1 Introduction

A greedy algorithm is often considered the easiest of algorithms to describe and implement, and is characterized by the following two properties:

1. the algorithm works in successive stages, and during each stage a choice is made that is locally optimal
2. the sum totality of all the locally optimal choices produces a globally optimal solution

If a greedy algorithm does not always lead to a globally optimal solution, then we refer to it as a heuristic, or a greedy heuristic. Heuristics often provide a “short cut” (not necessarily optimal) solution.

The following are some computational problems that that can be solved using a greedy algorithm.

**Huffman Coding** finding a code for a set of items that minimizes the expected code-length

**Minimum Spanning Tree** finding a spanning tree for a graph whose weighted edges sum to a minimum value

**Single source distances in a graph** finding the distance from a source vertex in a weighted graph to every other vertex in the graph

**Fractional Knapsack** selecting a subset of items to load in a container in order to maximize profit

**Task Selection** finding a maximum set of timewise non-overlapping tasks (each with a fixed start and finish time) that can be completed by a single processor
Unit Task Scheduling with Deadlines finding a task-completion schedule for a single processor in order to maximize the total earned profit

Like all families of algorithms, greedy algorithms tend to follow a similar analysis pattern.

**Greedy Correctness** Correctness is usually proved through some form of induction. For example, assume their is an optimal solution that agrees with the first \( k \) choices of the algorithm. Then show that there is an optimal solution that agrees with the first \( k + 1 \) choices. Conclude that the greedy solution is in fact optimal.

**Greedy Complexity** The running time of a greedy algorithm is determined by the ease in maintaining an ordering of the candidate choices in each round. This is usually accomplished via a static or dynamic sorting of the candidate choices.

**Greedy Implementation** Greedy algorithms are usually implemented with the help of a static sorting algorithm, such as Quicksort, or with a dynamic sorting structure, such as a binary heap. Additional data structures may be needed to efficiently update the candidate choices during each round.
2 Data Structures for Greedy Algorithms

2.1 Binary Heaps

Definition 2.1. A perfect binary tree is a binary tree whose leaves all have the same depth and an incomplete binary tree is a perfect binary tree that is missing zero or more leaves (starting from the far right and moving left) at the last level.

Example of a perfect tree.

Example of an incomplete tree.
Definition 2.2. A binary min heap $H$ may be defined as an incomplete binary tree, where each
tree node $n$ stores a member $s_n$ of an ordered set $S$, and has the property that if $n'$ is a child of $n$,
then $s_n \leq s_{n'}$. For $s_1, s_2 \in S$, if

1. $s_1 < s_2$, then $s_1$ has **higher priority** than $s_2$.
2. $s_1 > s_2$, then $s_1$ has **lower priority** than $s_2$.
3. $s_1 = s_2$, then $s_1$ and $s_2$ **equal priority**.

![Example of a binary min heap whose nodes store integers.](image-url)
Note that a heap may be implemented as an array, where

1. the root is stored at index 1

2. if node $n$ is stored at index $i$, then its left (respectively, right) child is stored at index $2i$ (respectively, $2i + 1$).

**Proposition 2.3.** A binary heap $H$ of size $n$ (nodes) has height equal to $\lfloor \log n \rfloor$.

**Proof.** If $H$ is a perfect tree, then it has $n = 2^{h+1} - 1$ nodes, where $h$ is the tree height. Thus,

$$h = \lfloor \log(2^{h+1} - 1) \rfloor = \lfloor \log n \rfloor.$$

If $H$ is an incomplete tree with at least one missing node, then it has $n = 2^h + k$ nodes, where $0 \leq k < 2^h$, and

$$h = \lfloor \log(2^h + k) \rfloor = \lfloor \log n \rfloor,$$

since

$$2^h \leq 2^h + k < 2^h + 2^h = 2 \cdot 2^h = 2^{h+1}. \quad \square$$
2.2 Heap Operations

```c
void insert(Item i) insert an item into the heap.

- **Percolate up:** insert i at the end of the bottom level and, while its parent has a lower priority, swap i with its parent.
- O(log n) complexity
```

```c
Item pop() Remove from the heap item i stored at the root and return i.

- **Percolate down:** replace root node with the last node n in the bottom level and, while n’s highest-priority child c has higher priority than n, swap n with c.
- O(log n) complexity
```

```c
void increase_priority(Item i) Increase i’s priority.

- Increase i’s priority and then percolate it up the tree (see insert()) starting at its current location.
- O(log n) complexity
```

```c
void build_heap(Item array[N]) build a heap from an array of items. Complexity: Θ(n)
```
Example 2.4. The following is an example of inserting an item (7 in blue) into a binary min heap.

\[
\begin{array}{c}
2 \\
4 \\
5 \\
10 \\
8 \\
4 \\
7 \\
12 \\
8 \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
2 \\
4 \\
5 \\
10 \\
8 \\
4 \\
7 \\
12 \\
8 \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
2 \\
4 \\
5 \\
10 \\
8 \\
4 \\
7 \\
12 \\
8 \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
2 \\
4 \\
5 \\
10 \\
8 \\
4 \\
7 \\
12 \\
8 \\
9 \\
\end{array}
\]

insert(): item 7 added to end of heap.

insert(): item 7 swapped with parent 9.
insert(): item 7 swapped with parent 8.

insert(): final state.
Example 2.5. The following is an example of popping a binary min heap.

pop(): initial state.

pop(): root node 2 removed and replaced by end-of-heap node 9.
pop(): item 9 swapped with item 4.

pop(): item 9 swapped with item 4.
Example 2.6. The following is an example of increasing a node's priority.

pop(): final state.

increase_priority: initial state.
increase_priority: increase 5’s priority to 3.

increase_priority: swap 3 with parent 4.
increase_priority: swap 3 with parent 4.

increase_priority: final state.
2.3 Disjoint Sets Data Structure

The disjoint sets data structure, also known as the union-find data structure, is used for maintaining a collection of disjoint sets of some underlying set. In what follows we describe how each set in the collection can be represented by a tree, where each tree can be represented by a collection of nodes, with each being called a membership node (M-node).

```c
structure MNode {
    MNode* parent; //points to the parent of this MNode
    Item item; //the item being stored in this MNode
}
```
Example 2.7. Suppose the underlying set is the set of nonnegative integers, and consider the subset $S = \{1, 3, 5, 9, 11, 15, 26\}$. This set can be represented by the following tree.

![Tree representation of S]

Notice that the order of the numbers does not matter. All that matters is that

1. each member of $S$ is stored in one of the M-nodes, and
2. every item stored in an M-node is a member of $S$.

Notice also that the parent of 5 is 26, the parent of 26 is 15, etc.
2.4 Disjoint Set Operations

void union(MNode $n_1$, MNode $n_2$) has the effect of setting $n_2$’s parent to $n_1$.

- $O(1)$ complexity since it requires a single assignment.

MNode find(MNode $n$) Returns the root node $r$ of the tree in which $n$ is located.

- **Path Compression:** has the side effect of setting $n''$’s parent to $r$, for every $n'$ along the path from $n$ to $r$ (except for $r$).
- Complexity is proportional to the path length from $n$ to $r$.

**Theorem 2.8.** If one begins with $n$ disjoint singleton sets and performs a sequence of $m$ union and find operations (without necessarily using path compression), then the total running time is equal to $O(n + m \log n)$. Note: this result assumes that, when performing unions, the smaller tree is merged into the larger one.
Example 2.9. Suppose that we begin with the singleton sets

\{1\}, \{3\}, \{5\}, \{9\}, \{11\}, \{15\}, \{26\}.

verify that the merge operations, merge(1, 11), merge(26, 5), merge(9, 3), merge(15, 26), merge(1, 15), merge(9, 1) results in the tree shown in Example 2.7 (assuming the children of each node are unordered).
Example 2.10. The following is an example of path compression as a side effect to the operation \texttt{find(26)}.

The tree for which 26 is a member.

The resulting tree after \texttt{find(26)} has been executed.