University of Toronto
Scarborough Campus
October 28, 2022

CSC C73 Midterm Test
Instructor: Vassos Hadzilacos

Aids allowed: One 8.5×11 'cheat sheet' (may be written on both sides)

Duration: Two hours

READ THE INSTRUCTIONS CAREFULLY

- There should be 9 pages in this exam booklet, including this cover page.
- Answer all questions.
- Put all answers in this booklet, in the spaces provided.
- For rough work, use the backs of the pages; ***these will not be graded***.
- The last two pages are blank and can be used for rough work or for overflow. ***Do not detach them*** from the booklet. If you use them for overflow, you must ***clearly indicate this*** on the page(s) containing the question(s) whose answer(s) you are providing in the overflow pages.
- In your answers you may use any result discussed in this course or its prerequisites merely by naming or describing it.
- Good luck!

QUESTION 1. (30 marks)

For each of the statements below, indicate whether it is true or false by circling the appropriate response. Do ***not*** justify your answers. No penalty for wrong answers, but don't rush to guess.

a. If all symbols have the same frequency, then Huffman's algorithm necessarily produces codewords whose lengths differ by at most one. . . . . **True** / **False** *T*

b. Regardless of the frequencies of the symbols, Huffman's algorithm will necessarily produce some codeword of length ***at most*** $\lfloor \log_2 n \rfloor$, where n is the number of symbols. . . . . **True** / **False** *T*

c. Depending on the frequencies of the symbols, Huffman's algorithm can produce some codeword of length $\Omega(n)$, where n is the number of symbols. . . . . **True** / **False** *T*

d. If P is a shortest path from node u to node v in a directed graph with non-negative edge weights, then P is also a shortest path from u to v if we increase the weight of each edge by one. . . . . **True** / **False** *F*

e. Dijkstra's algorithm can be implemented to run in $O(n^2)$ time, where n is the number of the graph's nodes. . . . . **True** / **False** *T*

f. If Dijkstra's algorithm adds node u before node v to the set R (of nodes to which it has found shortest paths from the source node s), then the weight of a shortest path from s to u is at most the weight of a shortest path from s to v. . . . . **True** / **False** *T*

g. If $T(n) = 4T(n/2) + n^2$ then $T(n) = \Theta(n^2)$. **True** / **False** *F*

h. Karatsuba's algorithm to multiply two n-bit integers runs in $O(n \log n)$ time. . . . . **True** / **False** *F*

i. The randomized order-statistics algorithm R-SELECT runs in $O(n \log n)$ time in the worst case. . . . . **True** / **False** *F*

j. The 0-1 knapsack problem can be solved in polynomial time if every item has weight at most 1000. . . . . **True** / **False** *T*

QUESTION 2. (30 marks)

Describe a ***divide-and-conquer*** algorithm that takes as input an array $A[1..n]$ of ***positive integers*** in arbitrary order, where $n \geq 2$, and returns a pair (i, j) such that $1 \leq i < j \leq n$ and the quantity $A[j] - A[i]$ has maximum value; note that this could be negative. Analyze the running time of your algorithm using the Master Theorem making clear the value of the relevant parameters a, b, and d. You may assume that $n \geq 2$ is a ***power of 2***. Do ***not*** justify the correctness of your algorithm.

Full marks for a correct and clearly described $\Theta(n)$ algorithm; significant partial marks for a correct and clearly described $\Theta(n \log n)$ algorithm. No credit for $\Omega(n^2)$ algorithms, even if correct. Note that we are looking specifically for a divide-and-conquer algorithm.

ANSWER: We give a divide-and conquer algorithm FindMaxDiff(A, ℓ, r), where $1 \leq \ell \leq r \leq n$, that solve $A[\ell..r]$, where $\ell < r$ and $r - \ell + 1$ is a power of 2. More precisely, if $\ell < r$, FindMaxDiff(A, ℓ, r) returns a four-tuple (i, j, min, max) such that $i < j$ and the quantity $A[j] - A[i]$ is maximized, min is the index of the minimum element on the first half of $A[\ell..r]$, and max is the index of the maximum element on the second half of $A[\ell..r]$; if $\ell = r$, it returns (ℓ, ℓ, ℓ, ℓ). Thus, to solve the given problem, we simply call FindMaxDiff$(A, 1, n)$, and return the pair consisting of the first two value (The version below doe that the length of $A[\ell..r]$ is a power of 2. With that assumption the base case is when $\ell = r - 1$, in which case we return (ℓ, r); and the conditions in line case that $i \neq j$ and $i_R \neq j_R$.)

FindMaxDiff(A, ℓ, r)

1 **if** $\ell = r$ **then return** (ℓ, ℓ, ℓ, ℓ)
2 **else**
3 $\quad m := \lfloor (\ell + r)/2 \rfloor$
4 $\quad (i_L, j_L, min_L, max_L) :=$ FindMaxDiff(A, ℓ, m)
5 $\quad (i_R, j_R, min_R, max_R) :=$ FindMaxDiff$(A, m + 1, r)$
6 $\quad min := min_L$
7 $\quad max := max_R$
8 $\quad (i, j) := (i_L, j_L)$
9 $\quad$ **if** $i = j$ **or** $(i_R \neq j_R$ **and** $A[j_R] - A[i_R] > A[j] - A[i])$ **then** $(i, j) := (i_R, j_R)$
10 $\quad$ **if** $i = j$ or $A[max] - A[min] > A[j] - A[i]$ **then** $(i, j) := (min, max)$
11 $\quad$ **return** (i, j, min, max)

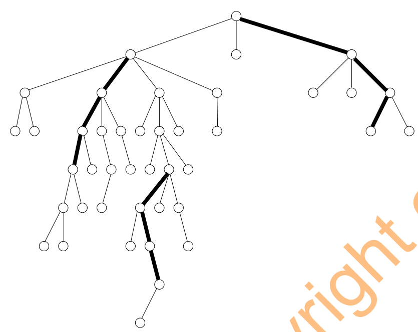
The running time $T(n)$ of this algorithm, where $n = r - \ell + 1$ (the length of $A[\ell..r]$), is de recurrence $T(n) = 2T(n/2) + c$, so in terms of the Master Theorem parameters we have $a = b = 2$ and $d = 0$. Since $a > b^d$, the third case of the theorem applie $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

(A $\Theta(n \log n)$ version of this algorithm returns just the pair (i, j) and spends linear time finding the min of the first half and the max of the second half to make po Then the recurrence for the running time is $T(n) = 2T(n/2) + cn$.)

QUESTION 3. (30 marks)

Let T be a rooted tree, not necessarily binary. Recall that a path in a rooted tree consists of a sequence of nodes such that each node other than the first is a child of the previous node in the sequence; the length of a path is the number of edges on it (i.e., one less than the number of nodes on it). Two paths in T ***intersect*** if they have a node in common. Given T and an integer $k \geq 0$ we want to find a maximum cardinality set of paths of T, each of length k, so that no two paths in the set intersect.

For example, the figure below shows a rooted tree and highlights three non-intersecting paths of length 3 (those shown with heavy edges). The set of these three paths is ***not*** a maximum cardinality set of non-intersecting paths of length 3: there are sets with more non-intersecting paths of length 3.



a. Describe an efficient ***greedy*** algorithm that, given a rooted tree T and an integer $k \geq 0$, finds a maximum cardinality set of non-intersecting paths of T, each of length k. You may assume that the tree's nodes are labeled by the positive integers 1 through n, the root is the node labeled 1, and the tree is specified by giving, for each node i, (a) the list of its children *children*(i) (empty, if i is a leaf), and (b) its parent *parent*(i) (0, if i is the root). Describe your algorithm in clear, high-level pseudocode; do not write detailed code. Do ***not*** justify the correctness of your algorithm.

Hint: In the above example, every path of length 3 intesects one of the three paths shown. What can be done to allow more non-intersecting paths of length 3?

ANSWER: The basic idea is to consider paths in decreasing de
(any) dee k. We kee
this manner, unle

NonIntersectingPaths(T, k)
find the depth of each node (using a BFS or preorder traversal, based on the children pointers)
$L :=$ list of nodes of depth $\geq k$ (these are the only possible last nodes of paths of length k)
sort L in decreasing depth
$P := \varnothing$
for each node u in L (in sorted order) **do**
    starting at u trace a path p of length k going up the tree, based on the parent pointers
    if all nodes on p are not on any path in P **then** $P := P \cup \{p\}$
return P

[part (b), and more space for part (a), if needed, on the next page]

[more space for part (a), if needed]

b. Analyze the running time of your algorithm in part (a) as a function of the number of nodes n of the tree T and the integer k.

<u>ANSWER:</u> *Each of line* $O(n)$ *time. Line 3 can be done in* $O(n \log n)$ *time using, say, heapsort. (Actually it can be done in* $O(n)$ *time since the de*
$0..n-1$*, so we can use bucket* n *time*
of line $O(k)$ *time. (For line 6 this is thanks to the parent pointers. For the te*
line 7 we can maintain a Boolean array of length n *that kee* P*, so*
we can check in constant time if a node is on one of the $O(nk)$ *time, and*
the overall running time of the algorithm is $O(n(\log n + k))$ *(or, using the above ob*
sorting L *in linear time,* $O(nk)$*).*

QUESTION 4. (30 marks)

Let $x[1..m]$, $y[1..n]$, and $z[1..m+n]$ be strings over some alphabet, where $m, n \in \mathbb{N}$. (Note that the length of z is the sum of the lengths of x and y.) Informally speaking, z is a "shuffle" of x and y if x and y can be broken into segments so that z is constructed as the interleaving of these segments. More precisely, the string z is a ***shuffle*** of x and y if, for some k, there are (***possibly empty***) strings $x_1, x_2, \ldots, x_k$ and $y_1, y_2, \ldots, y_k$ so that $x = x_1 x_2 \ldots x_k$, $y = y_1 y_2 \ldots y_k$, and $z = x_1 y_1 x_2 y_2 \ldots x_k y_k$. For example, $z =$ `paleolithic` is a shuffle of $x =$ `leoith` and $y =$ `palic`: take $x_1 = \epsilon$ (the empty string), $x_2 =$ `leo`, $x_3 =$ `ith`, $y_1 =$ `pa`, $y_2 =$ `l`, and $y_3 =$ `ic`.

For this question you are asked to develop a polynomial-time ***dynamic programming*** algorithm that, given strings $x[1..m]$, $y[1..n]$, and $z[1..m+n]$, returns true if z is a shuffle of x and y, and returns false otherwise.

a. Define clearly the subproblems that your dynamic programming algorithm will solve.

ANSWER: The sub $S(i,j)$ defined below for every i and j such that $0 \le i \le m$ and $0 \le j \le n$. Note that for any string w, $w[1..0]$ is the empty string.

$$S(i,j) = \begin{cases} \text{true}, & \text{if } z[1..i+j] \text{ is a shuffle of } x[1..i] \text{ and } y[1..j] \\ \text{fal} & \text{otherwise} \end{cases} \qquad (*)$$

b. Give a recursive formula to compute the solution to the subproblems in part (a). Do ***not*** explain why your formula is correct.

ANSWER:

$$S(i,j) = \begin{cases} \text{true}, & \text{if } i = 0 \text{ and } j = 0 \\ S(i, j-1) \wedge (z[j] = y[j]), & \text{if } i = 0 \text{ and } j > 0 \\ S(i-1, j) \wedge (z[i] = x[i]), & \text{if } i > 0 \text{ and } j = 0 \\ \big(S(i, j-1) \wedge (z[i+j] = y[j])\big) \vee \big(S(i-1, j) \wedge (z[i+j] = x[i])\big), & \text{if } i > 0 \text{ and } j > 0 \end{cases} \qquad (\dagger)$$

c. Describe your dynamic programming algorithm in pseudocode. Do not explain why your algorithm is correct.

ANSWER:

SHUFFLE$(x[1..m], y[1..n], z[1..m+n])$
$S(0,0) :=$ true
for $j := 1$ **to** n **do** $S(0,j) := \big(S(0,j-1)$ **and** $(z[j] = y[j])\big)$
for $i := 1$ **to** m **do** $S(i,0) := \big(S(i-1,0)$ **and** $(z[i] = x[i])\big)$
for $i := 1$ **to** m **do**
    for $j := 1$ **to** n **do**
        $S(i,j) := \Big(\big(S(i-1,j)$ **and** $z[i+j] = x[i]\big)$ **or** $\big(S(i,j-1)$ **and** $z[i+j] = y[j]\big)\Big)$
return $S(m,n)$

d. Analyze the running time of your algorithm as a function of m and n (the lengths of x and y).

ANSWER: The running time of this algorithm is dominated by the doubly-ne
which take $O(mn)$ time.

THE END

[Page for overflow or rough work — do not detach!]

[Page for overflow or rough work — do not detach!]

