

53 exams taken.

Key - in exam

first name \_\_\_\_\_ last name \_\_\_\_\_ cats account \_\_\_\_\_

CMPS 102 Midterm #2, Spring 2004

This exam is a 100 minute, timed, closed book, closed notes exam. No calculators are allowed. Please read the questions carefully and do your work on the examination itself. Use the back sides of pages if necessary. If a problem is difficult for you, please move on to the next one. Try to demonstrate your knowledge of the method in your solutions. Answers that cannot be easily read may be penalized. There is a page at the end of the exam with some formulas and definitions that can also be used as scratch paper. When not otherwise stated, assume worst case analysis and unit cost.

7 questions, 50 points. Good luck!

question 1: \_\_\_\_\_ ( out of 8 )

question 2: \_\_\_\_\_ ( out of 8 )

question 3: \_\_\_\_\_ ( out of 8 )

question 4: \_\_\_\_\_ ( out of 5 )

question 5: \_\_\_\_\_ ( out of 6 )

question 6: \_\_\_\_\_ ( out of 6 )

question 7: \_\_\_\_\_ ( out of 9 )

Total \_\_\_\_\_ ( out of 50 )

1. (8 pts) Short answer

- (a) (2 pts) Many (bottom-up, iterative) dynamic programming algorithms take constant time to fill in each cell of their table. However, the Floyd-Warshall all-pairs shortest path algorithm on a graph with  $n$  vertices takes  $\Theta(n^3)$  time, but needs only  $\Theta(n^2)$  space. Explain why.

it actually fills in an  $O(n^3)$  size table, but only the last  $O(n^2)$  slice is needed to compute the next slice. Therefore the space for the slices can be re-used, so  $O(n^2)$  space is needed.

- (b) (2 pts) Many (bottom-up, iterative) dynamic programming algorithms take constant time to fill in each cell of their table. However, the Matrix Chain multiplication algorithm to determine the best way to multiply  $n$  matrices takes  $\Theta(n^3)$  time, but uses only  $\Theta(n^2)$  cells. Explain why.

Filling in each cell can take up to  $\Theta(n)$  time. (at least  $\frac{1}{3}$  the cells take at least  $n/2$  time) therefore the total time is  $\Theta(n^3)$ .

- (c) (2 pts) Give the reason why "I have an  $O(n^2)$  lower bound on the algorithm's worst case running time" is not a very strong statement.

The lower bound could be just 1 is  $1 \in O(n^2)$ . Saying the algorithm's running time is at least 1 is a very weak statement.

- (d) (2 pts) Recall that the longest common subsequence problem takes strings  $X = \langle x_1, \dots, x_m \rangle$  and  $Y = \langle y_1, \dots, y_n \rangle$  and computes a longest common subsequence of  $X$  and  $Y$ . Our dynamic programming solution defined a function  $L(i, j)$  for the length of the longest common subsequence of  $\langle x_1, \dots, x_i \rangle$  and  $\langle y_1, \dots, y_j \rangle$ . Give the recurrence for  $L(i, j)$ .

for  $i, j > 0$  then

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if } x_i = y_j \end{cases}$$

$$\begin{cases} \max \{ L(i-1, j); L(i, j-1) \} & \text{otherwise} \end{cases}$$

$$L(0, j) = 0$$

$$L(i, 0) = 0$$

2. (8 pts) Short Answer 2

- (a) (2 pts) Consider a variation on the recurrence for the median of medians selection algorithm where

$$T(n) = T\left(\frac{6n}{10}\right) + T(cn) + n.$$

How big can  $c$  be while keeping  $T(n) \in \Theta(n)$ ?

$c$  can be any constant less than  $\frac{4}{10}$ .  
if  $c = \frac{4}{10}$  then  $T(n)$  will be  $\Theta(n \lg n)$ .

- (b) (3 pts) Consider radix sort running on  $n$  numbers between 1 and  $n^4$ . Recall that radix sort treats the numbers as lists of digits, and performs passes through the data sorting on each digit. Give asymptotic expressions (as a function of  $n$ ) for the running time if each number is interpreted as:

- a list of  $4 \log n$  binary digits

$$\Theta(n \cdot 4 \lg n)$$

- a list of 4 digits in base  $n$

$$\Theta(n)$$

- a list of 2 digits in base  $n^2$ .

$$\Theta(n^2)$$

- (c) (3 pts) Briefly describe how branch and bound differs from backtracking

Backtracking performs a DFS of the tree of partial solutions, shortcutting any branches that cannot lead to feasible solutions.

Branch & Bound performs a best-first search of the tree, optimistically estimating the value of each node on the "frontier" (visited, but with unvisited children), and visits the children of the node with the best estimate.

3. Induction (8 pts). A 3-tree is like a binary-tree, but each internal node has exactly *three* children. More formally, every 3-tree has a distinguished node called the root and:

- (A) either the root is an external node and is the only node in the tree, or
- (B) the root is an internal node, and is connected to the roots of three disjoint subtrees, each of which is a 3-tree.

If  $T$  is a 3-tree, use  $H(T)$  for the height (number of edges in a longest root-external node path) of  $T$ , and  $N(T)$  for the number of nodes in  $T$ . You can use without proof that

- (a) if  $T$  has a single node its height is 0,
- (b) if  $T$  is a 3-tree by case (B) with subtrees  $T_1, T_2$ , and  $T_3$  then  $H(T) = 1 + \max\{H(T_1), H(T_2), H(T_3)\}$ .

Give a formal proof by induction on height that if  $T$  is a three-tree then

$$N(T) \leq \frac{3^{H(T)+1}}{2} - \frac{1}{2}.$$

Format your induction in the style taught in class.

for  $h \geq 0$ , let  $P(h)$  be the property that all 3-trees  $T$  with:  $H(T) = h$  have  $N(T) \leq \frac{3^{h+1}}{2} - \frac{1}{2}$ .

Base case:  $h = 0$ , show  $P(0)$ .

there is only 1 3-tree  $T$  with  $H(T) = 0$ , the single node 3-tree. in this case  $N(T) = 1$

$$\& \frac{3^{0+1}}{2} - \frac{1}{2} = 1 \quad \text{so} \quad N(T) \leq \frac{3^{H(T)+1}}{2} - \frac{1}{2}.$$

Inductive step. let  $h > 0$ , assume  $P(j) \forall 0 \leq j < h$  and show  $P(h)$ .

Let  $T$  be any height  $h$  3-tree. since  $h > 0$ ,  $T$  is formed from case (B) of the definition.

therefore  $T$  has 3 subtrees,  $T_1, T_2$  &  $T_3$ , and

$$N(T) = N(T_1) + N(T_2) + N(T_3) + 1 \quad (\text{for the root}).$$

Since  $H(T) = 1 + \max \{H(T_1), H(T_2), H(T_3)\}$ .

the height of each subtree is at most

$h-1$ . Therefore, using  $P(H(T_1))$ ,  $P(H(T_2))$ ,  $P(H(T_3))$

$$N(T) \leq 3^{\frac{H(T_1)+1}{2} - \frac{1}{2}} + 3^{\frac{H(T_2)+1}{2} - \frac{1}{2}} + 3^{\frac{H(T_3)+1}{2} - \frac{1}{2} + 1}$$

$$\leq 3 \left( \frac{3^{h-1+1}}{2} \right) - \frac{1}{2}$$

$$\leq \frac{3^{h+1}}{2} - \frac{1}{2} \quad \text{showing } P(h).$$

4. (5 pts) Indicator Variables.

Assume that we have an array  $A[1..n]$  that contains the integers 1 to  $n$  in a random order (each order is equally likely so each integer is equally likely to appear in each position) and are interested in how many integers  $i$  are located in slot  $A[i]$  (i.e. how many integers start in their correctly sorted position).

For each  $i \in \{1, \dots, n\}$ , let  $x_i$  be the indicator variable that is 1 when  $A[i] = i$  and 0 otherwise.

- What is the expected value of  $x_i$ ?

Since  $i$  is equally likely to appear in each slot of the array, the prob.  $i$  appears in  $A[i]$  is  $\frac{1}{n}$ , so the expected value of  $x_i = \frac{1}{n}$ .

- Calculate the expected number of integers  $i$  stored in their correctly sorted position. First express this as the expected value of a sum of  $x_i$ 's and then evaluate it.

The number of integers stored in their correct position is  $\sum_{i=1}^n x_i$ . Using  $E(\cdot)$  for expected value:

$$E\left(\sum_{i=1}^n x_i\right) = \sum_{i=1}^n E(x_i) = \sum_{i=1}^n \frac{1}{n} = 1$$



6. (6 pts) Divide and Conquer.

Consider the ThreeSort algorithm below:

```
ThreeSort (A[i..j])
  n := j-i+1; /* the number of elements to be sorted */
  if n < 2 then return;
  elseif n = 2 and A[i] > A[j] then swap A[i] and A[j];
  elseif n > 2 then
    third := n/3 rounded down;
    ThreeSort(A[i .. j-third]);    \\ sort first 2/3rds
    ThreeSort(A[i+third .. j]);    \\ sort last 2/3rds
    ThreeSort(A[i .. j-third]);    \\ sort first 2/3rds
  endif;
```

Let  $C(n)$  be the worst case number of comparisons between elements of  $A$  done by ThreeSort.

(a) (2 pts) write a recurrence (with boundary conditions) for  $C(n)$ .

$$\text{if } n \geq 3 \quad C(n) = 3C\left(\left\lfloor \frac{2n}{3} \right\rfloor\right)$$
$$C(1) = 0$$
$$C(2) = 1$$

(b) (4 pts) find a  $\Theta$ -expression for the  $C(n)$  defined by your recurrence. (You may use the approximation:  $\log_{3/2}(3) \approx 2.7$ )

From the master thm.

$$\log_b a = \log_{\frac{3}{2}}(3) \approx 2.7$$

$$f(n) = 0 \in O(n^{\log_{\frac{3}{2}}(3) - \epsilon}) \text{ for } \epsilon = \frac{1}{2}$$

So by case 1 of the master thm -

$$C(n) \in \Theta(n^{\log_{3/2}(3)})$$

7. (9 pts) Dynamic Programming (use this page and the next one)

Recall that the (discrete) Knapsack problem is:

Given a capacity  $C$ ,  $n$  items numbered from 1 to  $n$ , an array of integer weights  $w[1..n]$ , and an array of integer values  $v[1..n]$  for the items, find a subset of the items whose total weight is at most  $C$  and whose total value is as large as possible.

First, describe how (part of) an optimal solution to the original knapsack problem can be used to optimally solve a smaller related problem.

Second, prove that your description above is correct.

Third, use your description above to find a recurrence for value of the optimal solution in terms of the optimal values for subproblems.

Fourth, show how the value of an optimal solution to the original knapsack problem can be computed by an iterative bottom-up algorithm.

Finally, describe how keeping some additional information will enable those items taken by an optimal solution to be easily determined.

- ① Let  $S$  be an optimal solution to a knapsack instance  $C$ , items  $1..n$ .
- Ⓐ if  $n \in S$  then  $S - \{n\}$  is an optimal solution to the instance  $C - w(n)$ , items  $1..n-1$
- Ⓑ if  $n \notin S$  then  $S$  is also an optimal solution to the instance  $C$ , items  $1..n-1$ .

- ② Proof by contradiction. We prove each case Ⓐ & Ⓑ.
- Ⓐ  $n \in S$ . Let  $S'$  be a better solution than  $S - \{n\}$  to the  $C - w(n)$  items  $1..n-1$  instance. The  $S' + \{n\}$  is a feasible solution to the  $C, 1..n$  instance and has value greater than  $S = (S - \{n\}) + n$ , but this contradicts the optimality of  $S$ .
- 7

(extra space for dynamic programming question)

② (B) if there is a feasible solution  $s'$  better than  $s$  to the  $c, 1..n-1$  instance then  $s'$  is also a better-than- $s$  solution to the  $c, 1..n$  instance, contradicting the optimality of  $s$ .

③ let  $V(c, i)$  be the optimal value of the

$c, 1..i$  instance.  
 $V(c, 0) = 0$        $V(c, i) = -\infty$  if  $c < 0$

otherwise,  $V(c, i) = \max \begin{cases} V(c, i-1) \\ V(c-w(i), i-1) + v(i) \end{cases}$

④ compute table of  $V(c, i)$  values  $V(C, n)$  has value for orig. instance.

$V(0, i) = 0 \forall i \leq n; V(c, 0) = 0 \forall 0 \leq c \leq C.$

for  $i = 1$  to  $n$  do

for  $c = 0$  to  $C$  do

if  $c < w(i)$  <sup>new</sup>  $V(c, i) = V(c, i-1)$

else  $V(c, i) = \max [V(c, i-1); V(c-w(i), i-1) + v(i)]$



⑤ by keeping track of where  
the  $v(c, i)$  value came from

Let  $T(c, i)$  be a table ( $0 \leq c \leq C, 1 \leq i \leq n$ )

$$T(c, i) = \begin{cases} \text{t} & \text{if } v(c, i) = v(c - w(i), i - 1) + v(i) \\ \text{l} & \text{if } v(c, i) = v(c, i - 1) \end{cases}$$

---

$$c_{\text{left}} = C \quad \left\{ \begin{array}{l} \text{t} \\ \text{l} \end{array} \right. \text{ if } v(c, i) = v(c, i - 1).$$

for  $i = n$  down to 1 do

if  $T(c_{\text{left}}) = \text{t}$

print  $i$  "is taken"

$c_{\text{left}} = c_{\text{left}} - w(i)$