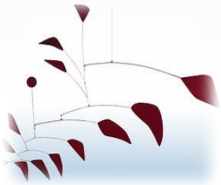
EECS 3101 - Design and Analysis of Algorithms



Shahin Kamali

Topic 5 - Greedy Algorithms

Overview

- Greedy Algorithms & Applications
- Activity-selection problem
- Huffman coding
- Fractional Knapsack

Greedy Algorithms Overview

- A **greedy algorithm** always makes the choice that looks best at the moment.
  - it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
  - Dynamic Programming applies when subproblems overlap, i.e., they share subsubproblems!

Greedy Algorithms Overview

- A **greedy algorithm** always makes the choice that looks best at the moment.
  - it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
  - Dynamic Programming applies when subproblems overlap, i.e., they share subsubproblems!
- Greedy algorithms do not always yield optimal solutions, but for many problems they do, and sometime lead to **approximation algorithms**.

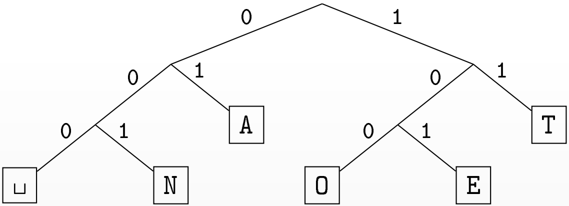
Greedy Algorithms vs Dynamic Programming

- Dynamic programming works by providing multiple **candidate** solutions for a problem (given by optimal solutions for the subproblem) and taking the best candidate.
- Greedy algorithms are special instances where only one candidate (given by the greedy choice) results in an optimal solution!
  - If this is the case for a problem, the greedy solution gives the optimal solution quicker!

Huffman Coding

- We want to create “codes" for different characters from a source.
  - More frequent characters, e.g., ‘A’ should get a smaller code than less frequent ones, e.g., ‘q’.
- Source alphabet is arbitrary (say Σ), coded alphabet is $\{0, 1\}$
- We build a binary tree to store the decoding dictionary D
- Each character of Σ is a leaf of the trie

Example: $\Sigma = \{\texttt{AENOT}\sqcup\}$



Encoding with Frequencies

- Consider a text with 100k characters over alphabet $\Sigma = \{a, b, c, d, e, f\}$. We want to store it in binary, using a fixed **codeword** for each character.
- We scan and see the following frequencies for characters.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

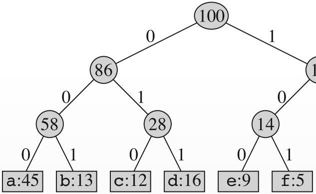
- **Questions:** how should we define codewords for characters to minimize the total length of the codes for (all characters of) the text?

Encoding with Frequencies

- **Option 1**: Assign a **fixed-length code** to each character. The fixed length of the codes will be $\lceil \log |\Sigma| \rceil = \lceil \log 6 \rceil = 3$.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

- This code requires $3 \times 100k = 300k$ bits to code the entire file. Can we do better?

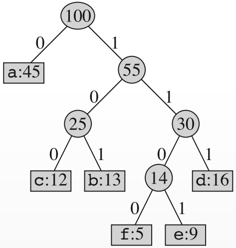


Encoding with Frequencies

- **Option 2**: Assign a **variable-length code** to each character by giving frequent characters short codewords and infrequent characters long codewords.

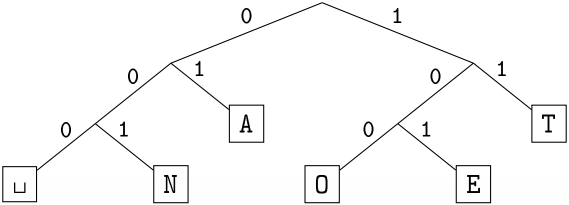
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224k$ bits.



Prefix-Free Encoding/Decoding

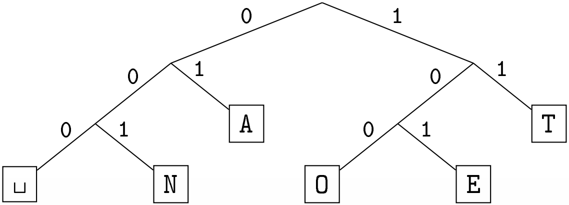
- Binary trees that represent codes are **prefix-free** in the sense that the code for a character c is not the prefix of a code for a character c'.
  - There is always an optimal encoding which is prefix-free.
  - Prefix-free codes are easy to decode!



- Encode `AN␣ANT`
- Decode `111000001010111`

Prefix-Free Encoding/Decoding

- Binary trees that represent codes are **prefix-free** in the sense that the code for a character c is not the prefix of a code for a character c'.
  - There is always an optimal encoding which is prefix-free.
  - Prefix-free codes are easy to decode!



- Encode `AN␣ANT` $\rightarrow$ `010010000100111`
- Decode `111000001010111` $\rightarrow$ `TO␣EAT`

Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$.
     Assign a "frequency" to each tree: sum of frequencies of all letters in tree (initially, these are just the character frequencies.)
  3. Merge two trees with the least frequencies, new frequency is their sum
     (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.
- What data structure should we store the trees in to make this efficient?

Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$.
     Assign a "frequency" to each tree: sum of frequencies of all letters in tree (initially, these are just the character frequencies.)
  3. Merge two trees with the least frequencies, new frequency is their sum
     (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.
- What data structure should we store the trees in to make this efficient?
  A min-ordered heap! Step 3 is two *delete-min*s and one *insert*

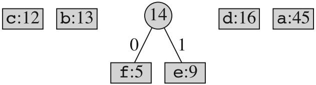
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

f:5 e:9 c:12 b:13 d:16 a:45

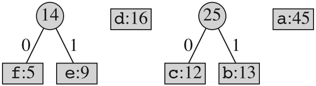
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



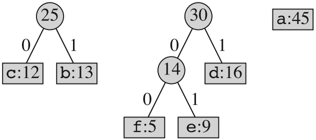
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



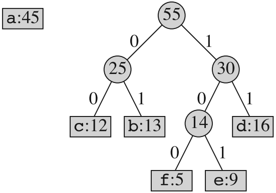
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



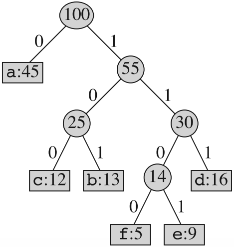
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



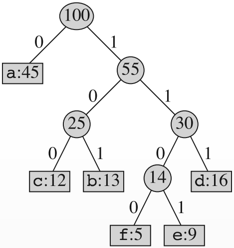
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



Building tree example

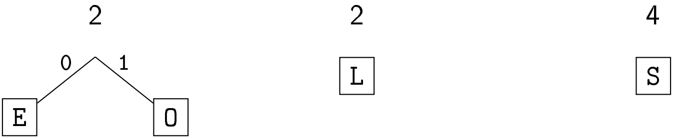
Example text: `LOSSLESS`

Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4

Building tree example

Example text: LOSSLESS

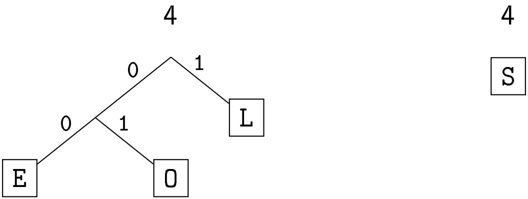
Character frequencies: E : 1,   L : 2,   O : 1,   S : 4



Building tree example

Example text: LOSSLESS

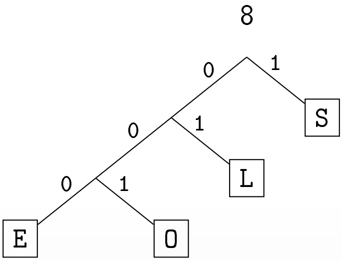
Character frequencies: E : 1, L : 2, O : 1, S : 4



Building tree example

Example text: LOSSLESS

Character frequencies: E : 1, L : 2, O : 1, S : 4

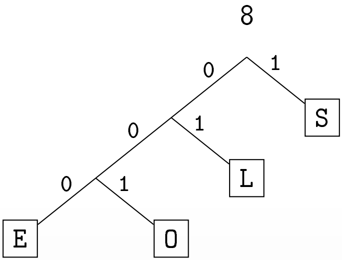


LOSSLESS $\rightarrow$

Building tree example

Example text: LOSSLESS

Character frequencies: E : 1, L : 2, O : 1, S : 4



LOSSLESS → 01 001 1 1 01 000 1 1

Building tree example

- It is possible to create alternative trees by merging trees other than the two with minimum frequencies.
  - Such trees, however, do not always the optimal solutions (shortest codes)!

Example text: `LOSSLESS`

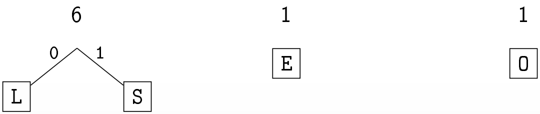
Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4

Building tree example

- It is possible to create alternative trees by merging trees other than the two with minimum frequencies.
    - Such trees, however, do not always the optimal solutions (shortest codes)!

Example text: `LOSSLESS`

Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4

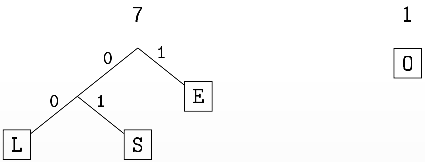


Building tree example

- It is possible to create alternative trees by merging trees other than the two with minimum frequencies.
  - Such trees, however, do not always the optimal solutions (shortest codes)!

Example text: `LOSSLESS`

Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4

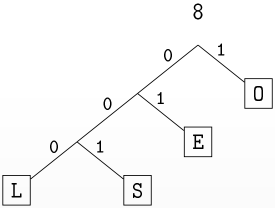


Building tree example

- It is possible to create alternative trees by merging trees other than the two with minimum frequencies.
  - Such trees, however, do not always the optimal solutions (shortest codes)!

Example text: LOSSLESS

Character frequencies: E : 1, L : 2, O : 1, S : 4



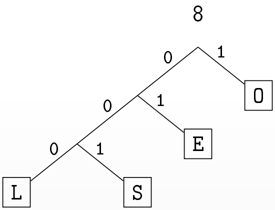
LOSSLESS →

Building tree example

- It is possible to create alternative trees by merging trees other than the two with minimum frequencies.
  - Such trees, however, do not always the optimal solutions (shortest codes)!

Example text: `LOSSLESS`

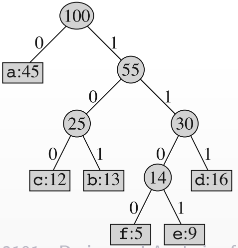
Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4



`LOSSLESS` → `000 1 001 001 000 01 001 001`

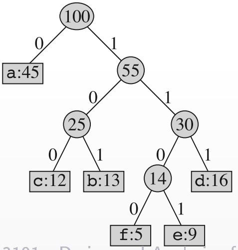
Huffman Tree is Greedy

- The cost of a tree is the total length of the code for it.
- Whenever an algorithm merges two trees, its cost is increased by frequency of the merged tree: the codelength for all characters in the leaves of the two trees increase by 1.
  - Here the cost is $(5+9)+(12+13)+(14+16)+(25+30)+(45+55)=224k$
- This code requires $(45\cdot 1+13\cdot 3+12\cdot 3+16\cdot 3+9\cdot 4+5\cdot 4)\cdot 1,000=224k$ bits.



Huffman Tree is Greedy

- The cost of a tree is the total length of the code for it.
- Whenever an algorithm merges two trees, its cost is increased by frequency of the merged tree: the codelength for all characters in the leaves of the two trees increase by 1.
  - Here the cost is $(5+9)+(12+13)+(14+16)+(25+30)+(45+55) = 224k$
- Huffman tree is **greedy** in the sense that it selects the merger with minimum cost!
- This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224k$ bits.



Greedy Algorithms Framework

- A **greedy** algorithm solves a problem sequentially, by making **greedy choices** that seems best at the moment.
- To ensure optimality by a greedy algorithm, a problem must have two properties:
  - **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to subproblems.
    - The optimal solution can be described recursively as a function of optimal solutions for subproblems.
    - This property is necessary for dynamic programming solutions as well.
  - **Greedy-choice property:** we can assemble a globally optimal solution by making locally optimal (greedy) choices.
    - We make the choice that looks best in the current problem, without considering results from subproblems.

Greedy vs Dynamic Programming

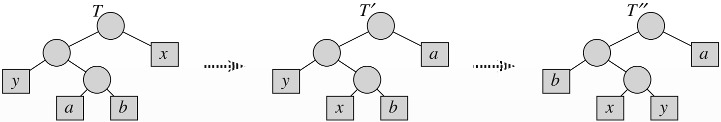
- In both paradigms, the optimal substructure property is present.
- In Dynamic Programming, there are often a few choices (candidates) at each step, and making the optimal choice requires looking at the outcome (value) of the optimal solution for the subproblems.
- In problems with Greedy-choice property, one candidate, selected by the greedy choice, results in an optimal solution, regardless of the value of the subproblems!

Huffman Code Revisit

- At each step, we must select two trees to merge:
  - **Optimal substrcuture property:** Given k trees, we have $\binom{n}{2}$ ways to choose two to merge.
    - If we choose two trees t_1 and t_2 to merge, the cost of this choice is $freq(t_1) + freq(t_2)$ plus the cost of merging the updated set of trees (in which t_1 and t_2 are merged with one merged tree).
    - A Dynamic Programming solution: One can try all possible candidates and take the one with minimum cost among them.

Huffman Code Revisit

- At each step, we must select two trees to merge:
  - **Greedy-choice property:** It is best to choose the two trees with lowest frequencies to merge.
    - Let x and y be the two trees with lowest frequencies.
    - For the sake of contradiction, suppose there is an optimal tree T' where two other nodes a and b are merged before x and y.
    - Replacing x with a and y with b results in a better tree than T'' with a cost no more than T, where x and y are merged first!



Activity Selection Problem

- We have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n. Activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i$.
- Activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity Selection Problem

- We have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n. Activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i$.
- Activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- Activities a_i and a_j ($i < j$) are **compatible** if their intervals do not overlap, that is a_i ends before a_j finishes.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity Selection Problem

- We have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n. Activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i$.
- Activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- Activities a_i and a_j $(i < j)$ are **compatible** if their intervals do not overlap, that is a_i ends before a_j finishes.
- We want to select (accept) the largest set of mutually compatible activities.
  - $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. But $\{a_1, a_4, a_8, a_{11}\}$ is an even larger (thus better) such subset.

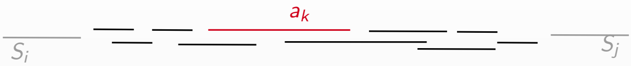
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Subproblem Optimality

- Trying a DP approach: Let S_{ij} the set of activities that start after a_i finishes and that finish before a_j starts.
- If the optimal solution for S_{ij} contains a_k, then it must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.

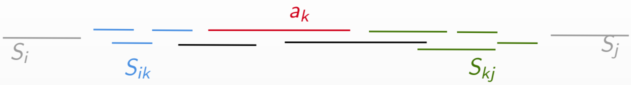
Subproblem Optimality

- Trying a DP approach: Let S_{ij} the set of activities that start after a_i finishes and that finish before a_j starts.
- If the optimal solution for S_{ij} contains a_k, then it must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.



Subproblem Optimality

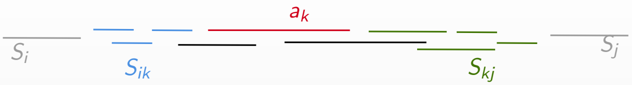
- Trying a DP approach: Let S_{ij} the set of activities that start after a_i finishes and that finish before a_j starts.
- If the optimal solution for S_{ij} contains a_k, then it must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.



Subproblem Optimality

- Trying a DP approach: Let S_{ij} the set of activities that start after a_i finishes and that finish before a_j starts.
- If the optimal solution for S_{ij} contains a_k, then it must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.
- If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we have:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset\ , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset\ . \end{cases}$$



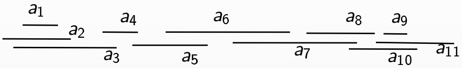
Greedy-Choice Property

- **Greedy choice:** Select activity a_k that ends earliest!
- There is an optimal solution that contains a_k:
  - For the sake of contradiction, suppose no optimal solution contains a_k, and let X be any optimal set of activities (X does not contain a_k).
    - If X does not contain any interval that intersect a_k, we can simply add a_k to X to get a better solution; this contradicts optimality of a_k.
    - If X contains an $a_{k'}$ which intersects a_k, replace a_k with $a_{k'}$ in X $\rightarrow$ since a_k ends earlier than a'_k all activities in $X - \{a_{k'}\}$ are compatible with a_k.

Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

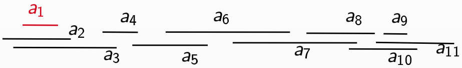
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

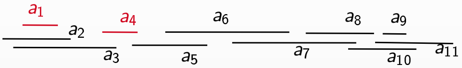
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

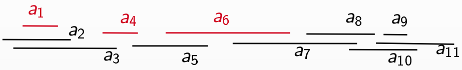
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

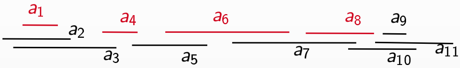
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

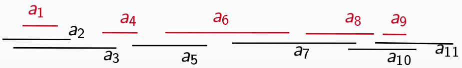
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Greedy Algorithm

- Since both subproblem optimality and greedy-choice property hold, we can devise a simple greedy algorithm which repeatedly applies the greedy choice.
- Recall that $f_1 \leq f_2 \leq \ldots \leq f_n$

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```



Knapsack Problem

- The input is a set of items $a_1, \ldots, a_n$; item a_i has a size s_i and a value v_i.
- The goal is to place items of total size at most S such that sum of the value of itedwqms in the knapsack is maximized.
- In the **0-1 knapsack problem**, we have to accept or reject each item.
  - In the example below, where $S = 15$, the optimal strategy is to do parts A, B, F, and G for a total of 34 points.
- In the **fractional knapsack**, we can accept **fractions** of each item.
  - Here, the optimal solution is $(1, A), (1, B), (1, C), (1/6, D), (1, G)$, for a total value of $7 + 9 + 5 + 2 + 12 = 35$ and total size of $3 + 4 + 2 + 1 + 5 = 15$.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5

Greedy Strategy

- Sort items by their value-to-size ratio, process items in the sorted order, and accept full fraction of an item as long as it fits (first A, then B, etc.); for the last item, accept a fraction to completely fill the knapsack.
  - For 0-1 knapsack, this selects $C(2.5), G(2.4), A(2.33)$, and $B(2.25)$ for a profit of 33 (which is not optimal because $\{A, B, F, G\}$ has profit 34.
  - For factional knpasack, this selects $(1, C), (1, G), (1, A), (1, B), (1/6, D)$.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5
	7/3	9/4	5/2	12/6	14/7	6/3	12/5
=	2.33	2.25	2.5	2	2	2	2.4

Greedy Framework

- Fractional knapsack has **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to subproblems.
  - If we know a fraction f_1 of the first items a_1 is accepted, we know the rest of the items should be optimally packed in a space $S - f_1 size(a_1)$.

Greedy Framework

- Fractional knapsack has the **Greedy-choice property**
  - There is an optimal solution that takes the item x with maximum frequency f_{max}. Let d denote the weight-to-size ratio of x. Here, we have $x = C$ and $d = 2.5$.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5
	7/3	9/4	5/2	12/6	14/7	6/3	12/5
=	2.33	2.25	2.5	2	2	2	2.4

Greedy Framework

- Fractional knapsack has the **Greedy-choice property**
  - There is an optimal solution that takes the item x with maximum frequency f_{max}. Let d denote the weight-to-size ratio of x. Here, we have $x = C$ and $d = 2.5$.
    - Consider any solution O which takes a smaller frequency of x, say $f' < f_{max}$, e.g., $O' = (0.9, C), (1, G), (1, A), (1, B), (0.2\bar{6}, D)$.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5
	7/3	9/4	5/2	12/6	14/7	6/3	12/5
=	2.33	2.25	2.5	2	2	2	2.4

Greedy Framework

- Fractional knapsack has the **Greedy-choice property**
  - There is an optimal solution that takes the item x with maximum frequency f_{max}. Let d denote the weight-to-size ratio of x. Here, we have $x = C$ and $d = 2.5$.
    - Consider any solution O which takes a smaller frequency of x, say $f' < f_{max}$, e.g., $O' = (0.9, C), (1, G), (1, A), (1, B), (0.2\bar{6}, D)$.
    - We increase share of x in O from f' to f_{max}, e.g., from 0.9 to 1 in the above example.
    - To make room for a more fraction of x, we must decrease share of any set of other items (say with density d'). In the example above, the density of A may be decreased from 1 to 0.9.
    - The total value increases, by $(f' - f_{max})(d - d')$ is non-negative, e.g., $0.1(2.5 - 2.25) > 0$, i.e., the updated solution with greedy choice property is no worse than O.

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
Size	3	4	2	6	7	3	5
	7/3	9/4	5/2	12/6	14/7	6/3	12/5
=	2.33	2.25	2.5	2	2	2	2.4

Greedy Algorithms Summary

- Dynamic programming, is a powerful tool that applies for all problems with optimal substructure. But it is an overkill sometimes, for problems that have greedy-choice property.

Greedy Algorithms Summary

- Dynamic programming, is a powerful tool that applies for all problems with optimal substructure. But it is an overkill sometimes, for problems that have greedy-choice property.
- Greedy algorithm are “greedy" for local optimization with the hope it will lead to a global optimal solution, not always, but in many situations, it works.

Greedy Algorithms Summary

- Dynamic programming, is a powerful tool that applies for all problems with optimal substructure. But it is an overkill sometimes, for problems that have greedy-choice property.
- Greedy algorithm are “greedy" for local optimization with the hope it will lead to a global optimal solution, not always, but in many situations, it works.
- Typical greedy algorithms that you must know:
    - Huffman encoding
    - Prim's algorithm for Minimum Spanning Tree: at each time add a new node which is closest to the existing subtree.
    - Kruskal's algorithm: at each time, add the edge with minimum weight which will not create cycle after added.
    - Dijkstra's algorithm Single source shortest path.