Making Change

Fact: US coins have denominations: 1.00, 0.50, 0.25, 0.10, 0.05, and 0.01.

The Making Change Problem (MCP)

Given: an amount \( a \)
Find: the smallest collection of coins that is worth amount \( a \).

Example: For \( a = 0.28 \) the (unique) answer is 0.25, 0.01, 0.01, 0.01.

A “greedy” algorithm

```
// Input: a, an amount
// Output: a list of coins giving a solution to the MCP for a
C ← { 1.00, 0.50, 0.25, 0.10, 0.05, 0.01 }
anS ← the empty list
while a ≥ 0 do
    if (a < min(C)) then
        return the empty list // Indicates no Solution
    else
        c ← max({ c ∈ C : c ≤ a })
        add c to the ans list
        a ← a − c
    endWhile
return ans
```

PROBLEM: The above returns the wrong answer for \( a = 28 \).
PROBLEM: How do we know the previous version is correct?
Standard Features of Greedy Algorithms

Greedy algorithms generally (but not always) have the following features:

- The goal is to build an optimal solution to a problem by selecting items from a candidate set.
- A feasibility function is used to check whether a candidate can be used in the solution.
- A greedy choice function picks a “best” candidate to be added to the solution.
- An objective function assigns a value to partial solutions.
- A solution function tests whether we have a complete solution yet.

```python
C ← \{1.00, 0.50, 0.25, 0.14, 0.01\}; ans ← []
while \(a \geq 0\) do
  if \((a < \min(C))\) then return []
  else \(c \leftarrow \max(\{c \in C : c \leq a\})\)
  ans ← cons(c, ans); \(a \leftarrow a - c\)
endWhile
return ans
```

- \(C = \text{candidate set}\)
- feasibility \(\equiv \lfloor \text{coin value} \leq a \rfloor\)
- greedy choice \(\equiv c \leftarrow \max(\{c \in C : c \leq a\})\)
- objective function \(\equiv \text{the number of coins used (minimize)}\)
- solution function \(\equiv [a = 0]\)

Event Scheduling, 1

The Meeting Scheduling Problem (MSP)

**Given:** \((s_1, f_1), \ldots, (s_k, f_k)\) where, for each \(i\), \((s_i, f_i)\) represents a meeting starting at time \(s_i\) and finishing at time \(f_i\) with \(0 \leq s_i < f_i\) for each \(i\).

**Goal:** Schedule as many of non-overlapping meetings as possible.

**Definition**

Two meetings overlap when their intervals intersect, i.e., \((s_i, f_i) \cap (s_j, f_j) \neq \emptyset\).

A maximal conflict-free schedule for a set of classes.

Image from: http://www.cs.uiuc.edu/class/fa05/cs473g/lectures/06-greedy.pdf

Standard Features of Greedy Algorithms

Greedy algorithms generally (but not always) have the following features:

- The goal is to build an optimal solution to a problem by selecting items from a candidate set.
- A feasibility function is used to check whether a candidate can be used in the solution.
- A greedy choice function picks a “best” candidate to be added to the solution.
- An objective function assigns a value to partial solutions.
- A solution function tests whether we have a complete solution yet.

Event Scheduling, 2

The Meeting Scheduling Problem (MSP)

**Given:** \((s_1, f_1), \ldots, (s_k, f_k)\) where, for each \(i\), \((s_i, f_i)\) represents a meeting starting at time \(s_i\) and finishing at time \(f_i\) with \(0 \leq s_i < f_i\) for each \(i\).

**Goal:** Schedule as many of non-overlapping meetings as possible.

**Definition**

Two meetings overlap when their intervals intersect, i.e., \((s_i, f_i) \cap (s_j, f_j) \neq \emptyset\).

- candidate set = \(\{(s_1, f_1), \ldots, (s_k, f_k)\}\).
- \(S = \text{a partial solution} \equiv S \text{ is a set of non-overlapping meetings}\)
- \((s_i, f_i)\) is feasible for \(S \iff (s_i, f_i)\) does not overlap with anything in \(S\).
- solution \(\equiv \text{no more meetings can be added to } S\)
- objective function \(\equiv \text{the size of } S\ (\text{to be maximized})\)
- greedy choice \(\equiv \text{shortest meeting?}, \text{earliest starting time?}, \text{latest finishing time?}, \text{fewest overlaps?}, \text{earliest finishing time!}\)
### Event Scheduling, 3

**Function**

```plaintext
function greedySch(L: a list of events)
    Sort L so that \( f_1 \leq \cdots \leq f_k \)
    \( S \leftarrow \emptyset; \ last \leftarrow 0 \)
    for \( i \leftarrow 2 \) to \( k \) do
        if \( \text{last} \leq s_i \) then
            \( S \leftarrow S \cup \{ (s_i, f_i) \}; \ last \leftarrow f_i \)
    return \( S \)
```

- The same classes sorted by finish times and the greedy schedule. Image from: http://www.cs.uiuc.edu/class/fa05/cs473g/lectures/06-greedy.pdf

### Event Scheduling, 4

**Strategy:** Turn \( S'' \) into \( S' \).

Suppose \( f_1' \leq \cdots \leq f_m' \) and \( f_1'' \leq \cdots \leq f_n'' \).

For \( \ell = 1, \ldots, m \), let \( S''_\ell = \{ (s'_1, f'_1), \ldots, (s'_\ell, f'_{\ell}) \} \).

We want to show each \( S''_\ell \) is feasible.

#### Base Case

**Claim 1:** \( f_1' \leq f_1'' \).

**Proof:** By greedySch’s choice, \( f_1' = \) the smallest finishing time. Hence \( f_1' \leq f_1'' \).

**Claim 2:** \( S''_\ell \) is feasible.

**Proof:** Since \( f_1' \leq f_1'' \) (Claim 1) and \( f_1'' \leq f_2'' \) (\( S'' \) has no overlaps), \( f_1' \leq f_2'' \). Hence, \( S''_\ell \) has no overlaps.

### Event Scheduling, 5

**Strategy:** Turn \( S'' \) into \( S' \).

Suppose \( f_1' \leq \cdots \leq f_m' \) and \( f_1'' \leq \cdots \leq f_n'' \).

For \( \ell = 1, \ldots, m \), let \( S''_\ell = \{ (s'_1, f'_1), \ldots, (s'_\ell, f'_{\ell}) \} \).

We want to show each \( S''_\ell \) is feasible.

#### Induction step Case

**IH:** \( S''_{\ell-1} \) is feasible, \( 1 \leq \ell \leq m \).

**Convention:** \( s''_{n+1} = \infty \).

**Claim 3:** \( f_1' \leq f_\ell'' \).

**Proof:** Since \( S''_{\ell-1} \) is consistent, \( s''_\ell \geq f_{\ell-1}' \). By greedySch’s greedy choice, \( f_\ell' \) is the smallest finishing time with a corresponding starting time \( \geq f_{\ell-1}' \). Hence, \( f_\ell' \leq f_\ell'' \).

**Claim 4:** \( S''_\ell \) is feasible.

**Proof:** Since \( f_\ell' \leq f_\ell'' \) (Claim 3) & \( f_\ell'' \leq f_{\ell+1}'' \) \( (S'' \) has no overlaps), \( f_\ell' \leq f_{\ell+1}'' \). Hence, \( S''_{\ell+1} \) has no overlaps.

**Goal:** Show \( m = n \).

### Event Scheduling, 6

**Strategy:** Turn \( S'' \) into \( S' \).

Suppose \( f_1' \leq \cdots \leq f_m' \) and \( f_1'' \leq \cdots \leq f_n'' \).

For \( \ell = 1, \ldots, m \), let \( S''_\ell = \{ (s'_1, f'_1), \ldots, (s'_\ell, f'_{\ell}) \} \).

We proved: \( S''_m \) is feasible.

#### Final step

Suppose by way of contradiction that \( m < n \).

Since \( S''_m \) is feasible, \( (s''_{n+1}, f''_{n+1}) \) does not overlap with any of \( (s'_1, f'_1), \ldots, (s'_m, f'_m) \).

But then by greedySch’s greedy choice, the algorithm would have chosen at least one more event to add to \( S \), a contradiction.

Therefore, \( m = n \) and \( S' \) is optimal.
Knapsack, 1

The Knapsack Problem (KP)

Given:
- A knapsack with capacity $W$ kgs.
- Items $1, \ldots, n$
- Item $i$ has weight $w_i$ and value $v_i$.

Find: $S \subseteq \{1, \ldots, n\}$ so that
- $\sum_{i \in S} w_i \leq W$ and
- $\sum_{i \in S} v_i$ is maximized.

Greedy Heuristic

Order items so that $\frac{v_i}{w_i} \geq \ldots \geq \frac{v_n}{w_n}$

for $i = 1$ to $n$

if $w_i \leq \text{cap}$ then $S \leftarrow S \cup \{i\}$

else return $S$

return $S$

Knapsack, 2

The Continuous Knapsack Problem (CKP)

Given:
- A knapsack and items (with weights and values) as before.

Find: $q_1, \ldots, q_n$ where $q_i \in [0, 1]$ is the fraction of item $i$ taken and
- $\sum_{i \in S} q_i \cdot w_i \leq W$ and
- $\sum_{i \in S} q_i \cdot v_i$ is maximized.

Proof: Suppose $[q_1, \ldots, q_n]$ is the result of $gk$.

$[p_1, \ldots, p_n]$ is an optimal solution.

If $q_i = p_i$ for $i = 1, \ldots, n$ we are done.

Otherwise, let $j = \min \{i : q_i \neq p_i\}$.

By the greedy choice, $q_j \geq p_j$.

Let $[p'_1, \ldots, p'_n]$ be such that

- $p'_i = p_i$ for $i < j$ and $p'_j = q_j$

- $[p'_1, \ldots, p'_n]$ is the result of subtracting $(p_i - q_i) \cdot w_i$, much weight from items $i + 1, \ldots, n$.

Claim 1: $[p'_1, \ldots, p'_n]$ is an optimal solution.

Lemma

The above solves the CKP.

Knapsack, 3

The Continuous Knapsack Problem (CKP)

Given: A knapsack and items as before.

Find: $q_1, \ldots, q_n \in [0, 1]$ such that
- $\sum_{i \in S} q_i \cdot w_i \leq W$ and
- $\sum_{i \in S} q_i \cdot v_i$ is maximized.

function $gk(\ldots)$

Order items so that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$

for $i = 1$ to $n$

if $w_i \leq \text{cap}$ then $q_i = 1$; $\text{cap} \leftarrow \text{cap} - w_i$

else if $\text{cap} > 0$ then $q_i = \frac{w_i}{\text{cap}}$; $\text{cap} \leftarrow 0$

else $q_i = 0$

return $[q_1, \ldots, q_n]$

Value = 13.50

Knapsack, 4

The Continuous Knapsack Problem (CKP)

Given: A knapsack and items as before.

Find: $q_1, \ldots, q_n \in [0, 1]$ such that
- $\sum_{i \in S} q_i \cdot w_i \leq W$ and
- $\sum_{i \in S} q_i \cdot v_i$ is maximized.

function $gk(\ldots)$

Order items so that $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$

for $i = 1$ to $n$

if $w_i \leq \text{cap}$ then $q_i = 1$; $\text{cap} \leftarrow \text{cap} - w_i$

else if $\text{cap} > 0$ then $q_i = \frac{w_i}{\text{cap}}$; $\text{cap} \leftarrow 0$

else $q_i = 0$

return $[q_1, \ldots, q_n]$

Proof: Suppose $[q_1, \ldots, q_n]$ is the result of $gk$.

$[p_1, \ldots, p_n]$ is an optimal solution.

If $q_i = p_i$ for $i = 1, \ldots, n$ we are done.

Otherwise, let $j = \min \{i : q_i \neq p_i\}$.

By the greedy choice, $q_j \geq p_j$.

Let $[p'_1, \ldots, p'_n]$ be such that

- $p'_i = p_i$ for $i < j$ and $p'_j = q_j$

- $[p'_1, \ldots, p'_n]$ is the result of subtracting $(p_i - q_i) \cdot w_i$, much weight from items $i + 1, \ldots, n$.

Claim 1: $[p'_1, \ldots, p'_n]$ is an optimal solution.

Claim 2: By repeating the above trick as many times as needed, $[p'_1, \ldots, p'_n]$ can be transformed into $[q_1, \ldots, q_n]$ maintaining optimality along the way.

Lemma

The above solves the CKP.
Minimal Spanning Trees, 1

Minimal Spanning Tree Problem (MSTP)

Given: A connected undirected graph \((V, E)\) & \(\text{len}: E \rightarrow \mathbb{R}^+\).
Find: A minimal cost subgraph that connects all the vertices.

Property 1
Deleting an edge from a cycle cannot disconnect a graph.

Corollary
(a) So the subgraph above must be a tree.
(b) Deleting an edge from a tree disconnects it (into two trees).

Image from: http://en.wikipedia.org/wiki/Minimum_spanning_tree

Digression on Trees, 1

Definition
A tree is a connected, acyclic, undirected graph.

Property 2
A tree on \(n\) vertices has \(n - 1\) edges.

Proof (continued).
Induction step. \(n > 1\).
Suppose \(T\) is a tree with \(#v(T) = n\).
IH: Prop. 2 holds all trees \(T'\) with \(#v(T') < n\).
Pick an edge \((u, v)\) in \(T\) and delete it.
This disconnects \(T\) into trees \(T_1\) and \(T_2\) (by Cor. (b)).
Note that:
\[#v(T_1), #v(T_2) < n\] \& \(#v(T_1) + #v(T_2) = n\).
So by the IH:
\[#e(T_1) = #v(T_1) - 1\] \& \[#e(T_2) = #v(T_2) - 1\].
∴ Adding \((u, v)\) back in results in
\[#e(T) = 1 + #e(T_1) + #e(T_2)\]
\[= 1 + (#v(T_1) - 1) + (#v(T_2) - 1)\]
\[= #v(T) - 1\]
\[= n - 1\].

Digression on Trees, 2

Definition
A tree is a connected, acyclic, undirected graph.

Property 3
Suppose \(G = (V, E)\) is a connected undirected graph with \(|E| = |V| - 1\).
Then \(G\) is a tree.

Proof.
Since \(G\) is connected, it has a spanning subtree \(T\).
\(#v(T) = |V|\) and \(#e(T) = #v(T) - 1\) (since \(T\) is a tree).
So \(#e(T) = |E|\).
∴ \(T = G\).

Digression on Trees, 3

Definition
A tree is a connected, acyclic, undirected graph.

Property 4
Suppose \(G\) is an undirected graph.
\(G\) is a tree \iff there is a unique path between any two vertices in \(G\).

Proof.
\((\Rightarrow)\) By Cor. (b), in an acyclic graph, there is at most one path between any two verts.
So in a tree (= acyclic + connected), there must be exactly one path between any two vertices.
\((\Leftarrow)\) Suppose there is a unique path between any two vertices in \(G\).
So \(G\) is connected . . . and \(G\) must be acyclic (Cor. (b) again).
How to make the Greedy Choice for MST, 1

Minimal Spanning Tree Problem (MSTP)

**Given:** A connected undirected graph $G = (V, E)$ & $\text{len} : E \rightarrow \mathbb{R}^+$.  
**Find:** A minimal cost subgraph that connects all the vertices.

**Cut Property**

Suppose
- $X \subseteq E$ is a part of a MST of G.
- $S \subseteq V$ is such that no $e \in E$ crosses the $S/(V - S)$ boundary.
- $e$ is a lightest crossing this boundary.

Then $X \cup \{e\}$ is part of some MST.

![Image from DPV](image.png)

How to make the Greedy Choice for MST, 2

**Cut Property**

Suppose
- $X \subseteq E$ is a part of a MST of G.
- $S \subseteq V$ is such that no $e \in E$ crosses the $S/(V - S)$ boundary.
- $e$ is a lightest crossing this boundary.

Then $X \cup \{e\}$ is part of some MST.

**Proof.**

Suppose $X \subset T$ is some MST for G. If $e$ is part of $T$ we are done.

So suppose $e$ is not part of $T$.

Then $T \cup \{e\}$ has a cycle. (Why?)

There is a $T$-edge $e'$ that crosses the $S/(V - S)$ boundary. (Why?)

Let $T' = T \cup \{e\} - \{e'\}$.

Claim: $T'$ is a tree.

Claim: $T'$ is a MST for G.

Claim: $T'$ is a tree.

Claim: $T'$ is a MST for G.

Output:

- $(X, \text{cost}(X))$ of $V$ by $\text{virg}$
- $\text{cost}(X)$ is a minimum across all $X \subseteq V$.

**MST Scheme**

$X \leftarrow \emptyset$  // edges picked so far  

while $(|X| < |V| - 1)$ do

Pick $S \subseteq V$  // $X$ has no edges between $S$ and $(V - S)$

e $\leftarrow$ a min-weight edge between $S$ and $(V - S)$

$X \leftarrow X \cup \{e\}$

return $X$

Now all we have to do is make this precise.

**Prim’s Algorithm: Growing a MST**

function Prim($G, \text{wght}$)

// **Input:** $G = (V, \text{E})$, a connected graph
// **Output:** $\text{undir. graph & wght : E} \rightarrow \mathbb{R}^+$
// **Output:** A MST given by $\text{prev}$

for each $u \in V$ do

$\text{cost}[u] \leftarrow \infty$; $\text{prev}[u] \leftarrow \text{null}$

$u_0 \leftarrow$ some element of $V$

$\text{cost}[u_0] \leftarrow 0$

// Make a priority queue using

// **cost-vals as keys**

$H \leftarrow \text{makequeue}(V, \text{cost})$

while $(H$ is not empty) do

$v \leftarrow \text{deleteMin}(H)$;

for each $z$ adjacent to $v$ do

if $\text{cost}[z] > \text{wght}(v, z)$

$\text{cost}[z] \leftarrow \text{wght}(v, z)$

$\text{prev}[z] \leftarrow v$

$\text{decreaseKey}(H, z, \text{cost}[z])$

return $\text{prev}$

Set $S$ | $A$ | $B$ | $C$ | $D$ | $E$ | $F$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\text{cost}/\text{mill}$</td>
<td>$\text{cost}/\text{mill}$</td>
<td>$\text{cost}/\text{mill}$</td>
<td>$\text{cost}/\text{mill}$</td>
<td>$\text{cost}/\text{mill}$</td>
</tr>
<tr>
<td>$A, D, B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A, D, B, C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A, D, B, C, F$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Images from DPV
Prim’s Algorithm: Growing a MST

### Disjoint Sets Data Structure

- **makeset(x)** make a single set \{x\}
- **find(x)** return the label of x’s set
- **union(x, y)** merge x’s and y’s sets

**Ideas:**
- **vertices** = cities
- **sets** = countries
- **labels** = capital cities

union(x, y) = x’s and y’s countries unify (and pick a new capital)

par[x] = x, if x is a capital city
par[x] = y (≠ x), if x is not a capital city where y is x’s regional capital

Representation of the sets \{B, E\} and \{A, C, D, F, G, H\}

### Kruskal’s Algorithm

**Disjoint Sets Data Structure**

- **makeset(x)** make a single set \{x\}
- **find(x)** return the label of x’s set
- **union(x, y)** merge x’s and y’s sets

**Cut Property**

Suppose:
- \( X \subseteq E \) is a part of a MST of G.
- \( S \subseteq V \) is such that no \( e \in E \) crosses the \( S/(V - S) \) boundary.
- \( e \) is a lightest crossing this boundary.

Then \( X \cup \{e\} \) is part of some MST.

**Claim 1:** In K’s Algorithm, the disjoint sets are the connected components of X.

**Claim 2:** In K’s Algorithm, when \( \text{find}(u) \neq \text{find}(v) \), \( (u, v) \) is always the lightest edge jointing u’s set to v’s set.
Union/Find, 3

Property 1

For any \( x \), \( \text{rank}[x] < \text{rank}[\text{par}[x]] \).

Proof: By an easy induction.

Property 2

For any \( x \), \( x \) has at least \( 2^{\text{rank}[x]} \) nodes in its tree.

Proof: By another easy induction.

Property 3

Suppose there are \( n \) elements total. Then there can be at most \( n/2^k \) nodes of rank \( k \).

Proof: By Property 2.

Corollary: The maximum rank is \( \log_2 n \).

Corollary: \text{find} and \text{union} are \( O(\log n) \) time.

Union/Find, 4: A Wonderful Trick

function \text{find}(x) \ // \text{find} with path compression
if \( x \neq \text{par}[x] \) then
\text{par}[x] \leftarrow \text{find}(\text{par}[x])
return \text{par}[x]

Back to Kruskal’s Algorithm

Disjoint Sets Data Structure

\text{makeset}(x) \ \text{make a single set} \ \{x\}
\text{find}(x) \ \text{return the label of } x \text{’s set}
\text{union}(x,y) \ \text{merge } x \text{’s set and } y \text{’s set}

Lemma

Suppose there are \( n \) elements total and there are a sequence of \( m \) many finds.

Then the total time for these is \( O(n \log^* n) \).

In fact: We can replace \( \log^* \) with the inverse of Ackermann’s function!

Examples:

\[
\log^*(1) = 0, \quad \log^*(2) = 1, \\
\log^*(4) = 2, \quad \log^*(16) = 3, \\
\log^*(2^{16}) = 4, \quad \log^*(2^{2^{16}}) = 5
\]

Runtime Analysis

- \(|V| \) many makesets: \( \Theta(|V|) \) time.
- Sorting \(|E| \): \( \Theta(|E| \log_2 |E|) \).
- The for-loop goes through at most \(|E| \)-many iterations.
- In the for-loop:
  - \( 2|E| \)-many finds
  - \( (|V| - 1) \)-many unions

Total:

\[
O(|V| + |E| \log |E| + (|E| + |V|) \log_s |V|) = O(|E| \log_2 |V|)
\]