad T = 11.$$

Then the string

AGACCATCACCAGAC

is encoded as

00100001010011010001010010000l

that is, using 30 bits. If we used the code

$$A = 0 \quad C = 10 \quad G = 110 \quad T = 111,$$

we would get

011001010011111001010011001O,

which requires only 27 bits. For longer texts over bigger alphabets, the savings can be much more dramatic, particularly if the character frequencies are very skewed as in most natural languages.

In an attempt to use even less space, could we have used the encoding

$$A = 0 \quad C = 1 \quad G = 01 \quad T = 11 \; ?$$

This would reduce the number of bits to 18. The problem is that, for example, 011 could encode AT, GC or ACC. Clearly we want a code where no two texts share the same encoding – we want to be able to decode the text. One type of codes that have this property is prefix codes:

For a code C of an alphabet A, let $C(a)$ denote the bit sequence used to encode the letter $a \in A$. C is a prefix code if there are no two characters $a, b \in A$ such that $C(a)$ is a prefix of $C(b)$. Any fixed-length code that satisfies $C(a) \neq C(b) \; \forall a, b \in A$ (e.g., ASCII) is a prefix code, as is the second code in our example that encodes the text in 27 bits. The third code is not a prefix code because, for example $C(A)$ is a prefix of $C(G)$.

Lemma: If C is a prefix code for an alphabet A, then $C(T) \neq C(T')$ for any two texts T, T' over A.

Proof: By induction on the length of $C(T)$. If $|C(T)| = 0$, then $T = \varepsilon$ (the empty string) because there cannot be any characters $a \in A$

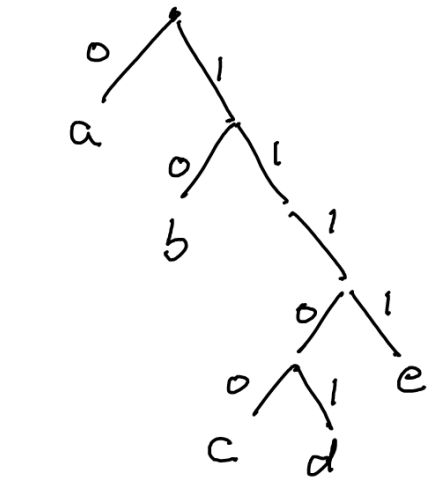
with $C(a) = \varepsilon$. (why?)

If $|C(T)| > 0$, we show that the first character of T is uniquely determined. Assume the contrary, that is, there exist two strings $T = a\ldots$ and $T' = b\ldots$ such that $C(T) = C(T')$. W.l.o.g. $|C(a)| \leq |C(b)|$. Since both $C(a)$ and $C(b)$ are prefixes of $C(T) = C(T')$, $C(a)$ must be a prefix of $C(b)$, a contradiction.

We have shown that, if $T = aT'$, then $T'' = aT'''$ for all T'' such that $C(T) = C(T'')$. Thus, $C(T) = C(a)C(T')$ and $C(T'') = C(a)C(T''')$. Since $C(T) = C(T'')$, we have $C(T') = C(T''')$. Since $|C(a)| > 0$, we have $|C(T')| < |C(T)|$. Thus, by the inductive hypothesis, $T' = T'''$ and $T = aT' = aT''' = T''$, that is, T is uniquely determined by $C(T)$. □

The proof highlights why prefix codes are attractive. To decode a text T encoded using a prefix code, we find the unique character $a \in A$ such that $C(a)$ is a prefix of $C(T)$. This is the first character of T and we decode the rest of T by removing $C(a)$ from $C(T)$ and iterating this procedure until we have consumed all bits in $C(T)$.

Now, there is a natural correspondence between prefix codes for A and binary trees with the elements of A at their leaves:



(Left edge = 0
Right edge = 1)

$C(a) = 0$
$C(b) = 10$
$C(c) = 11100$
$C(d) = 11101$
$C(e) = 1111$

We call a prefix code optimal for a text T if it minimizes $|C(T)|$.

Lemma: If C is an optimal prefix code for a text T, then every internal node in the tree corresponding to C has two children.

Proof: Let $f(a)$ be the number of times a occurs in T, and let $d_Y(a)$ be the depth of a in the tree Y corresponding to C. Then $|C(a)| = d_Y(a)$ and $|C(T)| = \sum_a |C(a)| f(a) = \sum_a d_Y(a) f(a)$.

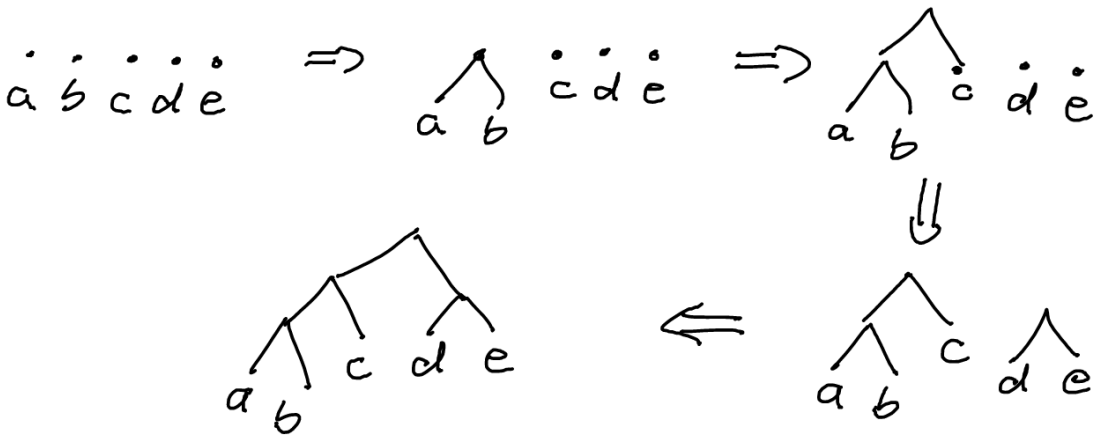
Now assume there exists a node v in Y with a single child w. Let Y' be the tree obtained by deleting v and making w a child of v's parent. Then $d_{Y'}(a) = d_Y(a) - 1$ for all $a \in A$ that are descendants of w and Y' represents a prefix code C' because Y' is binary and the elements of A label the leaves of Y'. This gives

$$|C'(T)| = \sum_a d_{T'}(a) f(a) < \sum_a d_T(a) f(a) = |C(T)|$$

because $d_{T'}(a) \leq d_T(a)$ $\forall a \in A$ and $d_{T'}(a) < d_T(a)$ for at least one $a \in A$. Thus, C is not optimal. □

So, our goal is to build a binary tree Y whose leaves represent A and such that $\sum_a d_Y(a) f(a)$ is minimized.

We can build any binary tree by starting with $|A|$ singleton trees and repeatedly picking two trees and making them children of a new root, thereby merging the two trees:



What determines the shape of the tree is how we choose the two trees to merge in each step. What's a natural greedy choice to minimize $\sum_a d_Y(a) f(a)$?

Whenever we merge two trees T_1 and T_2, we add one to $d_Y(a)$ for all $a \in T_1 \cup T_2$. Thus, if we want to minimize the resulting increase of $\sum_a d_Y(a) f(a)$, we should choose T_1 and T_2 such that $\sum_{a \in T_1 \cup T_2} f(a)$ is minimized.

This gives us Huffman's algorithm:

```
Huffman(T)
Count how often each character occurs in T.
Q = an empty priority queue
for all a ∈ A do
    Create a singleton tree a
    f(a) = the number of times a occurs in T
    Insert(Q, a, f(a))
while |Q| > 1 do
    a = DeleteMin(Q)
    b = DeleteMin(Q)
    Create a new node p
    Make a and b children of p
    f(p) = f(a) + f(b)
    Insert(Q, p, f(p))
r = DeleteMin(Q)
return r
```

The algorithm implements our greedy strategy and it does so in $O(n \lg n)$ time, where $n = |T|$ and $n = |A|$: To count character frequencies, we build a dictionary over the elements of A represented as a binary search tree. Each element stores its own frequency, initially 0. For each character in T, we look it up in the dictionary and increase its frequency. Now we create n leaves of Y and insert them into Q. Apart from these initial n insertions, we perform two DeleteMin operations per insertion. This gives $I = D$ (everything we delete must be inserted

first and we delete everything we insert) and $I = n + (D-1)/2$. Thus, $I = n + (I-1)/2$, $2I = 2n + I - 1$, $I + 1 = 2n$, $I = D = 2n - 1$. Thus, we perform less than $4n$ priority queue operations. So the counting part takes $O(m \lg n)$ time, building Y from the characters frequencies takes $O(n \lg n)$ time. Since $m \geq n$, the $O(m \lg n)$ part dominates. It remains to prove the algorithm's correctness.

Lemma: Huffman's algorithm constructs (a tree representing) a minimum-length prefix code for its input text T.

Proof: By induction on $n = |A| \geq 2$. (The case $n = 1$ is uninteresting because then all characters are the same and we need to transmit only m, which requires $\lceil \lg m \rceil$ bits.)

$n = 2$: Then Huffman's algorithm builds a tree with a root and two leaves. Thus, each character takes one bit to encode and we obviously cannot do better.

$n > 2$: Then the algorithm chooses two characters a and b with lowest frequencies, makes them children of a new node c with frequency $f(c) = f(a) + f(b)$. Note that the remaining steps of the algorithm do exactly the same as if we had replaced every occurrence of a and b in T with a new character c with frequency $f(c) = f(a) + f(b)$.

This modified text T' uses an alphabet of size $n-1$, so by the inductive hypothesis, the algorithm computes an optimal prefix code C' for T. Let Y' be the corresponding tree, and let Y be the tree computed for T. Then Y and Y' are the same, except that a and b are children of c in Y and c is a leaf in Y'. Thus, the code C represented by Y satisfies

$$C(x) = \begin{cases} C'(x) & x \notin \{a,b\} \\ C'(c)0 & x = a \\ C'(c)1 & x = b \end{cases}$$

Thus,
$$\begin{aligned} |C(T)| &= \sum_{x \in A} |C(x)| f(x) \\ &= \sum_{\substack{x \in A \\ x \notin \{a,b\}}} |C(x)| f(x) + |C(a)| f(a) + |C(b)| f(b) \\ &= \sum_{\substack{x \in A \\ x \notin \{a,b\}}} |C'(x)| f(x) + |C'(c)+1| f(a) + |C'(c)+1| f(b) \\ &= \sum_{\substack{x \in A \\ x \notin \{a,b\}}} |C'(x)| f(x) + |C'(c)| f(c) + f(a) + f(b) \\ &= |C'(T')| + f(a) + f(b) \end{aligned}$$

If this is not optimal, then there exists a code C'' such that $|C''(T)| < |C(T)|$. If $C''(a) = 50$ and $C''(b) = 51$, for some bit sequence 5, then we can define a code C''' for T' as

$$C'''(x) = \begin{cases} C(x) & x \neq c \\ \varepsilon & x = c \end{cases}$$

This code satisfies $|C''(T)| = |C'''(T')| + f(a) + f(b)$ as above. Thus, if $|C''(T)| < |C(T)|$, then $|C'''(T')| < |C'(T')|$, contradicting the optimality of C' for T'. So this cannot happen.

If there is no bit string ε such that $C''(a) = 0$ and $C''(b) = 1$, then we prove that there exists another code C'''' such that $C''''(a) = 0$, $C''''(b) = 1$ and $|C''''(T)| \leq |C''(T)| < |C(T)|$. By the above argument, this once again contradicts the optimality of C' for T'.

Since we assume C'' is an optimal prefix code for T, its corresponding tree Y'' has the property that all internal nodes have two children. Thus, the deepest leaf a' of Y'' must have a sibling b' and this sibling must also be a leaf. We choose a' and b' such that $f(a') \leq f(b')$, and a and b such that $f(a) \leq f(b)$. Since a and b have the minimum frequencies, this implies that $f(a) \leq f(a')$ and $f(b) \leq f(b')$. Now we build a new tree Y'''' representing C'''' by swapping a with a' and b with b' in Y''. Thus,

$$C''''(x) = \begin{cases} C''(x) & x \notin \{a, b, a', b'\} \\ C''(a) & x = a' \\ C''(a') & x = a \\ C''(b) & x = b' \\ C''(b') & x = b \end{cases}$$

Since a' and b' are deepest leaves, we have
$|C''(a)| = d_{y''}(a) \le d_{y''}(a') = |C''(a')|$ and
$|C''(b)| = d_{y''}(b) \le d_{y''}(b') = |C''(b')|$.

This gives

$$|C'''(T)| = \sum_{x \in A} |C'''(x)| f(x)$$

$$= \sum_{\substack{x \in A \\ x \notin \{a,b,a',b'\}}} |C''(x)| f(x) + |C''(a)| f(a') + |C''(a')| f(a) + |C''(b)| f(b') + |C''(b')| f(b)$$

and

$$|C''(T)| = \sum_{\substack{x \in A \\ x \notin \{a,b,a',b'\}}} |C''(x)| f(x) + |C''(a)| f(a) + |C''(a')| f(a') + |C''(b)| f(b) + |C''(b')| f(b')$$

Thus, $|C'''(T)| \le |C''(T)|$ if

$$|C''(a)| f(a') + |C''(a')| f(a) \le |C''(a)| f(a) + |C''(a')| f(a')$$

and

$$|C''(b)| f(b') + |C''(b')| f(b) \le |C''(b)| f(b) + |C''(b')| f(b')$$

However, $f(a) \le f(a')$ and $|C''(a)| \le |C''(a')|$.
This gives

$$(|C''(a)|f(a') + |C''(a')|f(a)) -$$
$$(|C''(a)|f(a) + |C''(a')|f(a')) =$$

$$\underbrace{(|C''(a)| - |C''(a')|)}_{\leq 0}\underbrace{(f(a') - f(a))}_{\geq 0} \leq 0$$

Thus,

$$|C''(a)|f(a') + |C''(a')|f(a) \leq |C''(a)|f(a) + |C''(a')|f(a')$$

as desired. The argument for b and b' is analogous. □