Start 2:40 PM   END: 4:40PM

Answer Sheet Problem 1 Solution

1. The idea will be similar to that of chain matrix multiplication. Just as in chain matrix multiplication, consider our choice of parenthesization as "partitioning" $L_1, L_2, \ldots, L_n$. Thus, if we represent this as a tree (similar to how we did so in lecture), the subchain substructure would be considering all possible partitions of the subtrees of $L_1, \ldots, L_n$.

b) Let $c(i,j)$ be the minimum cost of concatenating $L_i, \ldots, L_j$. Then we can define a recurrence like so:

$$c(i,j) = \min_{i \le k < j} \underbrace{c(i,k) + c(k+1,j)}_{\text{minimum cost of each subtree}} + \underbrace{L_{i-1} L_k L_j}_{\text{combining the sets of binary strings}}$$

c) for $i = 1$ to n:
    $c(i,i) = 0$ # No cost if just one set of binary strings
  for $s = 1$ to $n-1$:   # Solving the "diagonals"
    for $i = 1$ to $n-s$:
      $j = i + s$
      $c(i,j) = \min_{i \le k < j} c(i,k) + c(k+1,j) + L_{i-1} L_k L_j$
  return $c(1,n)$ # return minimum cost of computing
            # the concatenation of n different set
            # of strings $L_1, L_2, \ldots, L_n$.

d) Worst case time performance is $\Theta(n^3)$ since we have to loop through $s=1$ to $n-1$; $i=1$ to $n-s$; and k from i to j.

Ansner Sheet : Problem 2

2. a) Suppose we made the following greedy choice: choose a_j and b_K from the remaining elements of A and B such that a_j is the maximum remaining element of A and b_k is the maximum remaining element of B.

In the base case, when length of both A and B are 1, then the choice is trivial. Now, suppose $n > 1$.

Define $A_t = \{a_i \in A : a_i \neq a_t\}$
$B_t = \{b_i \in B : b_i \neq b_t\}$

When $n > 1$ we pick a_j and b_k to be the maximum remaining element from A and B respectively. Thus, the remaing subproblem is maximizing:
$a_j^{b_k} \cdot c(A_j, B_k)$
where $c(A_j, B_k)$ is defined to be the maximum of $a_0^{b_0} \cdots a_{n-2}^{b_{n-2}}$ from the remaining elements in A_j and B_k

```
def maximum_payout (A, B):
    a_max = maximum element of A; b_max = maximum element of B
    p = 1
    p = p * a_max^(b_max)
    A = A - a_max ; B = B - b_max
    while A is not empty:
        a_max = maximum element of A; b_max = max element of B
        p = p * a_max^(b_max)
        A = A - a_max ; B = B - b_max
    return p.
```

Answer Sheet: Problem 2

b) We now show this problem has the greedy-choice property. To show this, we need to show that our greedy choice is always contained in some globally optimal solution.

We know that each element of A and B is a positive ~~i~~nteger. Let $a_0^{b_0} \cdot a_1^{b_1} \cdot \ldots \cdot a_{n-1}^{b_{n-1}}$ be some optimal solution. Now let a_j be the maximum element of A.

Let the term $a_j^{b_k}$ be an element of the optimal solution.
Case 1: Suppose b_k was the maximum element of B. Then, we are done since we have shown that our greedy choice is in the optimal solution.

Case 2: Suppose b_k was not the maximum element of B. Then because we know A and B have only positive integers, the term $a_j^{b_k}$ would have been larger if b_k was the maximum element. Since a_j was the maximum element in A, that means the product $a_0^{b_0} \cdot a_1^{b_1} \cdot \ldots \cdot a_{n-1}^{b_{n-1}}$ was not actually optimal. Thus, by proof of contradiction, b_k must be the maximum element of B and this shows our greedy choice is always in the optimal solution.

Answer Sheet: Problem 2

c). Now we want to show the problem has the optimal substructure property. Since by part (b) we've already shown that our greedy choice is in the optimal solution, we need to show that the optimal solution to A_j and B_k where

$$A_j = A - a_j$$
$$B_k = B - b_k$$

where a_j and b_k are the maximum elements of A and B is also contained in the optimal solution.

We use the following cut and paste argument. Suppose not. Suppose that the optimal solution did not contain the optimal solution to A_j and B_k. That would mean there existed some ordering of the elements A_j and B_k such that the product is greater than the one in the optimal solution, which is a contradiction because we could have used that ordering and obtained a larger product. Thus, the original optimal solution must not have been optimal.

Answer Sheet: Problem 3

3. Let $\Phi(D_i) =$ number of non-null elements $-$ number of null elements. $+ 1$

Let D_0 be the initial state A with size 1 and null contents. In that case, $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ because the number of null elements can only at most be the number of non-null elements $+1$ since we only allocate to the list when it is "full"

If operation i is a push, then:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

There are two cases:

Case 1: $K+1 < \text{len}(A_{i-1})$ → length of A after $i-1$ operation.

In this case, it is unnecessary to allocate more memory. We know $c_i = 2$. Furthermore, $\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1}) = 1$ since we are adding one non-null element and 0 null elements

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2 + 1 = O(1)$$

Case 2: $K+1 = \text{len}(A_{i-1})$:

In this case, it is necessary to allocate more memory.

$$c_i = 2 + \text{len}(A_{i-1}).$$

However, now $\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1}) = -\text{len}(A_{i-1})$ since we have increased the number of null elements by $\text{len}(A_{i-1})$

$$\hat{c}_i = c_i + \Delta\Phi = 2 + \text{len}(A_{i-1}) - \text{len}(A_{i-1}) = 2 = O(1)$$

Answer Sheet : Problem 3

Now, suppose operation i is pop. Well, in this case, we never need to allocate more memory. Thus, the actual cost, $c_i = 2$ since we are setting the values of two variables.

$\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1})$ is always 0 because we are never adding or removing elements, just changing k.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 2 = O(1)$$

Thus, since we have shown each operation, push and pop have amortized cost $O(1)$, any sequence of n operations can have at most $O(n)$ cost.

Dynamic Programming, Greedy Algorithms, Amortized Analysis

...ysis

Data structure.

$\to D_n$ are states

...ost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

actual cost

$\Phi(D_n) - \Phi(D_0)$

...ure that $\Phi(D_i) \geq \Phi(D_0)$ $\forall i$

$\Phi(D_0) = 0$; $\Phi(D_i) \geq 0$ $\forall i$.

Example! Greedy Algorithms

Activity selection Problem, choose last activity to start compatible

Greedy choice Property:

Suppose $a_1, .., a_n \in S$;

$S_n = \{a_i \in S : f_i < s_n\}$

WTS a_n is an element of some optimal solution to S.

Let A be some maximum-size subset of mutually compatible activ. Let $a_k \in A$ be element with latest start time. Either $a_k = a_n$ or you can replace a_k with a_n.

```
def make_change(v):
    # initialize table
    for i in range(0, v+1):
        d = {0 if i = 0, ∞ otherwise
    for i in range(1, v+1):
        # for each value, loop through coin denom.
        for j in range(1, n+1) if i-x_j > 0
            d[i] = min(d[i], d[i-x_j] + 1)
    return d[v]
```

Optimal Substructure Problem:

WTS Let A_n is maximum-size subset of S_n.

Let A be solution to S then it must contain optimal solution to S_n, A_n.

Cut and paste.

...ming

...: optimal solutions to a problem incorporate optimal solutions to its subproblems

...ch: The procedure solves subproblems of sizes $j = 0, 1, \dots, n$ in that order.

...roblem: Define how to create subproblems that would eventually work back to the base case.

...docode: Initialize the data structure (list, table) with base case results. The loop bottom up to solve.

...$1 = x_1 < x_2 < \dots < x_n$ be n coin denominations. Give an algorithm to efficiently compute the minimum ... of coins needed to make change for v.

... 0

...$x_i) + 1)$ if $v \neq 0$ $\forall x_i$ s.t. $v - x_i > 0$

...ms

...Property: the greedy choice at each step yields a globally optimal solution. The proof ...globally optimal solution to some subproblem. It then shows how to modify the solution to ... greedy choice for some other choice, resulting in one similar, but smaller subproblem.

...red pair $M = (S, I)$ s.t.

M is weighted if it is associated with a strictly positive weight $w(x)$ to each element of $x \in S$. $w(A) = \sum_{x \in A} w(x)$ $\forall A \subseteq S$.

...e set

...mpty family of subsets of S, called the independent subsets of S, s.t. if $B \in I$ and $A \subseteq B$, then $A \in I$.

I is hereditary if it satisfies this property. $\emptyset \in I$

$\in I$, and $|A| < |B|$, then $\exists x \in B - A$ s.t. $A \cup \{x\} \in I$. We say that M satisfies the exchange property.

...le for Amortized Analysis

...:

...ngth and $A[i] = 1$

...0

...th

Suppose i^{th} increment resets t_i bits.

Cost is at most $t_i + 1 = c_i$

$\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$

$\Phi(D_i) - \Phi(D_{i-1}) = (\Phi(D_{i-1}) - t_i + 1) - \Phi(D_{i-1})$

$= -t_i + 1$

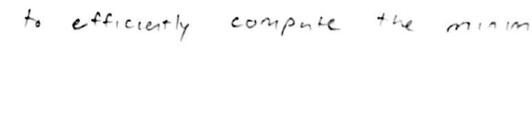
$\Phi(D_i)$ = number of 1's in counter after

Chain Matrix Multiplication Ex. Dynamic Programming

Given matrices $A_1, \dots, A_n$, compute the product in minimum number of operations.

(Intuitively, use this when your subproblems are like a "chain"). Subchain substructure:

$((AB)C)D$ $(AB)(CD)$ $A((BC)D)$

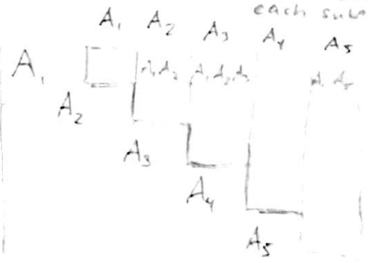


Intuitively, starting at top level: $A_1, \dots, A_n$, need to consider all partitions. Let $C(i,j)$ be the minimum cost of multiplying $A_i, \dots, A_j$.

$*C(i,j) = \min_{i \leq k < j} C(i,k) + C(k+1,j) + m_{i-1} m_k m_j$

minimum cost each subtree

combining the matrix



for $i = 1$ to n:
  $c(i,i) = 0$
for $s = 1$ to $n-1$:
  for $i = 1$ to $n-s$:
    $j = i + s$
    * (see above)
$\Theta(n^3)$

match the last

$c(i,j) = \min\{c(i-1, j-1) + \alpha(x_i, y_j),$
$c(i-1, j) + 1 \to$ leave x_i unmatched
$c(i, j-1) + 1 \to$ leave y_j unmatched

...se cases

$c(0, j) = j$; $c(i, 0) = i$

Sequence Alignment / Edit Distance

Sequences: $X = (x_1, \dots, x_m)$; $Y = (y_1, \dots, y_n)$

Cost(A) = # of unmatched characters $+ \sum_{(i,j) \in A} \alpha(x_i, y_j)$

$(n + m - 2|A|)$ cost of mismatched characters

Subproblem Substructure:

The optimal substructure comes from aligning prefixes of the sequences you want to align.

Let $C(i,j)$ be the minimum cost of aligning $x_1, \dots, x_i$ with $y_1, \dots, y_j$. Goal is to find $c(m,n)$.

fill out base cases

for $i = 0$ to m:
  $c(i, 0) = i$
for $j = 0$ to n
  $c(0, j) = j$
for $i = 1$ to m
  for $j = 1$ to n:
    $c(i,j) = \min\{c(i-1, j-1) + \alpha(x_i, y_j),$
    $c(i-1, j) + 1, c(i, j-1) + 1\}$

Floyd-Warshall Algorithm

for $i = 1$ to n
  for $j = 1$ to n
    $d[i][j] = \begin{cases} w(i,j) & \text{if edge} \\ 0 & \text{if } i = j \\ \infty & \text{otherw} \end{cases}$
for $k = 1$ to n:
  for $i = 1$ to n:
    for $j = 1$ to n:
      $d[i][j] = \min\{d[i][j], d[i][k] + d[k][j]\}$

All Pairs Shortest Path: Given a graph $G = (V, E)$ and $w: E \to \mathbb{R}$ find a shortest path between u and v for all $(u, v) \in V \times V$ (i.e.) minimize $\sum_{e \in P} w(e)$

Shortest Path Substructure: Recurse on intermediate vertices. WLOG assume $V = \{1, \dots, n\}$. Let $\delta(u, v, k)$ be the minimum weight of a path from u to v using only $1..k$ as intermediate vertices.

$\delta(u, v, k) = \min\{\delta(u, v, k-1), \delta(u, k, k-1) + \delta(k, v, k-1)\}$

Base case: $\delta(u, v, 0) = \begin{cases} w(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$

... longest palindromic sequence

Subproblem: longest Palindromic sequences on smaller substrings

Let $L(i,j)$ be the longest Palindromic sequence between $x_i \ldots x_j$

$$L(i,j) = \begin{cases} 0 & \text{if } n = 0 \text{ where } n \text{ is } len(x_i \ldots x_j) \\ 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \text{ \& } x_i = x_j \\ L(i+1, j-1) & \text{if } n > 2 \text{ \& } x_i = x_j \\ \max(L(i+1,j), L(i,j-1)) & \text{otherwise} \end{cases}$$

```
def largest_palindrome(x):
    # initialize base case
    for i = 1 to n:
        L(i,i) = 1
    for i = 1 to n-1:
        if (x_i = x_{i+1}):
            L(i, i+1) = 2
    for s = 1 to n-1:
        for i = 1 to n-s:
            j = i+s
        if x_i = x_j:
            L(i,j) = L(i+1, j-1) + 2
        if x_i ≠ x_j:
            L(i,j) = max(L(i,j-1), L(i+1,j))
```

1 PAGE NOTES

Greedy Algorithm matroid example

Task-Scheduling Problem

$S = \{a_0, a_1, \ldots, a_{n-1}\}$ unit time tasks

A schedule = permutation of S

$\underbrace{0 \to 1}_{\text{first}} \to \underbrace{2}_{\text{2nd}} \to \ldots \underbrace{\to n}_{\text{last task}}$

a_i has deadline $1 \le d_i \le n$ and a penalty $p_i > 0$ for failing to complete a_i at or before d_i.

Goal: choose S with minimum total penalty

Fact: A schedule can be arranged into a "canonical" form with all "on-time" tasks preceding all late tasks, and with on-time tasks sorted in increasing order by deadline.

$A \subseteq S$ is independent if they can be scheduled so that none of the tasks are late

Let $\mathcal{I}$ = independent sets of tasks.

Lemma: For $t = 0, 1, \ldots, n$, let $N_t(A)$ be the # of tasks in A whose deadline is t or earlier.

TFAE: (The following are equivalent)

a) A is independent

b) $N_t(A) \le t \quad \forall t$

c) If tasks scheduled in increasing order by d_i no task is late.

→ Claim: $(S, \mathcal{I})$ is a matroid.

Clearly S is a finite set

Hereditary: Obvious.

Exchange Property: Let $A \in \mathcal{I}$, $B \in \mathcal{I}$ and $|A| < |B|$.

Define k = largest t s.t $N_t(B) \le N_t(A)$

Such a k must exist since

$N_0(B) = 0 = N_0(A)$

$N_n(B) = |B| > |A| = N_n(A)$

So, $N_k(B) \le N_k(A)$ but $N_{k+1}(B) > N_{k+1}(A)$

So more tasks with deadline $k+1$ in B, than A

Pick $x \in B \setminus A$ with deadline $k+1$. We show $A' = A \cup \{x\} \in \mathcal{I}$

$\forall t \le k$, $N_t(A') = N_t(A) \le t$

For $t > k$, $N_t(A') \le N_t(B)$

$N_t(A') = N_t(A) + 1$

So for all t, $N_t(A') \le t \Rightarrow A' \in \mathcal{I}$

Now we need to assign a weight

$w(a_i) = p_i$

Max penalty ...

a_i ...