ECE 235: Lec 8: Greedy Algorithm 10/18/01 (Ch. 16)

Pai H. Chou, University of California at Irvine

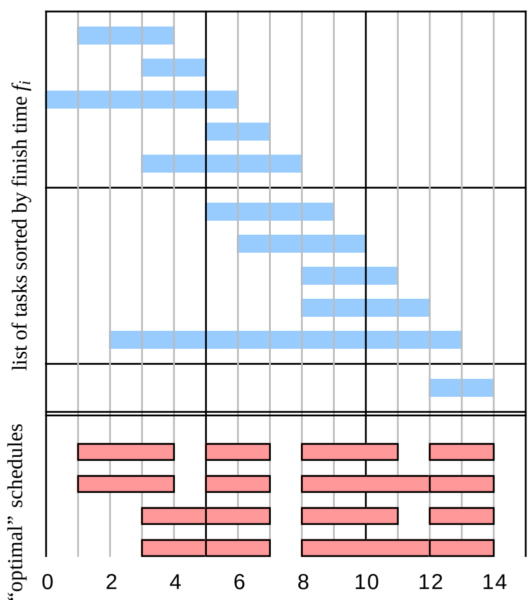
This lecture

- greedy algorithms: make choice that looks best at the moment
- activity selection problem
- Huffman (with Python programming hints)

1 Activity Selection problem

- Set S of n activities
- s_i = start time of activity i,
  f_i = finish time of i.
  assume $0 \le s_i < f_i < \infty$. (finite, positive/nonzero execution delay)
- i, j are *compatible*
  if $f_i < f_j$ implies $f_i < s_j$. (i.e., no overlap)
- Goal: find a maximal subset A of compatible activities
  - maximize $|A|$ = # of activities
  - NOT maximize utilization!!
  - there many be several optimal solutions, but it's sufficient to find just one.
- Greedy Solution: pick next compatible task with earliest finish time.

Example



Notation

- $S = \{a_1 \ldots a_n\}$ set of activities.
- a_0 = "start", a_{n+1} = "end" (fake) activities: $f_0 = 0, s_{n+1} = \infty$.
- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
  (i.e., set of activities compatible with a_i, a_j)
  $\Rightarrow S = S_{0,n+1}$
- assumption: sorted by finish time:
  $f_0 \le f_1 \le f_2 \le \cdots \le f_n \le f_{n+1}$
- Claim: if $i \ge j$ then $S_{ij} = \emptyset$.
  impossible to start later and finish earlier

Optimal substructure

- Let A = an optimal set (activity selection) for S
  $A_{i,j}$ = optimal for $S_{i,j}$.
- if $a_k \in A_{i,j}$ then
  - $A_{i,k}$ must be optimal solution to $S_{i,k}$
  - Proof: Assume there exist $A'_{i,k}$ with more activities than our $A_{i,k}$ (that is $|A'_{i,k}| > |A_{i,k}|$.
    $\Rightarrow$ we can construct a longer $A'_{i,j}$ by using $A'_{i,k}$ prefix. Contradiction to the original claim that A is optimal.

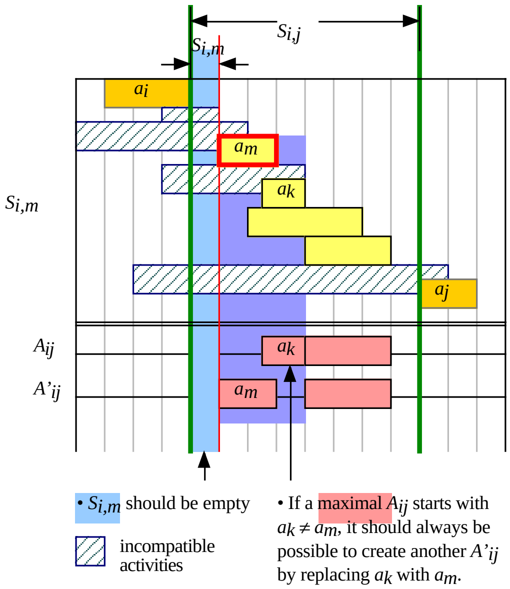
Formulation:

Want to maximize $c[0, n+1]$ where $c[i, j]$ = max # of compatible activities in $S_{i,j}$.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{i,j} = \emptyset \\ \max\limits_{i<k<j} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{i,j} \ne \emptyset \end{cases}$$

(note: $S_{i,k}$ and $S_{k,j}$ are disjoint and do not include a_k)

- looks like dynamic programming formulation, but
- we can do better with greedy.
- if $a_m \in S_{i,j}$ has earliest finish time, then
  1. $S_{i,m} = \emptyset$
     because if not empty, there must exist a_k that starts after a_i and finish before a_m,
     but if a_k finishes before a_m, then a_k should have been picked. Contradiction.
  2. a_m is a member of some maximal $A_{i,j}$
     if not, we can always construct $A'_{i,j} = A_{i,j} - \{a_k\} \cup \{a_m\}$, and $|A| = |A'|$.



What does this mean?

- earliest-finish-time-first is always as good as any optimal
- optimal substructures for later half don't depend on earlier optimal substructure

Greedy Algorithm (iterative version, p. 378)

GREEDY-ACTIVITY-SELECTOR(s, f)
    ▷ *assume f already sorted!*
$n \leftarrow length[s]$
$A \leftarrow \{a_1\}$
$i \leftarrow 1$
for $m \leftarrow 2$ to n
    do if $s_m \geq f_i$ ▷ *next compatible with earliest finish*
        then $A \leftarrow A \cup \{a_m\}$;
            $j \leftarrow m$
return A

Another way to view this:

- intuition - leave the largest remaining time
- What about "least duration"?

2 Greedy choice property

- global optimal is made up of the greedy choice plus an optimal subproblem
- prove greedy choice is a member of the global optimal
- Optimal substructure:
  showing that removing the greedy choice yields a solution to the subproblem.

Greedy vs. Dynamic programming: Knapsack problem

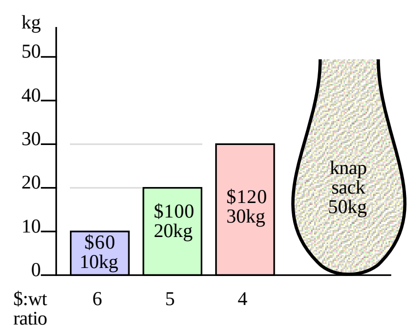
- Thief with a knapsack
- weight limited rather than volume limited
- Goal: maximize the value of stolen goods under weight limit

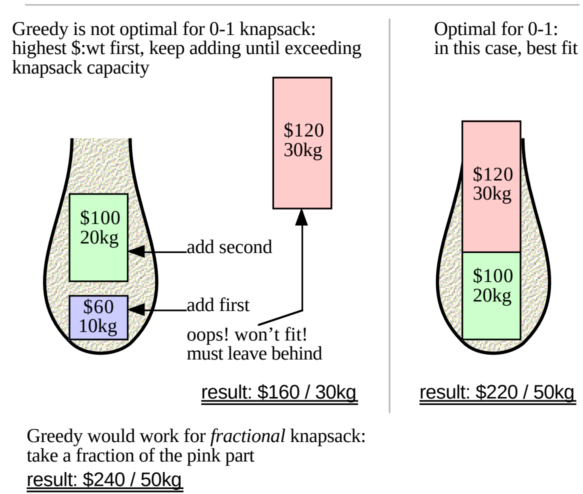
Variations:

- 0-1 knapsack problem: – whole items (watches, nuggets)
- fractional knapsack problem – can take parts (sugar, salt)

Important distinction:

- Greedy works for the fractional knapsack problem always take things with maximum value per unit weight
- Greedy does not work for 0-1 by counter example:
  - Limit: 50 lb, items \$60/10, \$100/20, \$120/30
  - Ratios \$6, \$5, \$4
  - Greedy solution: take item 1, item 2, but item 3 won't fit ⇒ get \$160/30
  - Optimal is take item 2, item 3, leave item 1 ⇒ \$220/50





How to solve 0-1 knapsack?

- dynamic programming, in $O(nW)$ time.
- idea: iterate over total weights up to the limit
- need to assume unit weight

3 Huffman codes

fixed-length code:

- same # bits for all characters

variable length code:

- more frequent $\Rightarrow$ use fewer bits (e.g. vowels like 'a' 'e' 'i' 'o' 'u' vs. "q" "z")

- Goal: minimize $\sum_{\text{codeword } c \in C} \text{frequency}(c) \times \text{bitlength}(c)$

- non-lossy! preserves the same information

Comparison example: 6 characters

	a	b	c	d	e	f
frequency	45	13	12	16	9	5
fixed length codeword	000	001	010	011	100	101
var-length codeword	0	101	100	111	1101	1100

Why is variable length better?

- fixed (3-bits per char) 100,000 chars $\Rightarrow$ 300,000 bits

- variable length: $(45 \cdot 1 + (13+12+16) \cdot 3 + (9+5) \cdot 4) = 224,000$ bits (25% saving)

problem: how do we know where to begin each codeword?

- prefix codes: no codeword is a prefix of another codeword. e.g., only `a` starts with 0. So, 00 means (`a a`).
  1 $\Rightarrow$ need to look at more bits:
  10 is prefix for either `b` (101) or `c` (100)
  11 is prefix for `d, e, f`

- Example: "001011101" uniquely parses as $0(\texttt{a}) \cdot 0(\texttt{a}) \cdot 101(\texttt{b}) \cdot 1101(\texttt{e})$, no ambiguity

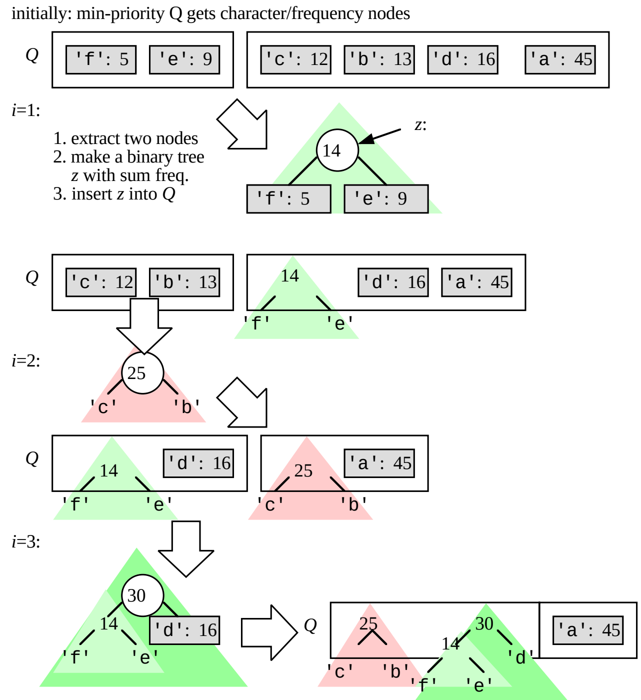
Algorithm

HUFFMAN(C) (page 388)
▷ *C is the array of nodes that carry (letter, frequency)*

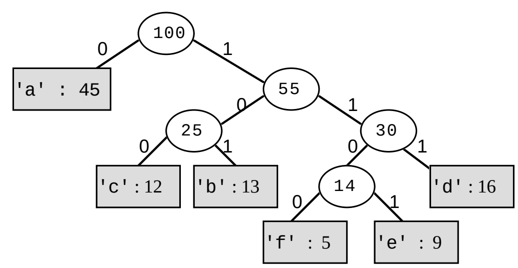
```
n ← |C|
Q ← C   ▷ put all elements into priority queue
for i ← 1 to n − 1
    do z ← MAKE NEW NODE
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

Python version

http://e3.uci.edu/01f/15545/huffman.html



continued... when all done,



Time complexity:

- $|n|-1$ calls, heap is $O(\lg n)$, so this is $O(n \lg n)$ for n characters

Correctness

Greedy property:

- C = alphabet. $x, y \in C$ are two chars with lowest frequencies. (most rarely used)

- then there exist an optimal prefix code such that
  - the codewords for x and y have the same length and
  - differ only by the last bit value.
  - If a, b are leaves at deepest depth and have higher frequency, then we can swap a, b with x, y and obtain a new tree with a lower cost of sum of freq $\cdot$ depth.

Optimal substructure:

- T' is an optimal tree for alphabet C'
  $\Rightarrow T$ is an optimal tree for alphabet C if
  - C and C' are the same except $x, y \in C$ while $z \in C'$,
  - f is the same except $f[z] = f[x] + f[y]$
  - we construct T from T' by replacing leaf node for z with internal node that is parent to x, y.