

COT 5407: Introduction to Algorithms

Giri Narasimhan

ECS 254A; Phone: x3748

giri@cis.fiu.edu

<http://www.cis.fiu.edu/~giri/teach/5407S17.html>

<https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494>

Greedy Algorithms

- Given a set of activities (s_i, f_i) , we want to schedule the maximum number of non-overlapping activities.
- GREEDY-ACTIVITY-SELECTOR (s, f)
 1. $n = \text{length}[s]$
 2. $S = \{a_1\}$
 3. $i = 1$
 4. **for** $m = 2$ **to** n **do**
 5. **if** s_m is not before f_i **then**
 6. $S = S \cup \{a_m\}$
 7. $i = m$
 8. **return** S

Why does it work?

- **THEOREM**

Let A be a set of activities and let a_1 be the activity with the earliest finish time. Then activity a_1 is in some maximum-sized subset of non-overlapping activities.

- **PROOF**

Let S' be a solution that does not contain a_1 . Let a'_1 be the activity with the earliest finish time in S' . Then replacing a'_1 by a_1 gives a solution S of the same size.

Why are we allowed to replace? Why is it of the same size?

Then apply induction! **How?**

Greedy Algorithms – Huffman Coding

- Huffman Coding Problem

Example: Release 29.1 of 15-Feb-2005 of TrEMBL Protein Database contains **1,614,107** sequence entries, comprising **505,947,503** amino acids. There are 20 possible amino acids. What is the minimum number of bits to store the compressed database?

~**2.5 G bits or 300MB.**

- How to improve this?

- Information: **Frequencies are not the same.**

Ala (A) 7.72	Gln (Q) 3.91	Leu (L) 9.56	Ser (S) 6.98
Arg (R) 5.24	Glu (E) 6.54	Lys (K) 5.96	Thr (T) 5.52
Asn (N) 4.28	Gly (G) 6.90	Met (M) 2.36	Trp (W) 1.18
Asp (D) 5.28	His (H) 2.26	Phe (F) 4.06	Tyr (Y) 3.13
Cys (C) 1.60	Ile (I) 5.88	Pro (P) 4.87	Val (V) 6.66

- **Idea:** Use shorter codes for more frequent amino acids and longer codes for less frequent ones.

Huffman Coding

2 million characters in file.

A, C, G, T, N, Y, R, S, M

IDEA 1: Use ASCII Code

Each need at least 8 bits,
Total = 16 M bits = **2 MB**

IDEA 2: Use 4-bit Codes

Each need at least 4 bits,
Total = 8 M bits = **1 MB**

Percentage
Frequencies

**IDEA 3: Use Variable
Length Codes**

A	22	11
T	22	10
C	18	011
G	18	010
N	10	001
Y	5	00011
R	4	00010
S	4	00001

How to Decode?

Need Unique decoding!
Easy for Ideas 1 & 2.
What about Idea 3?

110101101110010001100000000110

110101101110010001100000000110

2 million characters in file.

Length = ?

Expected length = ?

Sum up products of frequency times the code length, i.e.,

$$(.22 \times 2 + .22 \times 2 + .18 \times 3 + .18 \times 3 + .10 \times 3 + .05 \times 5 + .04 \times 5 + .04 \times 5 + .03 \times 5) \times 2 \text{ M bits} = 3.24 \text{ M bits} = \mathbf{.4 \text{ MB}}$$

Dynamic Programming

- **Activity Problem Revisited:** Given a set of n activities $a_i = (s_i, f_i)$, we want to schedule the maximum number of non-overlapping activities.
- **New Approach:**
 - **Observation:** To solve the problem on activities $A = \{a_1, \dots, a_n\}$, we notice that either
 - optimal solution does not include a_n
 - then enough to solve subproblem on $A_{n-1} = \{a_1, \dots, a_{n-1}\}$
 - optimal solution includes a_n
 - Enough to solve subproblem on $A_k = \{a_1, \dots, a_k\}$, the set A without activities that overlap a_n .

An efficient implementation

- Why not solve the subproblems on $A_1, A_2, \dots, A_{n-1}, A_n$ in that order?
- Is the problem on A_1 easy?
- Can the optimal solutions to the problems on A_1, \dots, A_i help to solve the problem on A_{i+1} ?
 - YES! Either:
 - optimal solution does not include a_{i+1}
 - problem on A_i
 - optimal solution includes a_{i+1}
 - problem on A_k (equal to A_i without activities that overlap a_{i+1})
 - but this has already been solved according to our ordering.

Dynamic Programming: Activity Selection

- Select the maximum number of non-overlapping activities from a set of n activities $A = \{a_1, \dots, a_n\}$ (sorted by finish times).
- Identify “easier” subproblems to solve.
 $A_1 = \{a_1\}$
 $A_2 = \{a_1, a_2\}$
 $A_3 = \{a_1, a_2, a_3\}, \dots,$
 $A_n = A$
- Subproblems: Select the max number of non-overlapping activities from A_i

Dynamic Programming: Activity Selection

- Solving for A_n solves the original problem.
- Solving for A_1 is easy.
- If you have optimal solutions S_1, \dots, S_{i-1} for subproblems on A_1, \dots, A_{i-1} , how to compute S_i ?
- The optimal solution for A_i either
 - Case 1: does not include a_i or
 - Case 2: includes a_i
- Case 1:
 - $S_i = S_{i-1}$
- Case 2:
 - $S_i = S_k \cup \{a_i\}$, for some $k < i$.
 - How to find such a k ? We know that a_k cannot overlap a_i .

Dynamic Programming: Activity Selection

• DP-ACTIVITY-SELECTOR (s, f)

1. $n = \text{length}[s]$
2. $N[1] = 1$ // number of activities in S_1
3. $F[1] = 1$ // last activity in S_1
4. for $i = 2$ to n do
5. let k be the last activity finished before s_i
6. if $(N[i-1] > N[k])$ then // Case 1
7. $N[i] = N[i-1]$
8. $F[i] = F[i-1]$
9. else // Case 2
10. $N[i] = N[k] + 1$
11. $F[i] = i$

How to output S_n ?
Backtrack!
Time Complexity?
 $O(n \lg n)$

Dynamic Programming Features

- Identification of **subproblems**
- **Recurrence relation** for solution of subproblems
- **Overlapping** subproblems (sometimes)
- Identification of a **hierarchy/ordering** of subproblems
- Use of table to store solutions of subproblems (**MEMOIZATION**)
- Optimal Substructure

Longest Common Subsequence

$S_1 =$ CORIANDER

CORIANDER

$S_2 =$ CREDITORS

CREDITORS

Longest Common Subsequence($S_1[1..9]$, $S_2[1..9]$) = CRIR

Subproblems:

- $LCS[S_1[1..i], S_2[1..j]]$, for all i and j [BETTER]

• Recurrence Relation:

- $LCS[i,j] = LCS[i-1, j-1] + 1$, if $S_1[i] = S_2[j]$

$LCS[i,j] = \max \{ LCS[i-1, j], LCS[i, j-1] \}$, otherwise

• Table ($m \times n$ table)

• Hierarchy of Solutions?

LCS Problem

LCS_Length (X, Y)

1. $m \leftarrow \text{length}[X]$
2. $n \leftarrow \text{Length}[Y]$
3. for $i = 1$ to m
4. do $c[i, 0] \leftarrow 0$
5. for $j = 1$ to n
6. do $c[0, j] \leftarrow 0$
7. for $i = 1$ to m
8. do for $j = 1$ to n
9. do if ($x_i = y_j$)
10. then $c[i, j] \leftarrow c[i-1, j-1] + 1$
11. $b[i, j] \leftarrow "$)"
12. else if $c[i-1, j] < c[i, j-1]$
13. then $c[i, j] \leftarrow c[i-1, j]$
14. $b[i, j] \leftarrow "$ ↑"
15. else
16. $c[i, j] \leftarrow c[i, j-1]$
17. $b[i, j] \leftarrow "$ ←"
18. return

LCS Example

		H	A	B	I	T	A	T
	0	0	0	0	0	0	0	0
A	0	0↑	1↖	1←	1←	1←	1↖	1←
L	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
P	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
H	0	1↖	1↑	1↑	1↑	1↑	1↑	1↑
A	0	1↑	2↖	2←	2←	2←	2↖	2←
B	0	1↑	2↑	3↖	3←	3←	3←	3←
E	0	1↑	2↑	3↑	3↑	3↑	3↑	3↑
T	0	1↑	2↑	3↑	3↑	4↖	4←	4↖

Dynamic Programming vs. Divide-&-conquer

- Divide-&-conquer works best when all subproblems are **independent**. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem.
- Dynamic programming is needed when subproblems are **dependent**; we don't know where to partition the problem.
For example, let $S_1 = \{\text{ALPHABET}\}$, and $S_2 = \{\text{HABITAT}\}$.
Consider the subproblem with $S_1' = \{\text{ALPH}\}$, $S_2' = \{\text{HABI}\}$.
Then, $LCS(S_1', S_2') + LCS(S_1 - S_1', S_2 - S_2') \neq LCS(S_1, S_2)$
- Divide-&-conquer is best suited for the case when no “overlapping subproblems” are encountered.
- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

Dynamic programming vs Greedy

1. Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. The solution comes up when the whole problem appears.

Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem, then reduce the problem to a smaller one. The solution is obtained when the whole problem disappears.

2. Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy. However, there are some problems that greedy can not solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.

Fractional Knapsack Problem

- Burglar's choices:

Items: x_1, x_2, \dots, x_n

Value: v_1, v_2, \dots, v_n

Max Quantity: q_1, q_2, \dots, q_n

Weight per unit quantity: w_1, w_2, \dots, w_n

Getaway Truck has a weight limit of B .

Burglar can take “fractional” amount of any item.

How can burglar maximize value of the loot?

- Greedy Algorithm works!

Pick the maximum possible quantity of highest value per weight item. Continue until weight limit of truck is reached.

0-1 Knapsack Problem

- Burglar's choices:

Items: x_1, x_2, \dots, x_n

Value: v_1, v_2, \dots, v_n

Weight: w_1, w_2, \dots, w_n

Getaway Truck has a weight limit of B .

Burglar cannot take “fractional” amount of item.

How can burglar maximize value of the loot?

- Greedy Algorithm does not work! Why?
- Need dynamic programming!

0-1 Knapsack Problem

- Subproblems?
 - $V[j, L]$ = Optimal solution for knapsack problem assuming a truck of weight limit L and choice of items from set $\{1, 2, \dots, j\}$.
 - $V[n, B]$ = Optimal solution for original problem
 - $V[1, L]$ = easy to compute for all values of L .
- Table of solutions?
 - $V[1..n, 1..B]$
- Ordering of subproblems?
 - Row-wise
- Recurrence Relation? [Either x_j included or not]
 - $V[j, L] = \max \{ V[j-1, L], v_j + V[j-1, L-w_j] \}$

1-d, 2-d, 3-d Dynamic Programming

- Classification based on the dimension of the table used to store solutions to subproblems.
- **1-dimensional DP**
 - Activity Problem
- **2-dimensional DP**
 - LCS Problem
 - 0-1 Knapsack Problem
 - Matrix-chain multiplication
- **3-dimensional DP**
 - All-pairs shortest paths problem