Computer Science 3711
Winter 2003

Midterm Exam

Answers

1. (9 marks) Circle the letters associated with the appropriate answers in the following multiple choice questions. Note that some of these questions may have more than one answer whose letter needs to be circled.

(a) (3 marks) Which of the following is true of the function $T(n) = 2^{n+1}$?

a) $T(n)$ is $\Omega(2^{n+3})$ b) $T(n)$ is $O(n^{1000})$ c) $T(n)$ is $\Theta(2^{n-10})$

Answer:

- **Part (a) is true**: $c2^{n+3} = 8c \cdot 2^n \leq 2 \cdot 2^n = 2^{n+1}$ when $c = \frac{1}{4}$.
- **Part (b) is false**: $2^{n+1} \leq cn^{1000} \rightarrow \frac{2^{n+1}}{n^{1000}} \leq c$ is false for any constant c relative to sufficiently large n.
- **Part (c) is true**: $c2^{n-10} = \frac{c}{2^{10}}2^n \leq 2 \cdot 2^n = 2^{n+1}$ for $c \leq 2^{11}$; moreover, $2^{n+1} = 2 \cdot 2^n \leq \frac{c}{2^{10}}2^n = c2^{n-10}$ for $c \geq 2^{11}$.

(b) (3 marks) Which of the following is true of the function $T(n) = 9^{\log_3 n}$?

a) $T(n)$ is $\Omega(1.2^n)$ b) $T(n)$ is $O(3^{\log_9 n})$ c) $T(n)$ is $\Theta(n^2)$

Answer: A key simplification here is $9^{\log_3 n} = (3 \cdot 3)^{\log_3 n} = 3^{\log_3 n} \cdot 3^{\log_3 n} = n \cdot n = n^2$.

- **Part (a) is false**: $n^2 \geq c \cdot 1.2^n \rightarrow \frac{n^2}{1.2^n} \geq c \rightarrow 2\log_2 n - n\log_2 1.2 \geq \log_2 c$ is not true for any constant c relative to sufficiently large n as the term $2\log_2 n - n\log_2 1.2$ goes to $-\infty$ as n goes to ∞.
- **Part (b) is false**: Note that $3^{\log_9 n} = n^{\log_9 3}$ and $\log_9 3 = \frac{1}{2}$. As $n^2/n^{\frac{1}{2}}$ goes to ∞ as n goes to ∞, $9^{\log_3 n} = n^2 \not\leq cn^{\frac{1}{2}} = c3^{\log_9 n}$ for any constant c when n is sufficiently large.
- **Part (c) is true**: $c_1 9^{\log_3 n} = c_1 n^2 \leq n^2 \leq c_2 n^2$ when $c_1 = c_2 = 1$.

(c) (3 marks) Which of the following is true of the function $T(n) = n^2$?

a) $T(n)$ is $\Theta((n-1)^2)$ b) $T(n)$ is $\Omega(57 \log_2 n)$ c) $T(n)$ is $\Theta((n+2)^2)$

Answer:

- **Part (a) is true**: $c(n-1)^2 \leq n^2$ as $c \leq n^2/(n-1)^2$ when $c = 1$ and $n_0 = 2$; moreover, $n^2 \leq c(n-1)^2$ as $n^2/(n-1)^2 \leq c$ when $c = 4$ and $n_0 = 2$.
- **Part (b) is true**: $c(57 \log_2 n) \leq n^2$ when $c = \frac{1}{57}$.
- **Part (c) is true**: $c(n+2)^2 \leq n^2$ as $c \leq n^2/(n+2)^2$ when $c = \frac{1}{9}$ and $n_0 = 1$; moreover, $n^2 \leq c(n+2)^2$ as $n^2/(n+2)^2 \leq c$ when $c = 1$ and $n_0 = 1$.

2. (12 marks)

a) (4 marks) Give the recurrence for the time complexity of the following recursive algorithm:

```
procedure FUNKY-REC(m, n)
    if (m == 1)
        sum = 0
        for (i = 1; i <= n; i++)
            for (j = 1; j <= m; j++)
                sum = sum + FUNKY-ITR(m) / i * j
        return(sum)
    else if (m > 1)
        sum = 0
        for i = 1 to n do
            l = FUNKY-REC(m/3, n)
            sum = FUNKY-ITR(n) * l
        return(sum)
    else
        return(m)
```

Assume that procedure `FUNKY-ITR(x)` runs in $O(x \log_2 x)$ time.

Answer:

$$T(m,n) = \begin{cases} O(1) & \text{if } m < 1 \\ O(n) & \text{if } m = 1 \\ nT(\frac{m}{3}, n) + O(n^2 \log_2 n) & \text{if } m > 1 \end{cases}$$

b) (8 marks) Derive an upper bound for $T(n)$ using any method discussed in class or in the textbook, where

$$T(n) = \begin{cases} 3 & n \leq 1 \\ 9T(n/3) + cn^2 & n > 1 \end{cases}$$

Answer: Personally, I prefer the iterative method for solving this recurrence:

$$\begin{aligned}
T(n) &= cn^2 + 9T(n/3) \\
&= cn^2 + 3^2T(n/3) \\
&= cn^2 + 3^2(c(n/3)^2 + 3^2T(n/3^2)) \\
&= cn^2 + 3^2c(n/3)^2 + (3^2)^2T(n/3^2) \\
&= cn^2 + 3^2c(n^2/3^2) + 3^4T(n/3^2) \\
&= cn^2 + cn^2 + 3^4T(n/3^2) \\
&= cn^2 + cn^2 + 3^4(c(n/3^2)^2 + 3^2T(n/3^3)) \\
&= cn^2 + cn^2 + 3^4c(n^2/3^4) + 3^4 \cdot 3^2T(n/3^3) \\
&= cn^2 + cn^2 + cn^2 + 3^6T(n/3^3) \\
&= cn^2 + cn^2 + cn^2 + 3^6(c(n/3^3)^2 + 3^2T(n/3^4)) \\
&= cn^2 + cn^2 + cn^2 + 3^6c(n/3^3)^2 + 3^6 \cdot 3^2T(n/3^4) \\
&= cn^2 + cn^2 + cn^2 + 3^6c(n^2/3^6) + 3^8T(n/3^4) \\
&= cn^2 + cn^2 + cn^2 + cn^2 + 3^8T(n/3^4) \\
&\quad \cdots \\
&= \sum_{k=1}^{\log_3 n} cn^2 + 9^{\log_3 n}T(1) \\
&= cn^2 \sum_{k=1}^{\log_3 n} 1 + n^2O(1) \\
&= cn^2 \log_3 n + n^2 \\
&= O(n^2 \log_3 n)
\end{aligned}$$

Note the inclusion of the "leaf-term" accounting for all recursive calls when $n \leq 1$; this term did not occur in the corresponding example in last year's midterm as, in that recurrence, $T(n) = 0$ when $n \leq 1$. The iterative method is particularly nice here in that it emphasizes how the 9-factor is canceled out by squaring the 1/3-factor relative to n in each iteration. A recursion tree would also show this, albeit a bit less clearly.

3. (9 marks) Consider the following algorithm:

```
sum = 0;
for i = 1 to n * n do
    j = T1(n)
    if (T2(i, j, n)
        for k = 1 to n do
            l = T3(n, k) - 2
            sum = (sum * j / l)
if ((sum % 2) == 0)
    sum = sum / 2
else
    sum = sum * 2
```

Give the asymptotic worst-case time complexity of this algorithm when the T1, T2, and T3 operations require:

a) $O(n^2)$, $O(1)$, and $O(1)$ time, respectively, and we don't know how many times T2 evaluates to **true**.

$$\begin{aligned} T(n) &= n^2T(T1) + n^2T(T2) + f(T2 = \text{true})nT(T3) \\ &= n^2T(T1) + n^2T(T2) + O(n^2)nT(T3) \\ &= n^2(O(n^2)) + n^2O(1) + O(n^3) \cdot O(1) \\ &= O(n^4) + O(n^2) + O(n^3) \\ &= O(n^4) \end{aligned}$$

b) $O(1)$, $O(n)$, and $O(1)$ time, respectively, and we know that T2 evaluates to **true** $O(n)$ times.

$$\begin{aligned} T(n) &= n^2T(T1) + n^2T(T2) + f(T2 = \text{true})nT(T3) \\ &= n^2T(T1) + n^2T(T2) + O(n)nT(T3) \\ &= n^2O(1) + n^2O(n) + O(n^2) \cdot O(1) \\ &= O(n^2) + O(n^3) + O(n^2) \\ &= O(n^3) \end{aligned}$$

c) $O(1)$, $O(1)$, and $O(1)$ time, respectively. and we know that T2 evaluates to **true** $O(n \log_2 n)$ times.

$$\begin{aligned} T(n) &= n^2 T(T1) + n^2 T(T2) + f(T2 = \text{true}) n T(T3) \\ &= n^2 T(T1) + n^2 T(T2) + O(n \log_2 n) n T(T3) \\ &= n^2 O(1) + n^2 O(1) + O(n^2 \log_2 n| \cdot O(1) \\ &= O(n^2) + O(n^2) + O(n^2 \log_2 n) \\ &= O(n^2 \log_2 n) \end{aligned}$$

4. (18 marks)

a) (4 marks) How are problems that have dynamic programming algorithms different from problems that have greedy algorithms? Phrase your answer in terms of the problem solution-space properties given in class.

Answer: Problems that have dynamic programming algorithms have recurrences with repeated subproblems, whereas problems that have greedy algorithms not only have recurrences in which subproblems do not repeat, but in addition these recurrences can be further simplified such that the subproblem that does occur can be solved in a locally-optimal "greedy" manner.

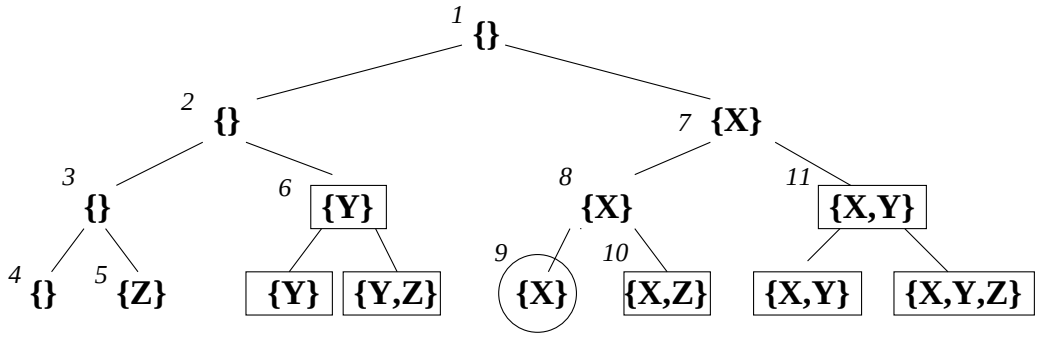
b) (14 marks) Consider the following algorithm:

```
DFS-V(i, sol, U, B, val)
    if (SIZE(sol) > B)
        print("pruned: solution too big for knapsack")
    else if (i == n)
        if (VALUE(sol) == val)
            print(sol)
    else
        DFS-V(i + 1, sol, U, B, val)
        DFS-V(i + 1, UNION-COPY(sol, U[i]), U, B, val)
```

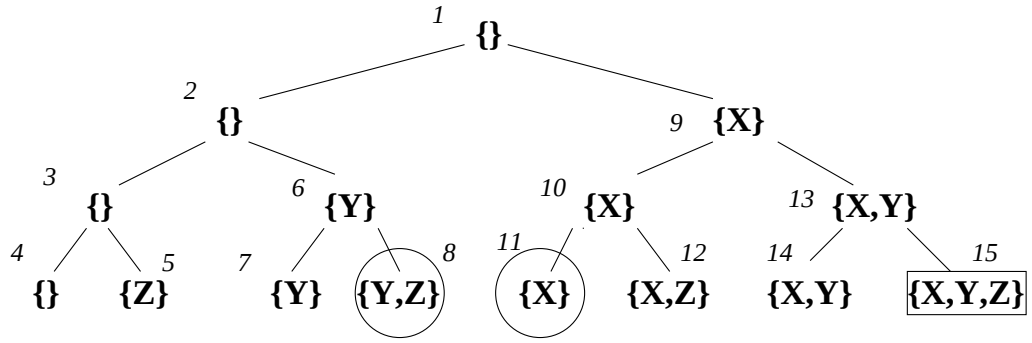
The problem solved here is a variant of 0/1 KNAPSACK that prints all viable knapsack-loads of a given value *val*. In this algorithm, U is the set of items, n is the number of items in U, B is the knapsack size bound, *sol* is a subset of U, SIZE(*sol*) returns the sum of the sizes of the items in *sol*, VALUE(*sol*) returns the sum of the values of the items in *sol*, and UNION-COPY($sol, U[i]$) returns a copy of *sol* to which the ith item in U has been added. To answer this question, sketch the implicit tree of solution-nodes generated by the algorithm above (marking leaf-nodes which are printed by a circle and nodes that are pruned with a square) with the call DFS-V($0, sol, U, B, val$) when $U = \{X, Y, Z\}$ such

that $size(X) = 1$, $size(Y) = 2$, $size(Z) = 1$, $value(X) = 2$, $value(Y) = 1$, and $value(Z) = 1$, sol is the empty set, and B and val have the following values:

i) (7 marks) $B = 1$ and $val = 2$:



ii) (7 marks) $B = 3$ and $val = 2$:



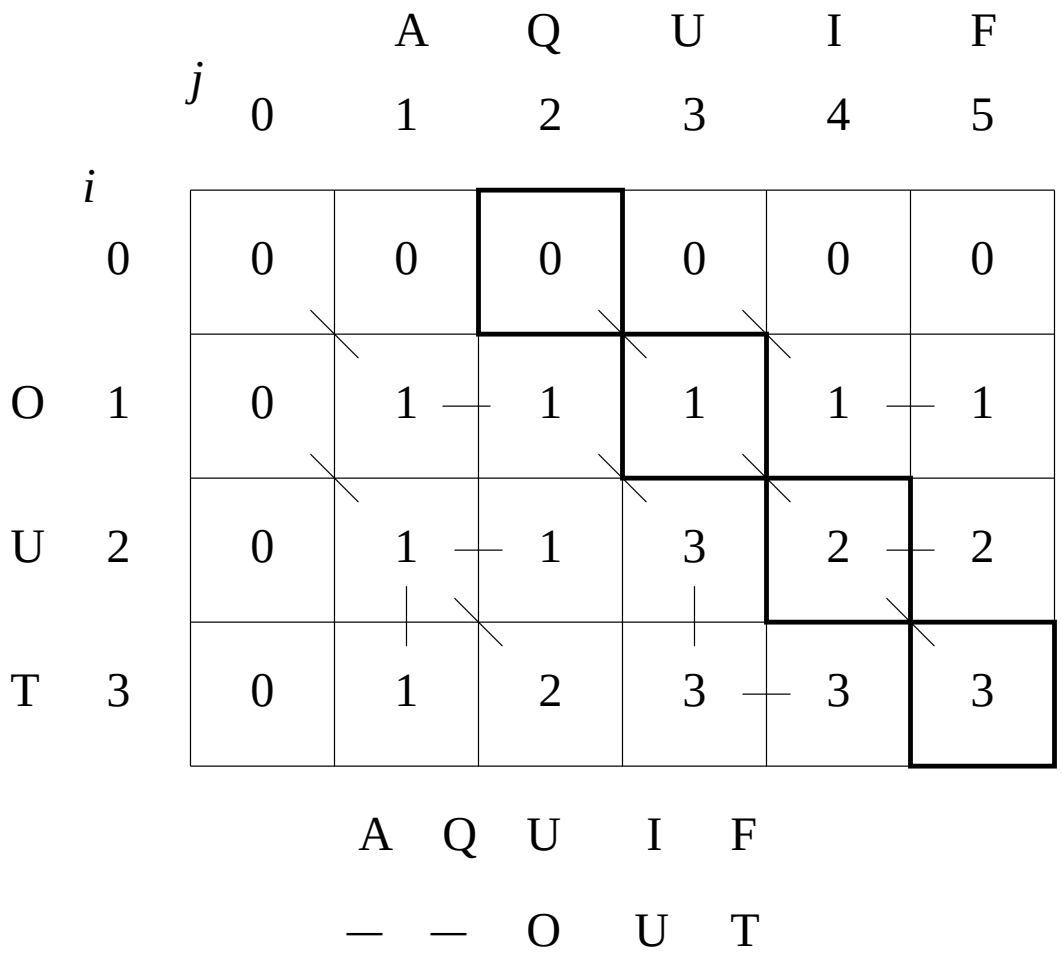
5. (12 marks) The length of the longest weighted common subsequence (LWCS) of two strings s and s' is defined by the following recurrence:

$$D(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ D(i-1, j-1) + 2 & \text{if } s(i) = s'(j) \\ D(i-1, j-1) + 1 & \text{if } s(i) \neq s'(j) \text{ and } SAME(s(i), s'(j)) \\ \max(D(i, j-1), D(i-1, j)) & \text{otherwise} \end{cases}$$

For any two symbols x and y, function $SAME(x, y)$ returns **true** if either (i) x and y are both vowels or (ii) x and y are both consonants, and **false** otherwise (recall that a vowel is one of the letters $\{'A', 'E', 'I', 'O', 'U'\}$ and a consonant is any letter of the alphabet that is not a vowel). Given the above, determine the longest weighted common subsequence of the strings **AQUIF** and **OUT** – that is, fill in the dynamic programming matrix given below, show all matrix-cell backpointers, show one of the

backpointer paths that gives an optimal LWCS, and show the LWCS corresponding to this path.

Answer: The matrix and LWCS specified by the recurrence given above is as follows:

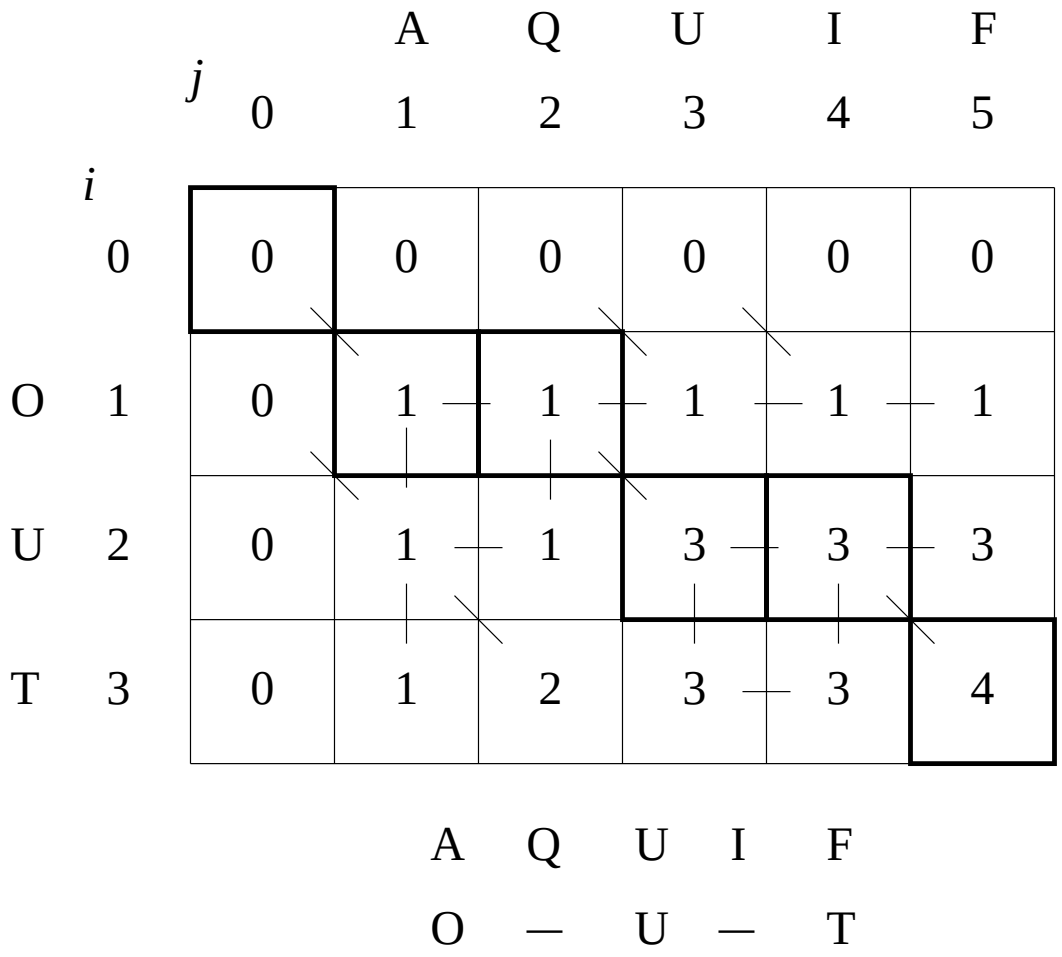


Note that relative to this recurrence, there is only one backpointer path; moreover, this path terminates when it reaches either the zero row or column. Some of you noticed that there seems to be an even better match to **OUT** – namely, the subsequence **AUF**. To obtain this solution, we need to change the recurrence to the following:

$$D'(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \max(COMP(i,j), D'(i,j-1), D'(i-1,j)) & \text{otherwise} \end{cases}$$

where function $COMP(i,j)$ returns $D'(i-1,j-1)+2$ if $s(i) = s'(j)$, $D'(i-1,j-1)+1$ if $s(i) \neq s'(j)$ but they are both vowels or both consonants, and

$-\infty$ otherwise. Under this revised recurrence, all options generating the optimal value for a cell get backpointers. The matrix and LWCS specified by this revised recurrence are as follows:



The revised recurrence above is one step towards specifying a general algorithm for approximate pattern matching; the interested are strongly encouraged to read the appropriate sections of [1] for further details.

References

[1] D. Gusfield. 1997. **Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology**. Cambridge University Press.