CS 170 | Assignment 1 | January 30th, 1990

R. Seidel | | Due: February 6th, 1990

1. Which of the following statements are true? Prove your answers.
   (i) $n^2 \log n = O(n^3)$
   (ii) $n^3 = O(n^2 \log n)$
   (iii) $2^{n+1} = O(2^n)$
   (iv) $(n+1)! = O(n!)$
   (v) For any function f: $\mathbf{N} \rightarrow \mathbf{R}^+$
   $$f(n) = O(n) \Rightarrow [f(n)]^2 = O(n^2)$$

2. Let f and g be two functions in $\mathbf{N} \rightarrow \mathbf{R}^+$. In class we showed that the existence of $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ implies that $f(n) = O((g(n))$. What about the converse? Does the fact that $f(n) = O((g(n))$ imply that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists? Prove your answer.

3. Let f and g be two functions in $\mathbf{N} \rightarrow \mathbf{R}^+$. We say that $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Give a characerization in terms of limits for when $f(n) = \Theta(g(n))$.

4. Prove that $(\log n)^k = O(\sqrt{n})$ for any $k > 0$ but that $\sqrt{n} \neq O((\log n)^k)$.

5. $O(.)$ and $\Theta(.)$ behave for functions like "$\leq$" and "$=$" behave for numbers, i.e. they define an ordering relation for functions. Put the following functions into a non-decreasing sequence according to this order. Prove your answers. (Assume that ε is an arbitrary but fixed positive real number.)

$$n \log n,\ n^8,\ n^{1+\varepsilon},\ (1+\varepsilon)^n,\ (n^2+8n+\log^3 n)^4, \text{ and } n^2/\log n$$

$\sqrt{n} \le c\,[\log(n)]^k \quad \forall n \geq n_0$

$\frac{1}{2}\log n \le \log c + k\log(\log n)$

call $\log n = x$

$\frac{1}{2}x \le c' + k\log x$

$\Leftrightarrow \quad x \le c'' + k'\log x$

$f(n) = O(g(n)) \qquad g(n) = O(f(n))$

$\exists\, n_0, c>0$ s.t. $\forall n \geq n_0 \quad f(n) \le c\,g(n)$

$\exists\, n_0', c'>0$ s.t. $\forall n \geq n_0' \quad g(n) \le c' f(n)$

$\therefore \quad \forall n \geq \max(n_0, n_0') = N$

$f(n) \le c\,g(n)$ and $g(n) \le c' f(n)$

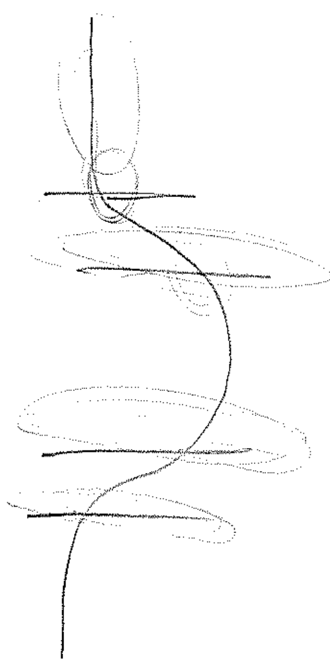
i.e. $f(n) \le cc' f(n)$ and $g(n) \le cc' g(n)$

$\frac{f(n)}{g(n)} \le c$ and $\frac{g(n)}{f(n)} \le c'$

$\frac{f(n)}{g(n)} \le c$ and $\frac{f(n)}{g(n)} \geq 1/c'$

2 4 11 b

2 4 11 a



CS170 ASSIGNMENT 1 Adnan Aziz

57/60 ~~54/60~~ corrected
R

Q1 (i) $n^2 \log n = O(n^3)$

True

2

Proof: $\lim_{n\to\infty} \left[n^2 \log n / n^3 \right] = \lim_{n\to\infty} \left(\frac{\log n}{n} \right) = 0$

$\therefore \quad n^2 \log n = O(n^3)$ QED

(ii) $n^3 = O(n^2 \log n)$

False Proof: lets assume the converse, ie $n^3 = O(n^2 \log n)$

Then, $\exists n_0, c > 0$ s.t. $\forall n \geq n_0$

$n^3 \leq c n^2 \log n \quad \forall n \geq n_0$

1 $\iff n \leq c \log n \quad \forall n \geq n_0$ justification?

but $\forall c$, we can find N st. $n > c \log n \quad \forall n > N$

thus we reach contradiction

Hence $n^3 \neq O(n^2 \log n)$ QED

(iii) $2^{n+1} = O(2^n)$

True

Proof: $2^{n+1} = (2) \times 2^n \quad \forall n \geq 1$

2

$\Rightarrow 2^n \leq 2 \times 2^n \quad \forall n \geq 1$ why do you need this?

$\therefore$ we have demonstrated the existence of c & n_0 (2 & 1 resp)

QED

(iv) $(n+1)! = O(n!)$

FALSE

Proof let assume the converse ie

$(n+1)! = O(n!)$

Then $\exists n_0, c > 0$ s.t. $\forall n \geq n_0$

2

$(n+1)! \leq c\, n! \quad \forall n \geq n_0$

iff $(n+1) \leq c \quad \forall \quad n \geq n_0$

clearly we can pick $n \geq \lceil c-1 \rceil$ $\therefore$ the conj

(v) For any fn $f: \mathbb{N} \to \mathbb{R}^+$ $\quad f(n) = O(n) \Rightarrow [f(n)]^2 = O(n^2)$

<u>FALSE</u>

consider $2^n = O(2^n)$

but $(2^n)^2 \neq O[2^{(n^2)}]$

O

as $2^{2n} \neq O[2^{n^2}]$

$4^n \neq O[2^{n^2}]$

For if this was so then

$\exists c > 0, \exists n_0$ s.t. $4^n \geq c\,2^{n^2}$ $\forall n \geq n_0$

$\Leftrightarrow n \log_2 4 \geq \log_2 c + n^2$ $\forall n \geq n_0$

$\Leftrightarrow 2n \geq \log_2 c + n^2$ $\forall n \geq n_0$

But $\forall c$ we can find N s.t. this rule fails $\forall$ all $n \geq N$.

QED

this statement is incorrect.
The problem is $f(n) = O(n)$,
not $f(n) = O(g(n))$

Q2 The fact $f(n) = O(g(n)) \not\Rightarrow \lim_{n\to\infty} f(n)/g(n)$ exists!

Proof: consider $f(n) = 1/2 \quad \forall\, n$ odd

$\qquad\qquad\qquad\quad = 1 \quad \forall\, n$ even

$\qquad\qquad g(n) = 2 \quad \forall\, n$ odd

$\qquad\qquad\qquad\quad = 1 \quad \forall\, n$ even

clearly $f(n)/g(n) = 1/4 \quad \forall\, n$ odd

$\qquad\qquad\qquad\qquad = 1 \quad \forall\, n$ even

10

$\therefore \lim_{n\to\infty} f(n)/g(n)$ doesn't exist

But $f(n) \le 1 \cdot g(n) \quad \forall\, n \geq 1$

ie $\exists\, n_0\ (=1),\ \exists\, c\ (=1)$ s.t. $f(n) \le c\,g(n)\ \forall\, n \geq n_0$

$\therefore f(n) = O(g(n))$

$\therefore f(n) = O(g(n))$ doesn't neccesarily mean that $\lim_{n\to\infty} f(n)/g(n)$ exists

Proved.

Q3. Given $f(n), g(n) : \mathbb{N} \to \mathbb{R}$

$f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ AND $g(n) = O(f(n))$

consider ~~this~~ special case where $\lim_{n\to\infty} f(n)/g(n)$ and $\lim_{n\to\infty} g(n)/f(n)$ exist. Then neither can be zero (else the other would not exist) hence one characterization is that if $0 < \lim_{n\to\infty} f(n)/g(n) < \infty$ (thus $0 < \lim_{n\to\infty} f(n)/g(n) < \infty$) then

Please be more obvious

out

$f(n) = \Theta[g(n)]$

There is another possibility ie that the sequence $[f(n)/g(n)]$ has no limit. Suppose that $f(n)/g(n)$ is bounded $\forall\, n \geq n_0$. Then by the Bolzano Weierstrass theorem it has at least one limit point. When a unique limit exists we have the previous case. When multiple

~~limit points exist, say $(l_1, l_2, \ldots)$ then if $\inf(l_1, l_2, \ldots) > 0$ we can see that $f(n) = \Theta(g(n))$.~~

~~In the case where $[f(n)/g(n)]$ is not bdd, then $g(n)/f(n) \to 0$. Thus $f(n) \neq \Theta(g(n))$~~

Q3. Given $f(n), g(n) : \mathbb{N} \to \mathbb{R}^+$

$f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ & $g(n) = O(f(n))$

Let $f(n) = \Theta(g(n))$.

$\iff$ $f(n) = O(g(n))$ AND $g(n) = O(f(n))$

$\iff$ $\exists\, n_0, \exists\, c > 0$ s.t. AND $\exists\, n_0', \exists\, c' > 0$ s.t.

$f(n) \le c\, g(n) \;\; \forall n \geq n_0$ $\quad$ $g(n) \le c' f(n) \;\; \forall n \geq n_0'$

$\iff$ $\forall n \geq \max(n_0, n_0')$ we have

$f(n) \le c\, g(n)$ and $g(n) \le c' f(n)$

$\iff$ $f(n)/g(n) \le c$ and $g(n)/f(n) \le c'$

$\iff$ $f(n)/g(n) \le c$ and $f(n)/g(n) \geq \frac{1}{c'}$

$\iff$ $[f(n)/g(n)]$ takes values only in some finite interval of $\mathbb{R}^+$ for all $n \geq$ some N

ie $f(n) = \Theta\, g(n)$

$\iff$ $\exists N, \exists\, a, b \in \mathbb{R}^+$ s.t.

$a \leqslant f(n)/g(n) \le b \quad \forall n > N$ $\quad$ (It is not necc. for a limit to exist)

10

Q3. f & g are fns in $\mathbb{N} \to \mathbb{R}^+$

$$f(n) = \Theta[g(n)]$$
$$\iff f(n) = O(g(n)) \ \& \ g(n) = O(f(n))$$

To obtain a characterization in terms of limits for when $f(n) = \Theta(g(n))$

(*)

$$f(n) = \Theta[g(n)] \iff f(n) = O(g(n)) \text{ AND } g(n) = O(f(n))$$
$$\iff \lim_{n\to\infty} f(n)/g(n) \text{ exists } \geq 0 \text{ AND } \lim_{n\to\infty} g(n)/f(n) \text{ exists } \geq 0$$

For both conditions to hold simultaneously it is nece & suff that $\left[\lim_{n\to\infty} f(n)/g(n) \text{ exists \& is } > 0\right]$ and $\left[\lim_{n\to\infty} g(n)/f(n) \text{ exists \& is } > 0\right]$

(*) N.B. there is a trivial case ie $f(n) = g(n) = 0$ where $f(n) = O(g(n))$ & $g(n) = O(f(n))$ Dis counting this case the above analysis holds

Q4. T.P.T. $(\log n)^k = O(\sqrt{n})$ $\forall k > 0$
but $\sqrt{n} \neq O[(\log n)^k]$

$d \frac{\log(n)^k}{\sqrt{n}}$

consider $\lim_{n\to\infty} (\log n)^k / \sqrt{n}$ (∞/∞ form)

Applying L'Hopitals rule $\lceil k \rceil$ times, we obtain an expression of the form

$\frac{k (\log(n))^{k-1} \frac{1}{n}}{d\sqrt{n}}$

$$\boxed{\frac{(k)(k-1)\cdots(k-\lceil k\rceil+1)\left(\frac{1}{n}\right)^{\lceil k\rceil}(\log n)^{k-\lceil k\rceil}}{(\frac{1}{2})(-\frac{1}{2})\cdots(\frac{1}{2}-\lceil k\rceil+1)(n)^{1/2-\lceil k\rceil}}}$$ wrong

correct

now have to also worry about $1/n$ in numerator!

$= \frac{c\,(\log n)^{(k-\lceil k\rceil)}}{\sqrt{n}}$, if $k \notin \mathbb{Z}$ then $k - \lceil k\rceil < 0 \Rightarrow$ Num $\to 0$
if $k \in \mathbb{Z}$ then $k = \lceil k\rceil \Rightarrow$ num $\to c$
but denominator $\to \infty$

$\therefore$ limit $= 0$

$\therefore (\log n)^k = O(\sqrt{n})$ QED

~~12~~ +3

15

again, $\sqrt{n} \neq O[(\log(n))^k]$

~~Proof~~ consider $a(n)$ & $b(n)$ general fns $\mathbb{N} \to \mathbb{R}^+$
then $a(n) = O(b(n)) \Leftrightarrow (e^{a(n)} = O(e^{b(n)})$
so $\sqrt{n} = O[\{\log(n)\}^k]$
$\Leftrightarrow e^{\sqrt{n}} = O[e^{\log(n)^k}] = O[n$

Proof: $\sqrt{n} = O[\log(n)^k]$
$\Leftrightarrow \exists c, \exists n_0$ s.t. $\sqrt{n} \le c[\log(n)]^k$ $\forall n \geq n_0$
$\Leftrightarrow \frac{1}{2}\log n \le \log c + k \log(\log n)$ $\forall n \geq n_0$
$\Leftrightarrow x \le c' + k' \log x$ $\forall x \geq \log n_0$ ($x = \log n$)

please use easily distinguish variable names

clearly this cannot be; for any fixed c' & k' we will reach an x_0 s.t. $x > c' + k'\log x$ $\forall x \geq x_0$
$\therefore \sqrt{n} \neq O[\log(n)^k]$

QED

Q5. $O(\cdot)$ & $\Theta(\cdot)$ behave like "$\le$" & "$=$"
To put $n\log n$, n^8, $n^{1+\epsilon}$, $(1+\epsilon)^n$, $(n^2+8n+\log^3 n)^4$, $n^2/\log n$ in an order

Answer $(1+\epsilon)^n > n^8 = (n^2+8n+\log^3 n)^4 > n^2/\log n > n\log n$

also depending on ϵ, $n^{1+\epsilon}$ will take different position

(i) $\forall\ \epsilon > 7$, $n^{1+\epsilon} < (1+\epsilon)^n$ & $n^{1+\epsilon} > n^8$

(ii) $\forall\ \epsilon = 7$, $n^{1+\epsilon} = n^8$

15

(iii) $\forall\ \epsilon \in [1,7)$, $n^{1+\epsilon} > n^2/\log n$ & $n^{1+\epsilon} < n^8$

(iv) $\forall\ \epsilon \in (0,1)$, $n^{1+\epsilon} < n^2/\log n$ & $n^{1+\epsilon} > n\log n$

Proof: $(1+\epsilon)^n > n^x \quad \forall\ \epsilon > 0,\ \forall\ x \in \mathbb{R}$ (Thm)

also $\lim_{n\to\infty} \dfrac{(n^2+8n+\log^3 n)^4}{n^8} = 1 = \lim_{n\to\infty} \dfrac{n^8}{(n^2+8n+\log^3 n)^4}$

$\therefore\ n^8 = (n^2+8n+\log^3 n)^4$

also $\lim_{n\to\infty} \dfrac{n^2/\log n}{n^8} = 0 \quad \therefore\ n^2/\log n = O(n^8)$

but $n^8 \ne O(n^2/\log n)$

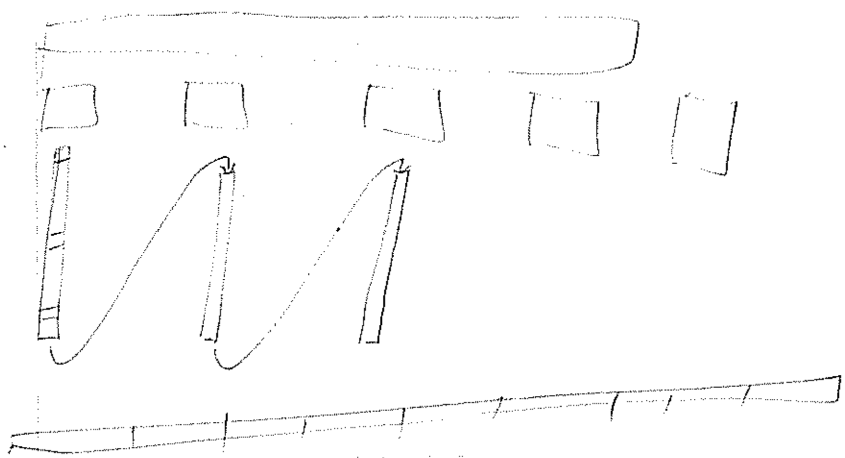
Also $n\log n = O(n^2/\log n)$ $\because \lim_{n\to\infty} \dfrac{n\log n}{n^2/\log n} = 0$

An introd to Fourier

analysis &

generalized fns

M. J. Lighthill



a b a ab

CS 170 **Assignment 2** **February 13th, 1990**
R. Seidel **Due: February 20th, 1990**

1. Consider the universe $U = \{0,1,2,3,4,5\}$ and a hash table of size 3. What is the smallest real number c so that the set $\mathbf{H} = \{h_1,h_2,h_3,h_4\}$ of functions from U to $\{0,1,2\}$ is c-universal, where

$$h_1(x) = x \bmod 3 \qquad h_2(x) = x^2 \bmod 3$$
$$h_3(x) = (2x+1) \bmod 3 \quad h_4(x) = x \bmod 2.$$

2. Show the AVL tree formed by inserting the number 1,2, . . . ,20 in order.

3. Show an AVL tree with a node whose deletion results in a non-AVL tree that cannot be made into an AVL tree by only one (single or double) rotation. Draw the tree, specify the node, and explain why the resulting tree cannot be balanced with one rotation.

4. A **concatenate** operations takes two binary search trees T_1 and T_2 where all keys in T_1 are less than all keys in T_2 and produces a new binary search tree T for the union of the keys in T_1 and T_2 (the old trees can be destroyed in the process).

   Design an algorithm to concatenate two AVL trees into one valid AVL tree. The worst case running time of the algorithm should be $O(h)$, where h is the maximal height of the two trees.

5. The AVL tree algorithms presented in class assumed that with every node in a tree one stored the height of the subtree rooted at that node. It is not difficult to see that it would suffice just to store the "balance factors" -1,0,+1, depending on whether the left or right subtree has greater height or whether their heights are equal. To represent these three values one needs three bits.

   Suggest a method for implementing AVL trees so that only one extra bit per node is necessary to store the balance information.

EXTRA CREDIT: Suggest a method for implementing AVL trees so that NO extra bit at all is necessary to store balance information.

6. Develop a technique to initialize an entry of an array $A[1 \cdots m]$ to zero the first time it is accessed, thus obviating the need to spend $O(m)$ time on initializing the entire array. Your solution is allowed to use additional storage.

   (*Hint:* Maintain a pointer for each initialized entry to a back pointer on a stack. Each time an entry is accessed, verify that the contents are not random by making sure the pointer in that entry points to the active region on the stack and that the back pointer points to the entry.)

free 35 mm

100 rolls Kodachrome 35¢

CS170 ASSIGNMENT 2 Adnan Aziz

70/75

ANS1 $U = \{0,1,2,3,4,5\}$ Hash table size $= 3$
Smallest real # c s.t. $H = \{h_1, h_2, h_3, h_4\}$ of fns from $U \to \{0,1,2\}$ is c universal where
$h_1(x) = x \bmod 3$ $h_2(x) = x^2 \bmod 3$
$h_3(x) = (2x+1) \bmod 3$ $h_4(x) = x \bmod 2$

By defn $\mathcal{H}$ is $c$universal $\Leftrightarrow \forall x, y \in \mathcal{U}; x \neq y$
$|\{h \in \mathcal{H}_x \mid h(x) = h(y)\}| \leq c|\mathcal{H}|/t$

15

Consider all sets $\{x, y : x \in \mathcal{U}, y \in \mathcal{U}\}$

✓ $\Rightarrow h(x) \neq h(y)$
× $\Rightarrow h(x) = h(y)$

	h1	h2	h3	h4	# of collision
0,1	✓	✓	✓	✓	= 0
0,2	✓	✓	✓	×	= 1
0,3	×	×	×	✓	= 3
0,4	✓	✓	✓	✓	= 1
0,5	✓	✓	✓	✓	= 0
1,2	✓	✓	×	✓	= 1
1,3	✓	✓	✓	×	= 1
1,4	×	×	×	✓	= 3
1,5	✓	✓	×	×	= 2
2,3	✓	✓	✓	✓	= 0
2,4	✓	✓	×	×	= 2
2,5	×	×	×	✓	= 3
3,4	✓	✓	✓	✓	= 0
3,5	✓	✓	✓	×	= 1
4,5	✓	✓	×	✓	= 1

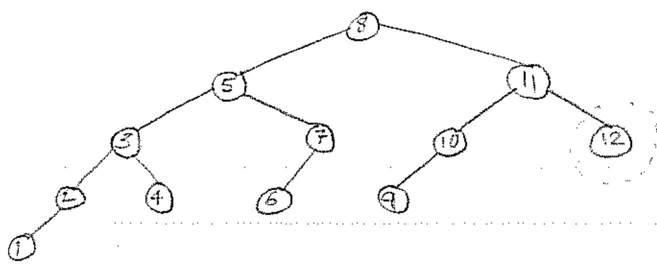
The pairs (0,3) (1,4) (2,5) result in 3 collisions each.
$\therefore 3 \leq c \times 4/3$
$\therefore c \geq 2.25$
$\therefore$ The smallest real # c is 2.25

ANS3. Example:

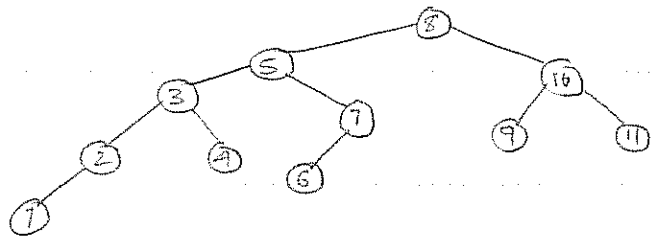
(FIBONACCI TREE of HT. 5)

/* DELETION OF NODE 12 */

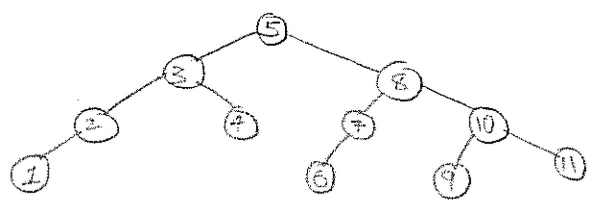
10



rotate right at 11



rotate right at 8

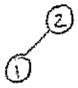


In general, in the worst case, a deletion of a node may require a rotation at every node along the search path. Consider for instance deletion of the right most node of a Fibonacci tree. In this case, the deletion of any single node leads to a reduction of the height of the tree [recall the AVL recursion i.e $N(t_h) \geq [(\sqrt{5}+1)/2]^h - 1$ from $N(t_h) \geq N(t_{h-1}) + N(t_{h-2}) + 1$] In addition the deletion of its right most node requires the maximum # of rotations. This is born out in the example above, where removing 12 results in the tree at 11 becoming unbalanced. The rotation at 11 to balance it results in the tree at 8 becoming unbalanced thus

ANS2

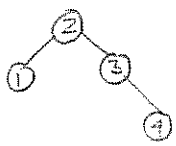
(i)

(ii)

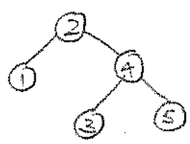


(iii)

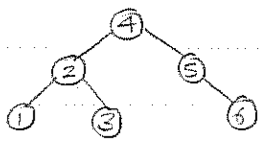
(iv)



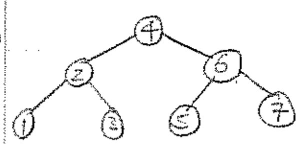
(v)



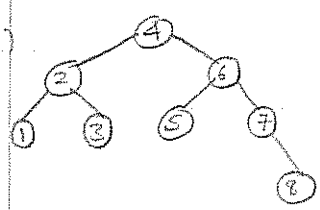
(vi)



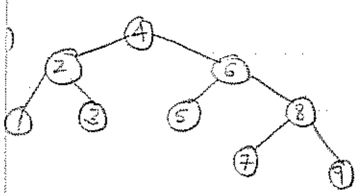
(vii)



(viii)

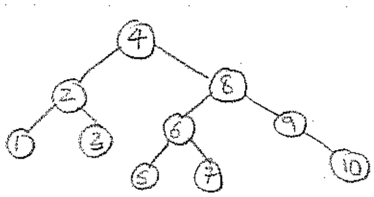


(ix)

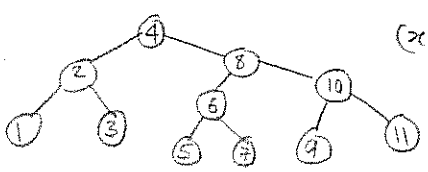


10

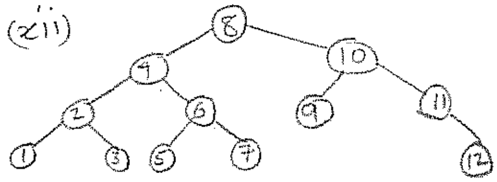
(x)



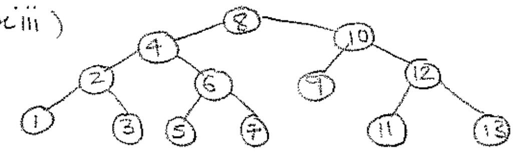
(xi)



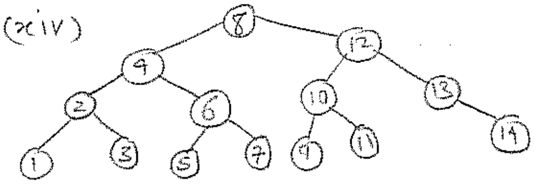
(xii)



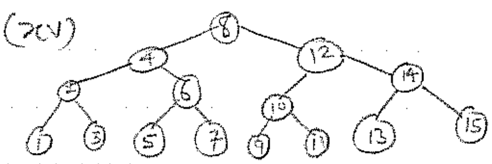
(xiii)



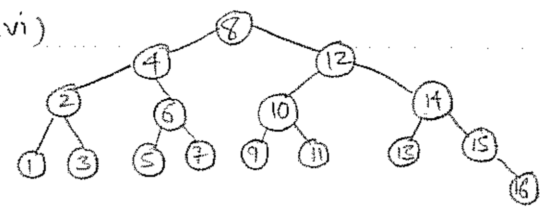
(xiv)



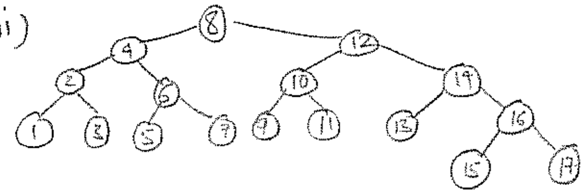
(xv)



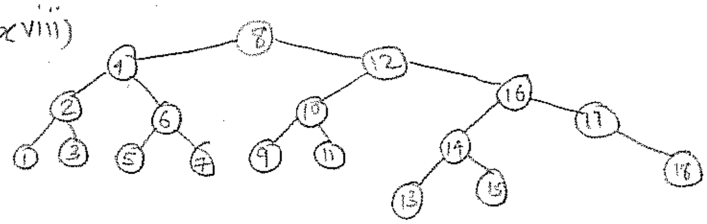
(xvi)



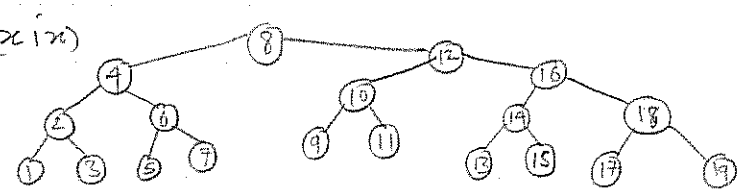
(xvii)



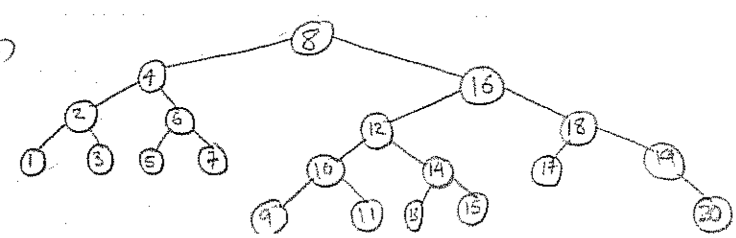
(xviii)



(xix)



(xx)



Ans 4. "Let the ht. of the "left" tree (i.e. the tree with the smaller elements) be h_1 and the right tree be h_2. Assume that $h_1 \geq h_2$; the other case is similar. First delete the maximal element from the left tree. Denote this element by r. Then use r as the new root for the right tree, and insert this root in the appropriate place on the right side of the left tree. More precisely, traverse the left tree, taking only right branches, for $h_1 - h_2$ steps. Let the node at that place be v and its parent p. The new concatenated tree will have r in place of v as the right child of p, v as the left child of r and the root of the right tree as the right child of r (the right tree remains below its root on the right side of r). It is easy to verify that this is a consistent binary search tree. This insertion may invalidate the AVL property; in which case we can use the usual remedy of a rotation"

15

ANS5 We know that AVL trees are BST where the difference in the heights of left & right subtrees at each nodes is at most one.

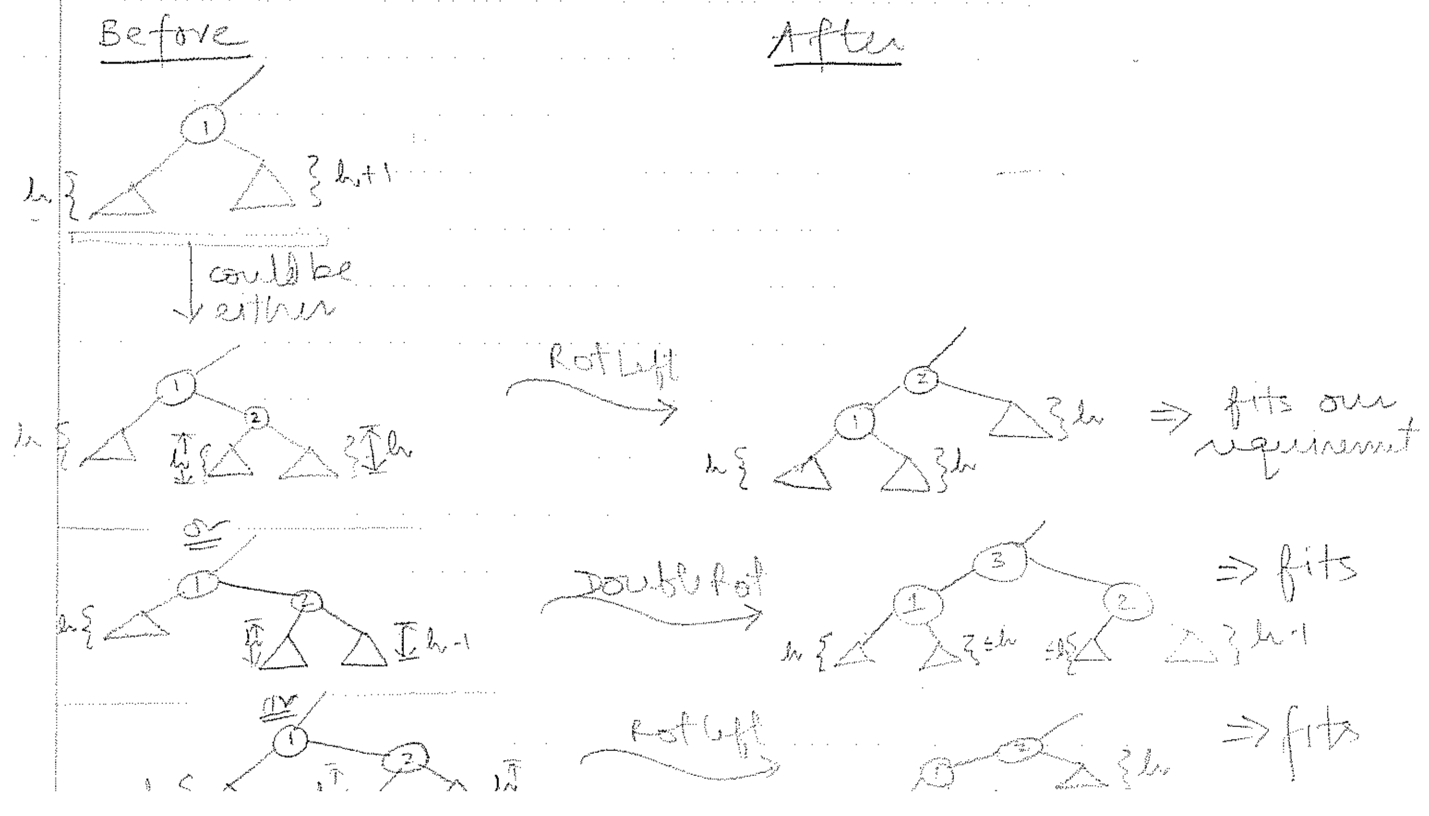
Consider restricting the set of AVL trees to be considered as only those in which the left subtrees at each node are never shorter than the right subtree: this means that the height of the left subtree will be equal to or one greater than that of the right subtree.

must be proved that this is valid

10

It is easy to see that we can allways maintain an AVL to conform to this model.

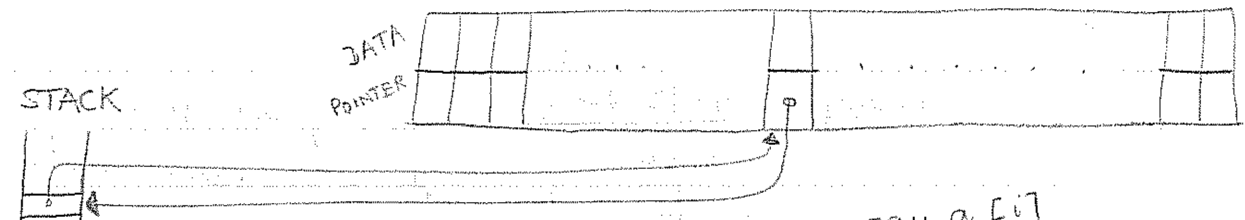
Consider the insertion of a node which results in a break of this rule. We can allways rectify this situation as depicted -

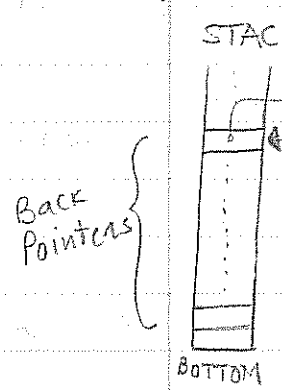


Now that we see that an AVL tree can be maintained in this constricted form, all we need is one bit with each node (to specify whether its left subtree is longer than the right subtree, or equal to it: only one bit is needed)

RECORD: DATA: datatype
ptr: POINTER
ARRAY[1..m] OF RECORDS

Q6.





Each time an entry (say a[i]) is accessed, we check to see if the pointer a[i].ptr points to a backpointer in the stack. If it doesnt ⇒ entry has never been accessed. If it does, it might still be junk so we look at wether the backpointer points to the location a[i] if it does then the entry has been written to.

The first time an entry is accessed, a[k].ptr is set to the top of the stack onto which the location a[k] is pushed.

10

~~Dear Julius~~
Happy Birthday Julie. I hope this book

~~I hope this~~
Happy Birthday!!
I hope this books gives you some insight into ~~what is to my mind the~~ most beautiful ~~field of human knowledge~~ world of mathematics

Dear Julie,

I hope this book gives you a glimpse of the beauty of mathematics.

Adnan

~~Dear~~ Julie, ~~HAPPY BIRTHDAY~~!!

I hope this book gives

HAPPY
HAPPY

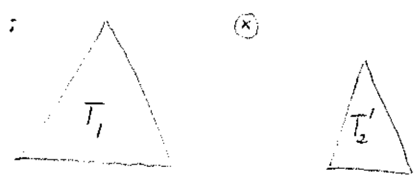
4. W.l.o.g. assume $height(T_1) \geq height(T_2)$
(if not, proceed symmetrically)

(i) delete the minimum (leftmost node) from T_2 yielding x (the minimum) and T_2'

time: $\Theta(height(T_1))$

(ii) note $\forall u$ stored in T_1: $u < x$
$\forall y$ stored in T_2': $x < y$

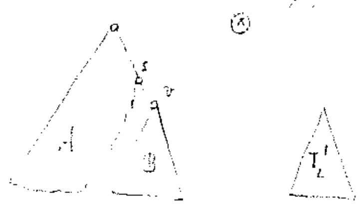
current situation:



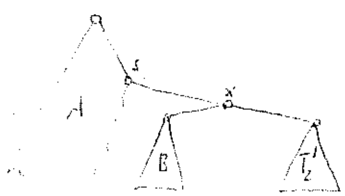
walk down rightmost branch of T_1 until node v is reached s.t. subtree rooted at v has the same height as T_2'

time: $\Theta(height(T_1) - height(T_2'))$

current situation:



replace as follows:



if necessary AVL rebalance at s

time: $\Theta(1)$

5. Method 1: store a 1 with node v iff the height of the subtree rooted at v exceeds the height of the subtree rooted at v's sibling

Method 2: store a 1 with node v iff the height of the subtree rooted at v is odd; store 0 ow.

How can you tell whether the left or right subtree of v has greater height? If both children of v store the same bit, the subtrees have the same height; otherwise the child whose bit differs from the bit of v is the root of the taller subtree

EXTRA CREDIT: We need to encode one bit per node; actually, considering method 2, we only need to encode one bit per non-leaf node. Encode a 1 at node v by swapping the children pointers; encode a 0 by leaving the pointers unchanged. How can one tell whether they were swapped? The key of the left child of v must be less than the key of v. If it really is, there was no swap, if it is not, there was a swap. (For nodes with only one child, one might also have to look at the right child.)

6. To handle array $T[1..n]$ use auxiliary arrays $PTR[1..n]$ and $STACK[1..n]$ and variable TOP. Initially, T, PTR, $STACK$ contain garbage values, $TOP = 0$.

To initialize an uninitialized $T[i]$:

TOP = TOP + 1
PTR[i] = TOP
STACK[TOP] = i

To test whether $T[i]$ has been initialized

if (PTR[i] ≥ 1 and PTR[i] ≤ TOP and STACK[PTR[i]] = i)
then yes
else no

CS 170
HWK 2
Solutions
(1)

1. $\mathcal{H}$ is c-universal iff $\forall x,y \in U,\ x \neq y$: $\left|\{h \in \mathcal{H} \mid h(x) = h(y)\}\right| \leq c \frac{|\mathcal{H}|}{t}$

In our case: $U = \{0, 1, \ldots, 5\}$ $t = 3$ $\mathcal{H} = \{h_1, h_2, h_3, h_4\}$ $|\mathcal{H}| = 4$

	0	1	2	3	4	5
$h_1(x) = x \bmod 3$	0	1	2	0	1	2
$h_2(x) = (2x+1) \bmod 3$	1	0	2	1	0	2
$h_3(x) = x^2 \bmod 3$	0	1	1	0	1	1
$h_4(x) = x \bmod 2$	0	1	0	1	0	1

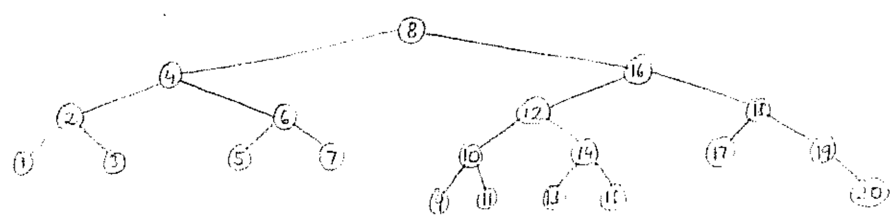
No two columns agree in all 4 places. Col's 0 and 3 as well as col's 2 and 5 agree in 3 places each. Thus

$$\max_{\substack{x,y \in U \\ x \neq y}} \left|\{h \in \mathcal{H} \mid h(x) = h(y)\}\right| = 3$$

Hence the minimal c s.t. $\left|\{h \in \mathcal{H} \mid h(x) = h(y)\}\right| \leq c \frac{|\mathcal{H}|}{t} = c\,\frac{4}{3}$

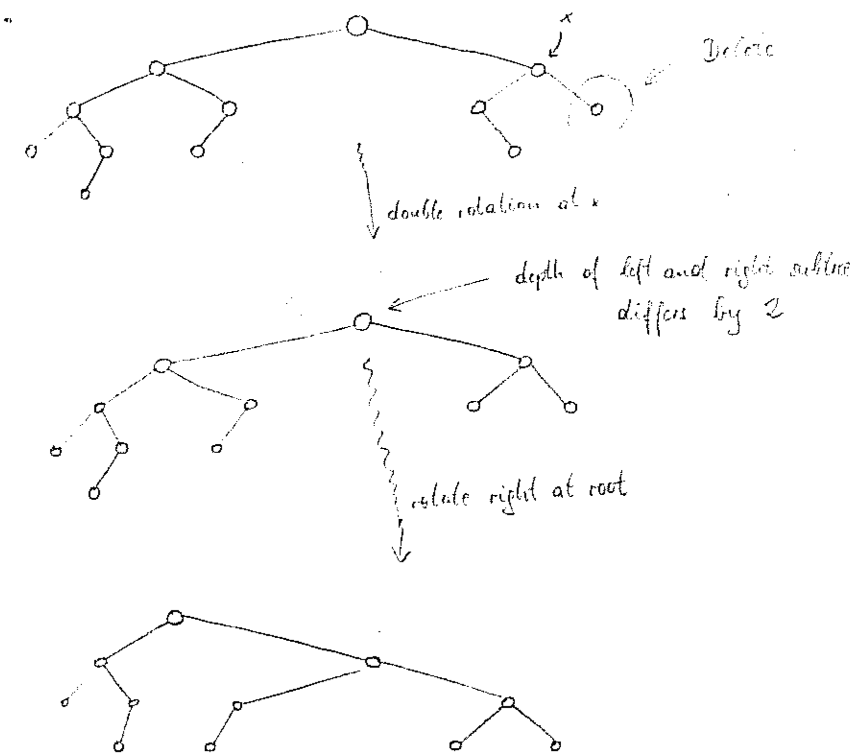
is $\frac{9}{4}$

2.



CS 170
HWK 2
Sol'ns
(2)

3.



CS 170 | Assignment 3 | February 22nd, 1990
R. Seidel | | Due: March 1st, 1990

1. Let T_1 and T_2 be two TREAPS so that all nodes in T_1 have *keys* that are smaller than the *keys* of all nodes in T_2. The operation *CONCATENATE* (T_1,T_2) returns a single TREAP that contains exactly all the items in T_1 and T_2.

   The operation *SPLIT* (T,x) achieves the opposite. It returns two TREAPS, T_1 and T_2, where T_1 contains all items in T whose *keys* are not greater than x, and T_2 contains all items in T whose *keys* are greater than x.

   Give non-recursive, top-down implementations of *CONCATENATE*() and *SPLIT*(). The running time is supposed to be $O(h)$, where h is the largest heigth of any TREAP involved in the operation.

   Give non-recursive, top-down implementations of $INSERT(x,p,T)$ and $DELETE(x,T)$, where x is a key, p is a priority, and T is a TREAP.

✓ 2. Design a data structure for the following dynamic query problem:

   The underlying universe are *items*. Each item is an ordered pair $(key, value)$, where keys are drawn from some totally ordered set K, and values are integers (positive and negative). The data structure is to store a set S of items.

   The update operations for your data structure are

   CREATE_EMPTY_STRUCTURE (S)
   INSERT $(item, S)$
   DELETE $(item, S)$

   with the usual semantics.

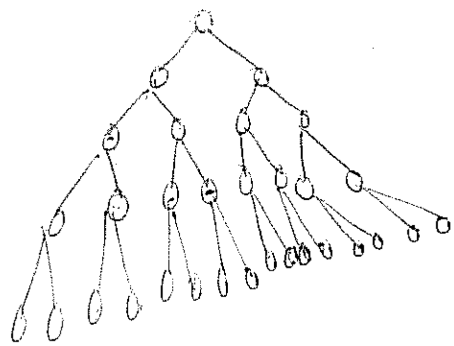
   The query operation $SUM(x,y,S)$ is supposed to return the sum of the values of all the items $(key, value)$ in S, with $x \le key \le y$.

   Your data structure is supposed to use only $O(n)$ space, where $n = |S|$. Each update and query operations should take time $O(\log n)$.

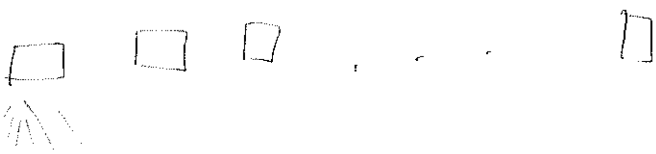
3. Suppose we have a set S of words, i.e. strings of the letters $a-z$. We want to sort S according to the usual "dictionary" order. This is the "lexicographic" order defined in class with the additional stipulation that if word α is a prefix of word β, then α precedes β in the ordering.

   Assume the sum of the lengths of all the words in S is n. Design an algorithm that sorts S in time $O(n)$.

   b.27 Note that if the maximum length of a word in S is constant, then one could "pad" the shorter words and apply radix-sort to achieve the desired $O(n)$ bound. However, your algorithm is supposed to work in $O(n)$ time even if the word sizes in S are very diverse.



10



10 10 10 10 10

5 stages

$$10 + 10\times 10 + 10\times 10\times 10 +$$

$O(n)$

$\hookrightarrow 10\, O(n/10)$

$+ 100\, O(n/100)$

$+ \cdots$

$O(n)$

confirm [illegible] come as sch[illegible] [illegible]

– tell hotel sch. to [illegible] hours till April

∴ you won't go to [illegible]

(Kampur)

– Visa (consulate + passport)

– Jamie's paper [illegible] (2pt Schum)

hand written yet

King s[illegible] ?

CS170 ASSIGNMENT 3 Adnan Aziz

39/60

Q1. (i) Non recursive top down implementations of INSERT(x, p, T) & DELETE(x, T)

We have recursive routines for insertion and deletion. It is straightforward to convert these to nonrecursive procedures using a stack.

```
INSERT(k, prio, T);     {assume NULL is a Universal Node with PRIORITY = ∞}
 label 1,2,3;
 constant stacksize=100;
 var STACKPOINTER : 1..100;
     TSTACK : array [1..stacksize] of TREEPOINTER;
     LABELSTACK : array [1..stacksize] of INTEGER;
     RETURNLABEL : integer;
 begin
    STACKPOINTER := 0;
1:     if T= NULL then set T to newnode(k, prio)
         else if k=T->key then EXIT;
           else if k<T->key then
              begin STACKPOINTER := STACKPOINTER+1;
                 if STACKPOINTER > stacksize then stackfull;
                 TSTACK[STACKPOINTER] := T;
                 LABELSTACK[STACKPOINTER] := 2;
                 T := T->lc;
                 GOTO 1;
              end;
2:  if T->lc->prio < T->prio then ROTATERIGHT(T)
       else
        begin
          STACKPOINTER := STACKPOINTER+1;
          if STACKPOINTER > stacksize then stackfull;
          TSTACK[STACKPOINTER] := T;
          LABELSTACK[STACKPOINTER] := 3;
          T := T->lc;
          GOTO 1;
          end;
```

* Not "Non-rec, top-down"

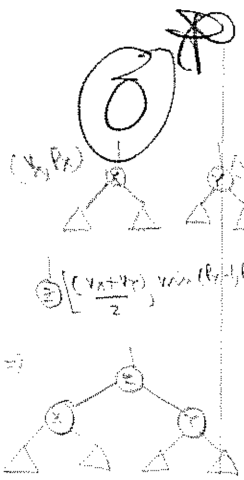
6 *

```
3:  if T->rc->prio < T->prio then ROTATELEFT(T);
    if STACKPOINTER <> 0
        then begin
                T := TSTACK [STACKPOINTER];
                RETURNLABEL := LABELSTACK [STACKPOINTER]
               STACKPOINTER := STACKPOINTER - 1;
               CASE RETURNLABEL OF
                 2 : GOTO 2 ;
                 3 : GOTO 3 ;
                 END; {case}
             end; {if-then}
    END;
```

```
(Q1 contd) DELETE(K,T);
 label 1,2,3;
 constant  STACKSIZE=100;
 var   STACKPOINTER:1..100;
       TSTACK: array [1..STACKSIZE] of TREEPOINTER;
       LABELSTACK: array [1..STACKSIZE] of integer;
       RETURNLABEL: integer;
 begin
     STACKPOINTER:=0;
 1 :  if T=NULL then exit;
       if T→key = K then
           if (T→LC=T→RC=NULL) then set T to NULL;
       else if T→LC→prio <T→RC→prio then ROTATERIGHT(
                                                else ROTATELEFT(T)

       if K<T then
          BEGIN
             STACKPOINTER:=STACKPOINTER+1;
             if STACKPOINTER>STACKSIZE then STACKFULL;
             TSTACK[STACKPOINTER]:=T;
             LABELSTACK[STACKPOINTER]:=2;
             T:=T→LC;
             GOTO 1;
          END;
 2 :      else
           BEGIN
              STACKPOINTER:=STACKPOINTER+1;
              if STACKPOINTER>STACKSIZE then STACKFULL;
              TSTACK[STACKPOINTER]:=T;
              LABELSTACK[STACKPOINTER]:=3;
              T:=T→RC;
              GOTO 1;
           END;
```

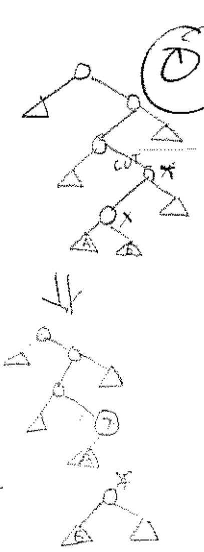
```
8: if STACKPOINTER <> 0
     THEN BEGIN
             T := TSTACK[STACKPOINTER];
             RETURNLABEL := LABELSTACK[STACKPOINTER];
             STACKPOINTER := STACKPOINTER-1;
          CASE RETURNLABEL OF
             2: GOTO 2;
             3: GOTO 3;
          END;
             END;
   END;
```

CONCATENATE (T_1, T_2)



This is now trivial. Consider the treaps T_1 & T_2 with roots X & Y as shown. Let the values and priorities of X & Y be (V_X, P_X) & (V_Y, P_Y). Then create an artificial node Z with value $((V_X+V_Y)/2, \min(P_X-1, P_Y-1))$ make T_1 & T_2 the left & right children of Z. Then the new structure is a treap. Now delete Z (using the previous routine for delete) $\Rightarrow$ Resultant treap is concatenation of T_1 and T_2. As delete routine is $O(\log n)$, the concatenate is $O(\log n) = O(h)$

SPLIT (T, x)



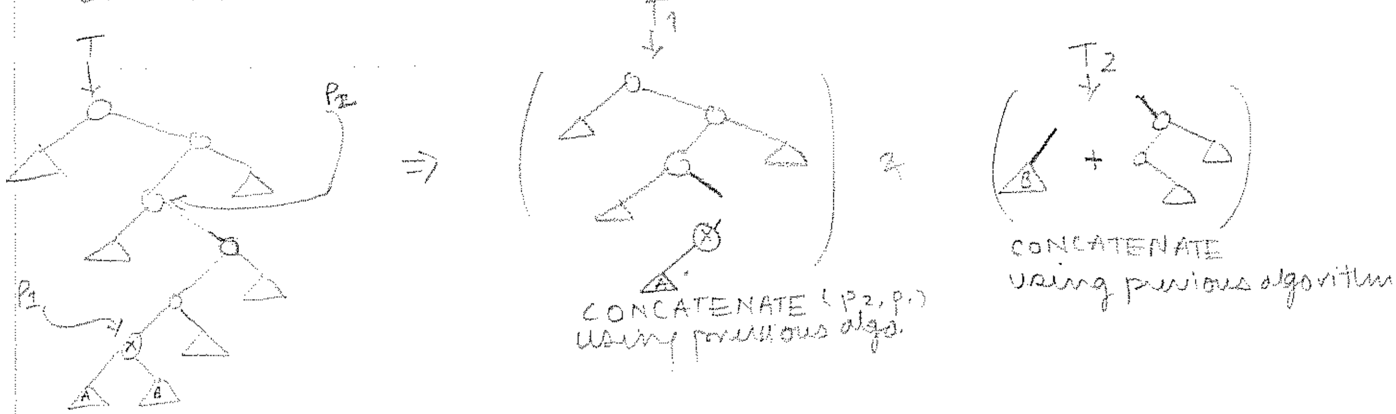
This is done by first finding the node x. March back the search path. Split the tree at the first node where you go off (G). Concatenate the tree consisting of x & its left child at this node. Concatenate the right child of x with the remainder. The result is two trees with the required property. The complexity is again $O(\log n) = O(h)$ as we move not more than 2 times the length of the maximal search path $= 2 \times h$

Pseudo Code for

```
CONCATENATE (T1, T2);
   Z := NEW(T);   {create new node}
   Z -> KEY := (T1 -> KEY + T2 -> KEY) / 2;
   Z -> PRIO := min (T1 -> PRIO - 1, T2 -> PRIO - 1);
   Z -> lc := T1;
   Z -> rc := T2;
   T := Z;
   delete (Z -> KEY, T);   {T now points to the concatenated treap}
   RETURN (T);
```

```
SPLIT (T, x);
   p1 := Find (T, x);   {return a pointer to x}
   p2 := Backtrack (T, x);  {return a pointer to first node in the
                             searchpath for x which is reached from
                             the left}
T1 := CONCATENATE (p2, p1);   {Join x and its left subtree with p2}
T2 := CONCATENATE (p1 -> rightchild, p2 -> rightchild);
   RETURN (T1, T2)
```

SPLIT in action



p_1 = FIND (T, x)
p_2 = Backtrack (T, x)

Q2 Our data structure is an AVL tree. The node structure is shown:

KEY	VALUE	LEFT SUM	RIGHT SUM
		POINTER TO L.C	POINTER TO R.C.

LeftSum is the sum of all the values of nodes in the left child
Right sum is the sum of all the values of nodes in the right child.

Clearly the data structure takes only O(n) space. Also, creating the AVL tree is just a matter of New(Ptr_Tree)

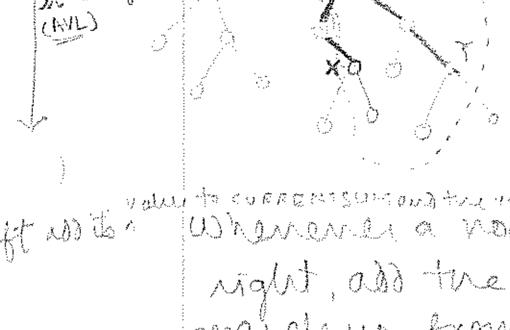
Insertion is done as we regularly do for Binary trees. However, after we insert we must update the left/right sum in the nodes on the search path to the new node. Also when we balance the tree, the nodes which take new positions must be updated. Insertion per se takes O(log n); retracing the seach path & updating the sums also is O(log n); finally balancing the tree takes O(c) (∵ only 1 or 2 rotations are needed to balance the tree ⇒ not more than 4 nodes are affected)

∴ Insertion is O(log n)

19

Deletion is done as we regularly do for Binary trees. After we delete, we must update the left/right sum in the nodes on the search path to the deleted node. Again when we balance the tree we must adjust the sum values in the nodes moved by rotations. Again, there are atmost O(log n) nodes affected; and setting the new values will take constant time, so deletion takes O(log n)

what about when we delete node ~~Left~~ w/2 children?



SUM(x,y,S) : This is done by just going to X&Y. This takes O(log n) time. Obtain the node * where the seach paths for X&Y diverge. Now add the value of the right child of X to the CURRENTSUM. Add value(Y) to CURRENTSUM. Trace the path back to * and after you reach a node from the left add its value to CURRENTSUM and the values of all successive nodes.

Whenever a node has a right child & we did not come from the right, add the Right childs sum to CURRENTTOTAL. Similarly march up from Y. Add the left child sum to CURRENT TOTAL. Add value (Y) to CURRENTSUM. March back up the search path upto *. After you reach a node from the left, add its value to currentsum & that of all successive nodes on the path. If a node has a left child and we approached from the right, add the left sum to the CURRENTSUM.

Clearly this procedure works: we add the value of every element where $X \le key \le Y$.

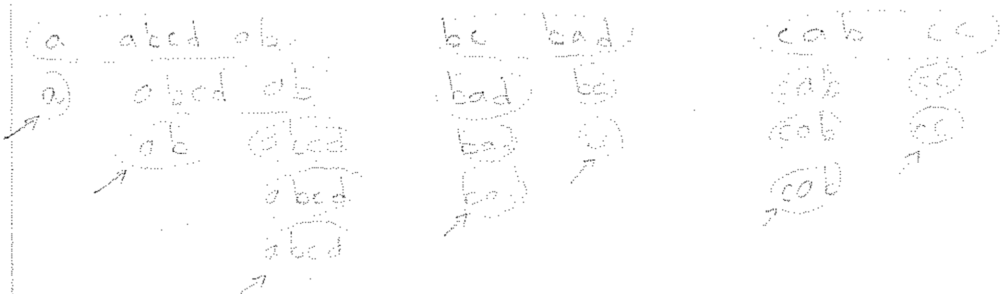
The query takes $O(\log n)$ time. For we pass through at most $O(\log n)$ nodes, and at each node we do $O(c)$ operations. Thus the procedure is $O(\log n)$

Q3 Given a set S of words, strings of the letters 'a'-'z'. We are to sort S in dictionary order in O(n) where n is the sum of the lengths of all the words in S.

Consider the algorithm where the words are sorted into 26 buckets on the basis of their first (i.e leftmost) character. Now apply the same sort on the next letter of the word if there is one. The key to this algorithm is that we look each letter only <u>ONCE</u>, and we stop looking at words after we reach the end of the word. Thus the complexity of this algorithm is O(n + w) where w = # of words. But w = O(n) (There can't be more than n words!) ∴ complexity is O(n)

The actual implementation of this scheme could be done by maintaining a pointer to the first letter of each word. Additional characters in the word may be examined by adding the appropriate offset value to the pointer. Assume a special End of Word character exists at the end of the word. When we reach it we know that we will not go further on the word so we can mark the corresponding pointer as being done. As a result once we are done with the word, we need not do any further manipulation with it.

Example: bc cab a cc abcd ab bad



Total # of operations
= 24
= 17 + 7
(# ch) (# words)

In fact if we modified our algorithm slightly be stopping the growth into lower buckets when there is only one word in the current bucket, we could do on the average substantially better. (The worst case i.e all single charac. words still needs 2n oper.

To delete the item with key a, find it in the treap and replace it by the concatenation of the two children.

```
void DELETE (a, T)
  keytype a; tnode *T;
  tnode *S, *T' = T;
    MYNULL->key = a;   (set up sentinel)
    while (T'->key ≠ a) do
       if a < T'->key then T' = T'->lchild
                      else T' = T'->rchild
    if T' = MYNULL then return;
    S = T';
    T = CONCAT(T'->lchild, T'->rchild)
    free(S);
```

To insert an item (a,p) with key a and priority p, walk down the path dictated by key a. When the priorities encountered get too large (larger than p) replace the current subtree T' by a new tree whose root is item (a,p) and whose left+right subtrees are given by SPLIT(T', a).

```
void INSERT (a, p, T)          note: this does not check for duplicates
  keytype a; int p;
  tnode *T;
    tnode *L, *R, *T' = T;
    while (p ≥ T'->prio) do
       if a ≤ T'->key then T' = T'->lchild
                      else T' = T'->rchild
    (L, R) = SPLIT(T', a)
    T' = newnode();
    T'->key = a;  T'->prio = p;  T'->lchild = L;  T'->rchild = R
```


2. Store the set S in an AVL tree (ordered w.r.t. key).

With each node of the tree store two more pieces of information:

- value ... the value of the item
- treesum ... the sum of the values of all items stored in the subtree rooted at that node

The routines create-empty-structure, insert, delete are the usual AVL tree routines augmented so as to update also "treesum" of every node on the access path.

As for node n, n->treesum can be computed in constant time from n->value and n->lchild->treesum and n->rchild->treesum, this does not affect the logarithmic running time of INSERT and DELETE (create-empty-structure is trivial).

Let LSUM(y,S) return $\sum_{\substack{n\in S\\ n\to key\le y}} n\to value$ and

RSUM(x,S) return $\sum_{\substack{n\in S\\ x\le n\to key}} n\to value$.

Then $SUM(x,y,S) = LSUM(y,S) + RSUM(x,S) - S\to treesum$

```
LSUM(y, S)                        (assume NULL->treesum = 0)
  if S = NULL then return(0)
  if S->key ≤ y then return(S->value + S->lchild->treesum + LSUM(y, S->rchild))
               else return(LSUM(y, S->lchild))
```

RSUM(x,S) can be defined symmetrically.

Since an AVL tree has logarithmic height, LSUM, RSUM and also SUM need only logarithmic time.

CS170
HWK 3

①

1. Assume C-like programming language, but with call by reference.

TREAP-node has the structure

```
struct tnode { keytype key;
               int     prio;
               tnode   *lchild, *rchild }
```

Assume MYNULL points to special tnode with MYNULL->prio = +∞
MYNULL->lchild = MYNULL->rchild = MYNULL

recursive version of Concat:

```
tnode *CONCAT1(X,Y)
  tnode *X,*Y;
    if X=Y==MYNULL then return(MYNULL)
    if X->prio < Y->prio then
        X->rchild = CONCAT1(X->rchild, Y)
        return(X)
    else
        Y->lchild = CONCAT1(X, Y->lchild)
        return(Y)
```

this gives the following iterative version:

```
tnode *CONCAT(X,Y)      tnode *X,*Y,
  tnode *T, **t;
    t = &T;
    T = MYNULL;
    while X ≠ Y    (i.e. not both MYNULL)
      if X->prio < Y->prio then
          *t = X;   t = &(X->rchild);  X = X->rchild,
      else
          *t = Y;   t = &(Y->lchild);  Y = Y->lchild,
    return(T)
```

CS 170 HWK 3

②

recursive version for SPLIT

```
SPLIT1(T,a)  .   returns pair (L,R)
                 with tnode *L,*R
  tnode *T;
  keytype a.
  if T==MYNULL then return((MYNULL,MYNULL))
  if T->key ≤ a then
        (X,Y) = SPLIT1(T->rchild, a)
        T->rchild = X;
        return((T,Y))
  else
        (X,Y) = SPLIT1(T->lchild, a)
        T->lchild = Y;
        return((X,T));
```

this gives the following iterative version:

```
SPLIT(T,a)      returns pair (L,R)
                    with tnode *R,*L
  tnode *T;
  keytype a,
    tnode *R,*L,**r,**l;
    l = &L;  r = &R;
    while T ≠ MYNULL
      if T->key ≤ a then
          *l = T;  l = &(T->rchild);  T = T->rchild;
      else
          *r = T;  r = &(T->lchild);  T = T->lchild;
  *l=*r=MYNULL,
  return(L,R);
```

S ... set of words w

$$w = (w_1, w_2, \ldots, w_{length(w)}) \qquad w_i \in \{'a', \ldots, 'z'\}$$

$$\sum_{w \in S} length(w) = n$$

IDEA: do a RADIX sort, i.e. repeated bucket sort from least significant to most significant "digit"; however, when performing the bucket sort on the i^{th} digit only involve the words in S whose length is at least i.

In order to achieve this bucket the words of S first w.r.t. their lengths.

data structures: $T['a'..'z']$ of list of words

$L[0..n]$ of list of words

1. for $k = 0$ to n do $L[k]$ = emptylist
2. for each $w \in S$ do insert w into $L[length(w)]$ } $O(n)$

3. for $k = n$ downto 1 do

3.1 for l = 'a' to 'z' do $T[l]$ = emptylist

3.2 for each $w \in L[k]$ do (insert w into $T[w_k]$) 3.2.*

3.3 for l = 'a' to 'z' do append $T[l]$ to $L[k-1]$

return($L[0]$)

Each iteration of 3 excepting step 3.2 takes constant time & hence $O(n)$ time overall.
Step 3.2* is executed for each letter of every word exactly once and takes $O(1)$ time.
But there are exactly n letters overall.

CS 170
R. Seidel

Assignment 4

March 6th, 1990
Due: March 13th, 1990

1. Let S be a sequence $x_1, \ldots, x_n$ of n real numbers and let A be a real number.
   (a) Design an algorithm to determine whether there are two members of S whose sum is exactly A. The algorithm should run in worst case time $O(n \log n)$.
   (b) Suppose now that the sequence S is given in sorted order. Design an algorithm to solve the above problem in $O(n)$ worst case time.

2. Let S be a set with n real numbers. Design an $O(n)$ time algorithm to find a number that is **not** in the set. What kind of lower bound can you prove for this problem?

3. The *weighted selection problem* is defined as follows: The input is a sequence of n distinct numbers $x_1, \ldots, x_n$, where each number x_i has a positive weight $w(x_i)$ associated with it. Let W be the sum of all the weights. The problem is to find, given a value X, $0<X<W$, the nuber x_j so that

$$\sum_{x_i<x_j} w(x_i) \le X,$$

and

$$w(x_j) + \sum_{x_i<x_j} w(x_i) > X.$$

/* choose x_j pivot on it $O(n)$
compute $\sum_{x_i<x_j} w(x_i)$ $O(n)$ $\boxed{\text{Small} \mid \text{Big}}$
if $\Sigma < X$ then solve prob. c (Big, $X - \Sigma w(x_i)$)
if $\Sigma > X$ then solve prob. c (Small, X)
Prob: Random Pivot
Det: Pivot according to the α |S| rule

(Notice that when all weights are 1, this problem becomes the regular selection problem.)

Design a randomized algorithm to solve this problem, and also a deterministic algorithm. Both algorithms should be as efficient as possible.

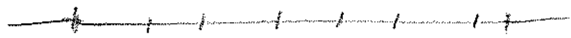
4. Draw a decision tree that corresponds to *"merge-sorting"* four keys.

(a) OK (sort – $O(n \log n)$)
then sum (binary search for element for each)

S 1 2 4

1 3 6 8 9 11

$\{x_1, x_2, \ldots x_n\}$
(i) sort $\{x_i\}$ $O(n \log n)$
(ii) compute partial sums from ... & see where you are X
$O(n \log n)$ algo



choose x_j randomly

pivot on x_j

find sum of $\sum_{x_i < x_j} w(x_i)$

if $\sum \cancel{w(x_j)} \leq X$

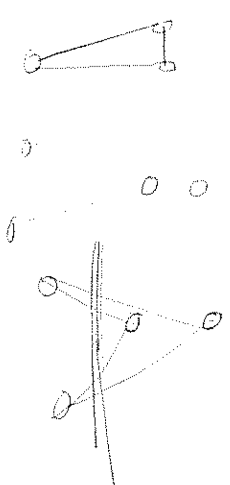
& $\sum + w(x_j) > X$ then ret x_j

else if $\sum > X$ then look for ~~x_j~~ in

an x_j in ~~[illegible]~~ Big, $\leq X - \sum$

else if $\sum \leq X$ then look for x_j in

(Small, $\sum$)



CS.10 1: 20/20 3,4: 40/40 2: 15/20 75/80

ASSGN 4

ADNAN
A212

Q1 $S = \{x_i\}_{i=1}^{n}$ $x_i \in \mathbb{R}$ $A \in \mathbb{R}$

(a) Algorithm to determine whether there are two members of S whose sum is exactly A. Worst case time $O(n \log n)$

ALGORITHM CHECK

uaranteed!

$O(n\log n)$ (i) HEAPSORT S to obtain $\hat{S}$: sorted sequence $\{\hat{x}_1, \hat{x}_2, \dots \hat{x}_n\}$

$O(n\log n)$ (ii) for each $\hat{x}_i$ perform binary search in $\hat{S}$ for $A-\hat{x}_i$

$\because$ binary search is $O(\log n)$ & we carry it out n times once for each $\hat{x}_i$)

$\therefore$ total running time is $O(n\log n)$ 20

(b) S is given in sorted order: solve in $O(n)$

consider the sum $\hat{x}_1 + \hat{x}_n = t$ ~~two~~ three possibilities arise

(i) $t < A \Rightarrow$ can never have $\hat{x}_1$ as one of the two #s

$\therefore$ look at $S' = \{\hat{x}_2 \dots \hat{x}_n\} \Leftarrow$ solve this problem; if only one element then return FALSE

(ii) $t > A \Rightarrow$ can never have $\hat{x}_n$ as one of the two #s

$\therefore$ look at $S' = \{\hat{x}_1, \dots \hat{x}_{n-1}\} \Leftarrow$ solve this problem; if only one element the return FALSE

* (iii) $t = A \Rightarrow$ return $\{\hat{x}_1, \hat{x}_n\}$

at each step we eliminate an element from consideration $\therefore$ there are at most n steps, ie the algorithm is $O(n)$

Q2. $S = \{x_1, x_2, \ldots x_n\}$ $x_i \in \mathbb{R}$

(i) To design an $O(n)$ time algorithm to find a number not in the set.

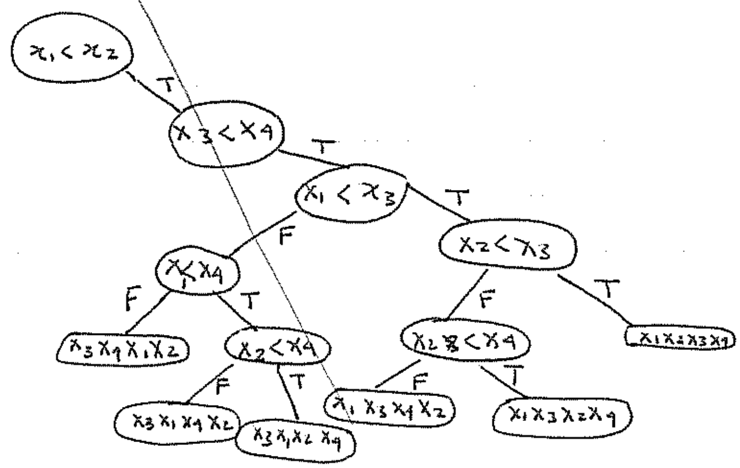
$O(n)$ ‖ Find $\min(S) \Rightarrow$ takes $n-1$ comparisons $\therefore O(n)$

Return $(\min(S) - 1) \leftarrow$ we are guaranteed that $\min(S)-1 < x_i \;\forall i \in S \;\therefore\; \min(S)-1 \notin S$

(ii) lower bound is $\Omega(n)$

Each member of S must be examined once atleast. So the problem cannot be better than $O(n)$; ~~infact~~/atleast

Q4. $\{x_1, x_2, x_3, x_4\}$



Q2 $S = \{x_1, x_2, \ldots x_n\}$ $x_i \in \mathbb{R} \;\forall i$

(i) to design an $O(n)$ time algorithm to find a number not in the set.

<u>Model of Comp assumed</u>: M/c capable of doing comparisons & subtraction of unity

<u>Algorithm</u>: (i) Find minimal element of S = M

(ii) Return (M−1)

(15)

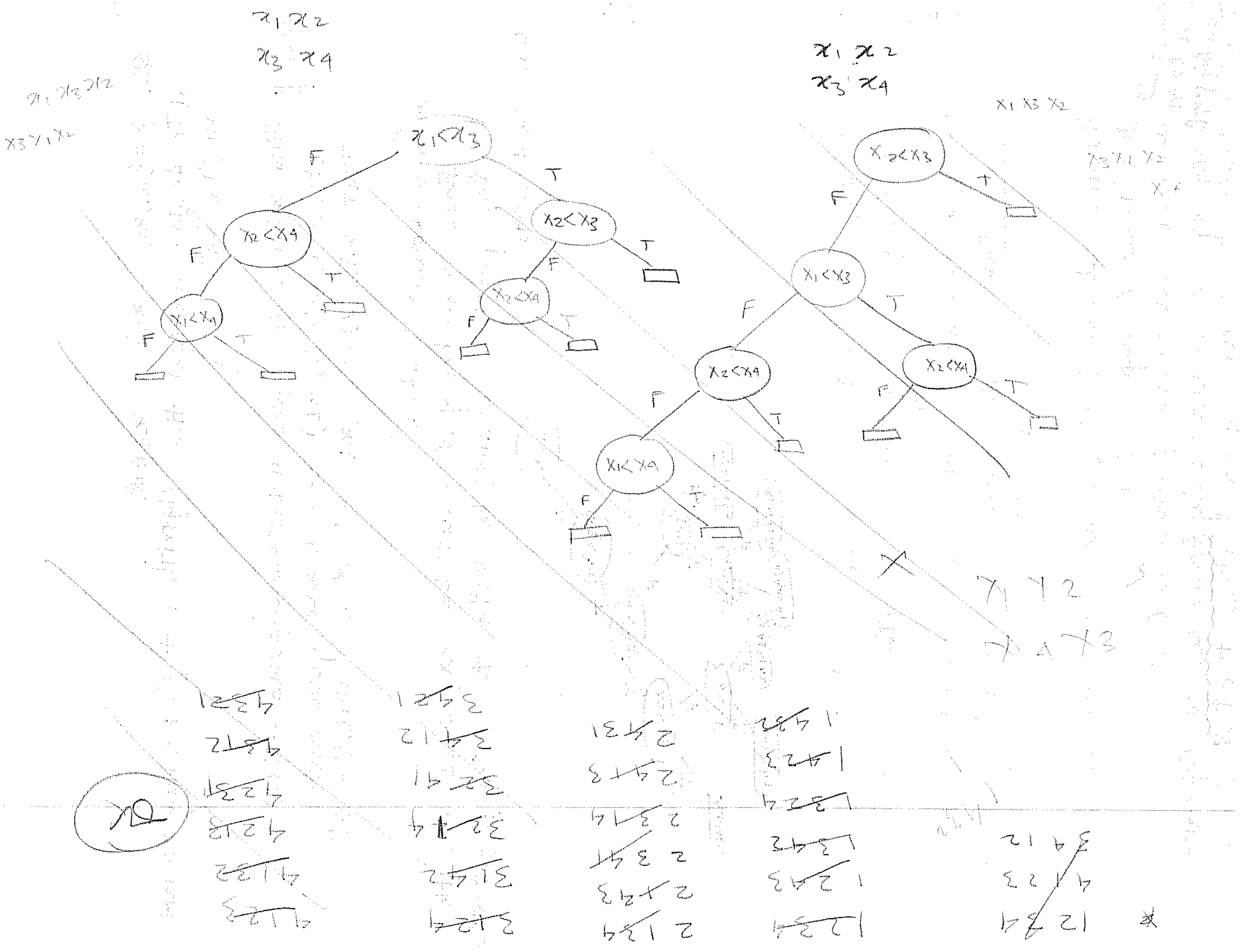
The first step is $O(n)$ (can do in $n-1$ comparisons)

The second step is $O(1)$

the running time is $O(n)$

Proof of Correctness: By <u>Zorns Lemma</u> (cf Axiom of Choice) we are guaranteed that S has

I don't think you need to appeal to such a big gun, Zorn's lemma



$x_1\ x_2$
$x_3\ x_4$
$x_1 < x_3$
$x_2 < x_4$
$x_2 < x_3$
$x_1 < x_4$
$x_2 < x_3$
$x_1 < x_3$
$x_2 < x_4$
$x_1 < x_4$
$x_2 < x_4$
$x_1\ x_3\ x_2$
$x_3\ x_1\ x_2$
1234
1243
1342
1324
1423
1432
2134
2143
2341
2314
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321

$\exists M \in S$ st $M \leq x_i$

a minimal element M. Thus ~~[illegible]~~ $\forall x_i \in S$

$\therefore (M-1) < x_i \quad \forall x_i \in S$

$\therefore (M-1) \neq x_i \quad \forall x_i \in S$

ie $(M-1) \notin S$

QED

what about "subtract 1" ?

But $\left(1+\sum x_i^2\right)$ is not $=$ (the set) can be computed without any comparisons.

Model of computation: Only comparisons allowed

(ii) Lower Bound on this algorithm: $\Omega(n)$

Proof: every element of S must be examined. Thus every algorithm must be at least $O(n)$

When we restrict ourselves to comparison based algorithms, we see that every element must go through at least one comparison. Thus there must be at least $(n+1)$ comparisons (for each element to undergo a comparison) ~~Hence the algo outlined in (i) is optimal for all comparison based algorithms.~~

Q3. WEIGHTED SELECTION PROBLEM:

Comment: we know in the special case $w(x_i) = 1 \;\; \forall x_i$ this problem reduces to that of finding the $\boxed{X}^{th}$ element in the input. Thus this problem is at least as hard as the regular selection problem. We know that the regular selection problem ~~is~~ has a lower bound of $O(n)$. Thus we can not do better than $O(n)$ for the "weighted selection" problem. The algorithms described below are $O(n)$ in expected time & $O(n)$ (deterministic). Thus they are the best [big Oh wise] possible.

3a RANDOMIZED ALGORITHM looking for $\hat{x}_j$

Find_$\hat{x}_j$(SET)

$O(1)$ Choose x_j randomly;

$O(n)$ Pivot on x_j from the sets SMALL(all $x_i < x_j$) and LARGE(all $x_i \geq x_j$)

$O(n)$ Find $\sum_{x_i < x_j} W(x_i) = SUM$

$O(1)$ if $SUM \leq X$ & ~~$\,$~~ $SUM + W(x_j) > X$ then return x_j;

$O(n/2)$ if $SUM > X$ then Find_$\hat{x}_j$(SMALL)

$O(n/2)$ if $SUM \leq X$ then $\{X := X - SUM\}$ Find_$\hat{x}_j$[(LARGE) ~~$X = X - SUM$~~]

Explanation: This algorithm randomly picks an x_j from S and pivots on x_j to form the set SMALL consisting of all $x_i < x_j$ & LARGE (all $x_i \geq x_j$) [x_j is included in large] Then the sum $\sum_{x_i < x_j} W(x_i)$ is computed. If the desired result holds, x_j is the correct element. If not, and the sum is too big, ~~xxxx~~ it suffices to examine only the set SMALL. If the sum is too small, it suffices to look at LARGE, with the new X being ~~x~~ the original $X - SUM$.

30

QED

Complexity: Let $R(n)$ be avg running time. We spend $O(n)$ time in partitioning, and $O(n)$ time in computing SUM. The new call to Find_$\hat{x}_j$ is with a smaller set. The set's size will on the avg be $n/2$. Thus we obtain the recurrance relation

$$R(n) = O(n) + R(n/2) \quad ; \text{naturally } R(n) = O(n)$$

(for $R(n) = cn + cn/2 + \cdots = 2cn = O(n)$)

Thus the algo. is $O(n)$ in expected case.

(The randomisation is in choosing x_j)

3b Deterministic Algorithm:

<u>Comment:</u> We know that the recurrance relu

$$R(n) \le cn + R(\alpha n) \; (\alpha<1) \Rightarrow R(n) = O(n)$$

n prev. algo, If we can guarantee that

$|SMALL| \le \alpha n$; $\alpha < 1$

$|LARGE| \le \alpha n$; $\alpha < 1$

then we can guarantee that the algorithm i of $O(n)$ complexity even in worst case.

But we know how to choose x_j such that we can guarantee $|SMALL|, |LARGE| \le \frac{3}{4}n$, in time ~~R(n)~~ $\underline{O(n)}$ [cf Tarjan, Blum, Floyd, Rivest, Pratt - 1972]

$\therefore$ the relation is $R(n) \le O(n) + R(3n/4)$

$\Rightarrow R(n)$ is $O(n)$

Thus to make the previous algorithm determinist & still have $O(n)$ complexity, choose x_j in the following way $\rightarrow$ (Recount of class notes follows)

$O(n)$ (i) Break S into $\lceil n/5 \rceil$ groups of at most 5 elements each

$O(n)$ (ii) Sort each group

$R(n/5)$ (iii) Let M be the set of medians from the groups. Find a the median of M

this guarantees $|SMALL| \le 3/4\,|S|$

$|LARGE| \le 3/4\,|S|$

$\therefore$ ~~$R(n) \le cn + R(n/5) + R(3n/4)$, for entire algorithm~~

~~$\Rightarrow R(n)$ is $O(n)$~~

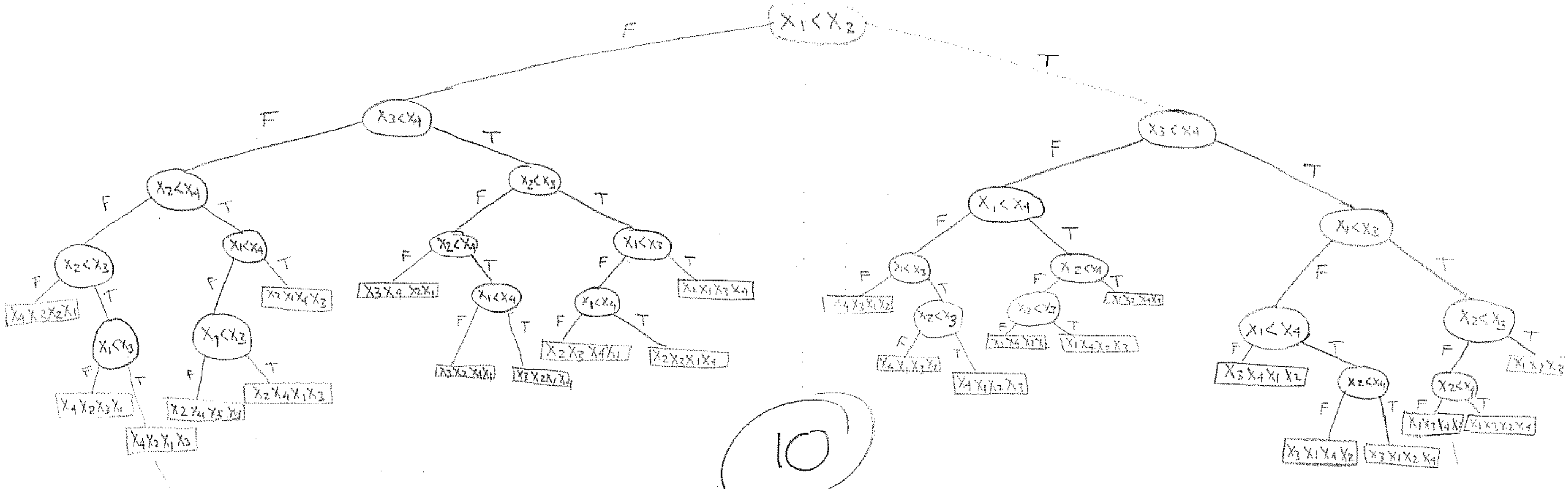
This is the only change in the previous algorithm, i.e. choose x_j in this way, rather than randomly

The complexity for obtaining x_j is governed by that of finding the median which is $R(n) \le O(n) + R(\frac{3}{4}n) + R(n/5)$

$\Rightarrow$ can obtain required x_j in $O(n)$

QED

Q4: $S = \{x_1, x_2, x_3, x_4\}$



10

CS 170

Assignment 4
Solutions

1. Assume the sequence S is stored in array $X[1..n]$.

(b) Assume X is sorted in increasing order.

```
i=1; j=n;
while i<j do
    if X[i]+X[j] < A then i=i+1
    else if X[i]+X[j] > A then j=j-1
    else return (true)   (* X[i]+X[j]=A *)
return (false)
```

This loop maintains the invariant that if two elements $X[k]$, $X[\ell]$ $(k<\ell)$ sum to A, then $i \le k < \ell \le j$.
This invariant is obviously initially true. It remains true when i is incremented, as in this case $X[i]+X[\ell] < A$ for all ℓ, $i<\ell\le j$, because $X[\ell] \le X[j]$. Likewise it remains true when j is decremented.

This algorithm takes $O(n)$ time since the loop is executed at most $n-1$ times ($j-i$ decreases by one with each iteration) and each iteration takes constant time.

(a) Assume X is not sorted.

(1.) Sort X (say, using HEAPSORT) $\quad O(n \log n)$
(2.) Apply the algorithm from (b) $\quad O(n)$

The two steps together take $O(n \log n)$ time.

2. Here is an easy solution:

(i) Determine m, the maximum of S
(ii) return$(m+1)$

(i) needs $n-1$ comparisons; (ii) needs one addition.
Thus $O(n)$ operations suffice. (Clearly $m+1$ is a number not in S, as it is greater than any number in S.)

One is tempted to say that any algorithm for solving this problem requires a linear number of comparisons. But, what algorithms are we talking about? What is the model of computation? Comparison-based algorithms do not seem appropriate, since our algorithm uses one addition. However, arithmetic operations change the situation quite drastically. Here is a way for computing a number not in S ~~that~~ which does not use a single comparison.

compute $\quad 1 + \sum_{x \in S} x^2$

Clearly this requires n additions and n multiplications, and produces a number bigger than any number of S.

Consider the following very general model of computation: Each of its primitive operations involve at most k operands, where k is a fixed number.

Any algorithm conforming to such a model requires at least $\lceil \frac{n}{k} \rceil$ primitive operations to compute a number x not in S ($|S|=n$). Why? If there are fewer operations, then at least one of the elements of S, say z, was not involved in any operation. An adversary can then make z equal to whatever x the algorithm decided to return.

3. Assume we have a procedure SELECT(S, n, k) at our disposal (as described in class) that determines the k^{th}-smallest number of an n-element set S.

(* assume all x in S are distinct *)

Weighted-Select(S, n, X) (* assume $\sum_{x \in S} w(x) \geq X \geq 0$ *)
(* $n = |S|$ *)

if $n = 1$ **then** return the only element of S $\qquad \Theta(1)$

$z = \text{SELECT}(S, n, \lfloor n/2 \rfloor)$ $\qquad \Theta(n)$

$\text{SMALL} = \{x \in S \mid x \leq z\}$ $\qquad \Theta(n)$

$\text{LARGE} = \{x \in S \mid x \geq z\}$ $\qquad \Theta(n)$

$W_{\text{SMALL}} = \sum_{x \in \text{SMALL}} w(x)$ $\qquad \Theta(n)$

if $W_{\text{SMALL}} \leq X$ **then** return(Weighted-Select(LARGE, $\lceil n/2 \rceil$, $X - W_{\text{SMALL}}$))
else return(Weighted-Select(SMALL, $\lfloor n/2 \rfloor$, X))

This algorithm works by repeatedly eliminating about half of the elements, which definitely can not be the correct answer.

SELECT was shown to run in $O(n)$ linear time in class (both, the deterministic version, and the randomized version).

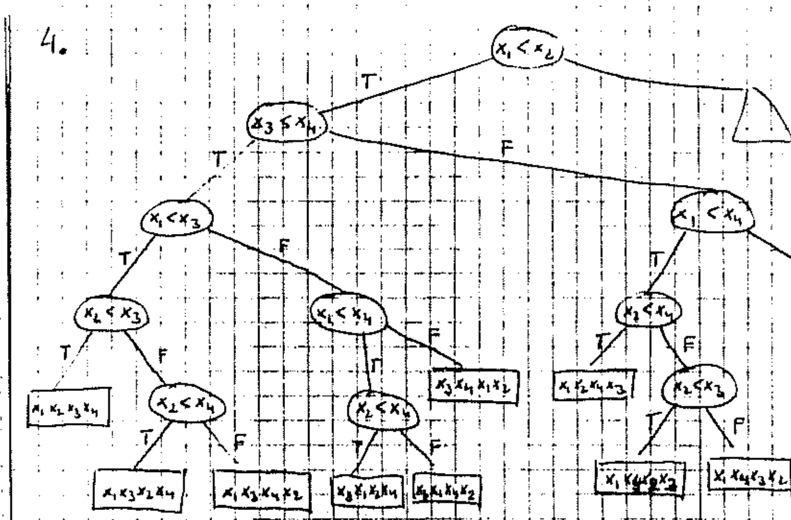
Thus $T(n)$, the running time for Weighted-Select satisfies the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(n) + T(n/2) & \text{if } n > 1 \end{cases},$$

and thus $T(n) = \Theta(n)$.

(Weighted-Select can be taken to be deterministic or randomized, depending on the version of SELECT that is used.)

4.



same tree as on the left but with x_1 & x_2 exchanged.

CS 170 **Assignment 5** **March 15th, 1990**

R. Seidel **Due: March 22nd, 1990**

1. Let M be an $n \times n$ integer matrix in which the entries of each row are in increasing order (reading left to right) and the entries in each column are in increasing order (reading top to bottom). Give an efficient algorithm that either finds the position of an integer x in M or determines that x does not appear in M. Tell how many comparisons of x with matrix entries your algorithm performs in the worst case.

2. Consider the problem of question 1. Give an adversary argument to establish a lower bound on the number of comparisons of x with matrix entries needed to solve this problem. Your lower bound should be applicable for all algorithms whose only primitive operations are comparisons between x and matrix entries.

   Can you make the lower bound match the upper bound obtained in question 1 ?

3. Let P be a simple polygon (not necessarily convex) with vertices $v_0, v_1, \ldots, v_{n-1}$. A *chord* of P is a straight line segment that connects two vertices of P and that lies entirely in the interior of P. A *triangulation* of P is a set of chords such that no two chords cross each other and the entire polygon P is divided into triangles.

   Of course, a simple polygon P might have many different triangulations. Design an algorithm that computes the number of different triangulations of a given simple polygon P.

   The input to your algorithm will be n, the number of vertices of the polygon P, and a procedure

   function *CHORD* (i, j : **integer**) : **boolean**

   that given two intgers $0 \le i < j < n$ returns **true** if and only if the line segment that connects vertices v_i and v_j is a chord of P. Assume that a call to *CHORD* takes constant time.

   Assuming that arithmetic operations on integers can be performed at constant cost per operation what is the running time of your algorithm? (It should be polynomial.)

   Hints: Use a dynamic programming approach. Note that in every triangulation of P every edge of P (in particular the edge $v_{n-1}v_0$) is part of exactly one triangle.

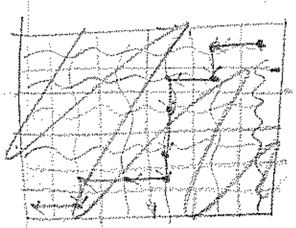
CS170 ASSGN 5 Adnan Aziz

5: 7/25

Q1 M is an n×n integer matrix with rows & column vectors in ascending order. To find an efficient algo for finding position of X or determining if X not present.

Algo: - Start from ~~upper right hand element~~ of matrix
- If X is equal to it then return (True, Position)
- If X is less than this element, we can scratch out the entire column its in as the column is sorted in ascending order & proceed to the element on the immediate left & repeat the check for X taking identical action
- If X is more than this element, we can scratch out the entire corresponding row (as its sorted in ascending order) and look at the element immediately below & repeat the check for X taking identical action
- If you are left with no element to compare with as per the one required by the above then the element is not in the matrix

This is clearly illustrated in the example:



(i) look for 8

1	2	3	4
2	7<	9<	11
13<	14	15	16
14	15	16	17

can't go anyplace → ∴ not in M!

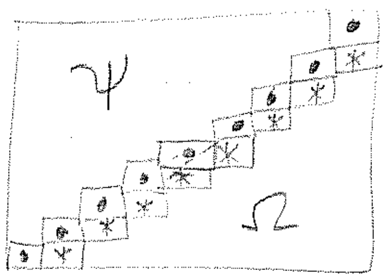
(ii) look for 59

1	3	5	7	11
2	6	9	20	26
9	13	17	22	30
10	15	21	28	31
50	56	59	70	100

At most this algo makes 2N-1 comparisons because we move either one position down or one left at each step and we will move to a terminating position at the lower left hand corner in (N)+(N-1) = 2N-1 comparison steps

25

~~This algo works because at each step we remove elements known to be impossible to be X and move towards~~

Q2. M = 

<u>Adversary Strategy</u>: (i) If X is compared with an element from Ω reply X is less than the element
(ii) If X is compared with an element from Ψ reply X is more than Ψ
(iii) If X is compared with • reply X more than •
(iv) If X is compared with * reply X less than *

25

It is necessary to compare X with <u>each</u> • & * for if not by the above strategy we keep the value of •'s & *'s from the algorithm & hence if a • or * is not examined the algo could not return a certain answer.

There are 2N-1 * & • So 2N-1 is a lower bound on the algo. Also it is best lower bound for the 1st algo actually solves it in 2N-1 proving (1) is optimal

Adnan Aziz

Q3. $P: v_0 v_1 v_2 \dots v_{n-1}$

SUM = 0

Fix edge $v_{n-1}v_0$; for $k = 1$ to $n-2$ {form $\Delta v_0 v_k v_{n-1}$
~~Compute~~ # of triangulations for $v_0 .. v_k$ & $v_k .. v_{n-1}$
if $\Delta v_0 v_k v_{n-1}$ is valid then add the product of the
of Δgulations for $v_0 .. v_k$ & $v_k .. v_{n-1}$ to SUM}

So we need to know # of triangulations in $v_0 .. v_k$ & $v_k .. v_{n-1}$ for all k. This can be done by forming the following matrix:

$T(v_0)$ $T(v_1)$ $T(v_2)$ .... $T(v_{n-1})$
$T(v_0v_1)$ $T(v_1v_2)$ .... $T(v_{n-2}v_{n-1})$
$T(v_0v_1v_2)$ $T(v_1v_2v_3)$ ..... $T(v_{n-3}v_{n-2}v_{n-1})$
....
$T(v_0 .. v_{n-2})$ $T(v_1 .. v_{n-1})$
$T(v_0 .. v_{n-1})$

$T(\cdot)$ = # of triang for given set of vertic

This won't work as base case

Each element in ~~a~~ the matrix can be computed from the previous rows in a manner exactly analogous to that for finding the final answer ie by fixing an edge, constructing triangles to each vertex & adding the products of # of triangulations for corresponding parts when the triangle is valid. (Validity of Δ from checking chord) $T(v_i) = T(v_i v_{i+1}) = 0$ by convention. For each $T(\cdot)$ we compute at the very most n multiplications; there are $\frac{n(n+1)}{2}$ $T(\cdot)$ to compute

$\Rightarrow$ Algo is $O(n^2)\,O(n) = O(n^3)$

You either need the base case $T(P)$ (where $|P| = 3$) $= 1$, or ~~$T(P)$~~ $= 1$

27

23

SOLUTIONS TO ASG 5, CS 170 Y. Emiris

1. Integer $n \times n$ matrix M, entries in increasing order reading left to right and top to bottom. Give algorithm to find position of x in M or determine x does not appear in M. Tell how many comparisons of x with matrix entries your algorithm performs in the worst case.

The idea is to examine matrix entries in such an order so that we can always eliminate an entire column or row of entries. Intuitively, since the smallest elements are at the upper left corner of M and the largest at the lower right corner, "middle" elements should be close to the diagonal that runs from the top right to the bottom left corner.

Let m_{ij} denote the entry $[i,j]$, c_k and r_k denote the k^{th} column and row respectively. The algorithm first compares x to m_{n1}.

(i) $x = m_{n1}$. Return $[n,1]$ and terminate

(ii) $x > m_{n1}$. All elements in c_1 are smaller than m_{n1}, thus we can eliminate c_1 and look for x in the rectangular matrix with columns $c_2, c_3, \dots, c_n$

(iii) $x < m_{n1}$. All elements in r_n are larger than m_{n1} $\Rightarrow$ eliminate r_n and repeat the algorithm on the rectangular matrix with rows $r_1, r_2, \dots, r_{n-1}$

Observe that in cases (ii), (iii) we are left with a rectangular matrix with fewer elements than the previous one.

In general we have a $[k \times \ell]$ matrix M', where $k, \ell \leq n$. Again we compare x to the element at the low left corner which is $m_{k(n-\ell+1)}$

(i) $x = m_{k(n-\ell+1)}$ Return $[k, n-\ell+1]$ and halt.

(ii) $x > m_{k(n-\ell+1)}$ Eliminate what remains of $c_{n-\ell+1}$ thus having to search the resulting submatrix of M' with columns $c_{n-\ell+2} \dots c_n$

(iii) $x < m_{k(n-\ell+1)}$ Eliminate the part of r_k in M' and search the submatrix of M' with rows $r_1, r_2, \dots, r_{k-1}$

In cases (ii), (iii) we're left with a rectangular matrix with fewer elements

1 (cont'd) The algorithm terminates and returns the correct answer. If x is in M it is encountered since these entries of M that are not compared to x are either too small or too large to equal x. If x is not in M the algorithm will eliminate all elements eventually exc m_{1n}; after finding that $x \neq m_{1n}$ the algorithm safely answers that x is not found and terminates

In the worst case the algorithm has to eliminate all elements of M one row or one column at a time, in the given $[k \times \ell]$ matrix M'. Think of each elimination as a step to the direction of m_{1n}, starting at m_{n1}; steps can be only vertical or horizontal corresponding respectiv to row or column elimination. There are $2n-1$ steps required $\Rightarrow$ $2n-1$ comparisons of x to M elements. Hence $2n-1$ is an upper bound on the number of comparisons needed to solve this problem.

2 Previous problem. Adversary argument for lower bound on algorithms with primitive operation the comparison between x and M entries.

Consider the following "bad" instance of the problem. For some x, let

$$M = \begin{bmatrix} & & & & x-1 \\ & & & & x+1 \\ & & & \ldots & \\ & x-1 & x+1 & & \\ x-1 & x+1 & & & \end{bmatrix}$$

The diagonal has entries $x-1$ as shown, the adjacent secondary diagonal has entries $x+1$, the upper triangle has entries $\leq x-1$ and the lower one $\geq x+1$ to satisfy the requ condition

Claim: $\forall$ algorithm that solves the problem with primitive operation the comparisons between x and the matrix entries must examine all entries on the shown diagonal and the secondary one. Thus it must make at least $2n-1$ comparisons, hence proving a tight lower bound ie. a lower bound which cannot be improved, since there exist an equal upper bound.

2(cont'd) Proof of the claim: Assume there is an entry on the diagonal which is not examined. Then this entry could equal x without violating the required row and column order of the matrix elements, in which case any algorithm would incorrectly answer that x is not in M. Hence every correct algorithm must examine all diagonal entries and similarly we can prove it must examine the entire secondary diagonal

To formulate the above argument as a formal adversary argument, we can consider the adversary strategy as follows: Whenever x is compared to a diagonal element or some entry in the upper triangle, the adversary answers $x-1$; queries about the secondary diagonal or below are answered with $x+1$. The same idea as above proves that if the adversary is not asked about some entry on the diagonal or on the secondary one, it may fill in this entry x.

∴ For this problem we have described an algorithm that perform in the worst case, $2n-1$ comparisons between x and M elements as well as a lower bound of $2n-1$ comparisons on any algorithm that correctly solves this problem relying only on such comparisons

3 Simple polygon P with vertices $v_0, v_1, \ldots v_{n-1}$. A chord connects two vertices and lies in the interior of P. A triangulation is a set of chords non-intersecting and dividing P into triangles. Design algorithm to compute number of triangulations of given P. The input to the algorithm is n, the number of vertices, and function CHORD(i,j:integer):boolean that returns true iff line (v_i, v_j) is a chord of P, for integers $0 \leq i,j < n$. A call to CHORD takes constant time. Assuming arithmetic operations take constant time, what is the running time of your algorithm? (We assume [illegible] always returns True).

As hinted, we observe that edge (v_0, v_{n-1}) belongs to exactly one triangle in any triangulation of P. let this triangle be $v_0 \hat{v}_{n-1} v_k$ for some $0 < k < n-1$. Every triangle including some edge (v_i, v_j) for $0 < i,j < k$ must have the third vertex v_ℓ such that $0 < \ell < k$ because we consider valid triangulati of simple polygon P. Similarly edges (v_i, v_j) with $k < i,j < n-1$ must have a third vertex v_ℓ : $k < \ell < n-1$

let $T[i,j]$ be the number of triangulations of simple polygon $(v_i, v_{i+1}, \ldots v_j)$ where $i \leq j$ and the edges $(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{j-1}, v_j)$ are those of P and (v_i, v_j) is a valid chord of P.
$T[0,n-1]$ is the number of triangulations of given simplex polygon P. From the above arguments the number of triangulations of P including $v_0 \hat{v}_{n-1} v_k$ is $T[0,k] * T[k,n-1]$. for different choices of k we get different triangulations (differing at least on the triangle that includes $(v_0 v_{n-1})$), thus $T[0,n-1] = \sum_{k=1}^{n-2} (T[0,k] * T[k,n-1])(\text{CHORD}(0,k) \text{ AND } \text{CHORD}(k,n-1))$

$T[i,j] = \sum_{k=i+1}^{j-1} T[i,k] * T[k,j] (\text{CHORD}(i,k) \text{ AND } \text{CHORD}(k,j))$

The crux of the problem was to derive this recursive relation. Now we can easily write the algorithm. Both versions refer to a 2-dimensional array called $T[i,j]$ where $0 \leq i,j < n$. Actually we can consider only entries for which $i \leq j$; intuitively this is because we can always take the vertices $v_i, v_{i+1}, \ldots, v_j$ of a polygon in ascending-index order. We initialize $T[i,j] = -1$, $\forall\ 0 \leq i \leq j < n$

3(cont'd) Recursive version : triang(i,j)

```
if T[i,j] ≠ -1 then return T[i,j]
else begin
   s = 0.
   for k=i+1 to j-1 do begin
      if CHORD(i,k) and CHORD(k,j) then
         s = s + triang(i,k) * triang(k,j)
   T[i,j] = s
   return T[i,j]
```

A call to triang(0,n-1) will return the final answer

Iterative version : triang(a,b)

```
for i:=1 to n do T[i,i]=0
for i:=1 to n-1 do T[i,i+1]=0
for i:=1 to n-2 do
   if CHORD(i,i+2) then T[i,i+2]=1
   else T[i,i+2]=0
for d=3 to n-1 do
   for i=0 to n-1-d do
      j=i+d
      for k=i+1 to j-1 do
         s=0
         if CHORD(i,k) AND CHORD(k,j) then
            s=s+T[i,k] * T[k,j]
return T[0,n-1]
```

all this is needed here

Both versions have to fill in more than half of the 2-dimensional array T[], spending O(n) time to fill in each entry. Hence both versions' running time = $O(n^3)$.

CS 170 | Assignment 6 | April 10th, 1990
R. Seidel | | Due: April 17th, 1990

1. The input is a connected undirected graph $G=(V,E)$, a spanning tree T of G, and a vertex v. Design an algorithm to determine whether T is a valid DFS tree of G rooted at v. In other words, determine whether T can be the output of DFS under some order of the edges starting with v. The running time of the algorithm should be $O(|V|+|E|)$.

2. Let $G=(V,E)$ be an undirected weighted graph, and let T be a shortest paths-tree rooted at a vertex v. Suppose now that all the weights in G are increased by a constant number c. Is T still the shortest-paths tree rooted at v?

3. Let $G=(V,E)$ be a directed graph in which a depth-first-search has been performed that constructed a DFS spanning forest F and assigned $prenum[v]$ and $postnum[v]$ to each vertex $v \in V$ according to the call initiation and call completion sequence of the search.
Without knowing G you are presented with an arc $e=(v,w)$ of G along with the ordered quadruple of integers

$$(prenum[v], postnum[v]; prenum[w], postnum[w]).$$

a) How can you tell whether e is a "back edge" with respect to F?

b) How can you tell whether e is a "cross edge" with respect to F?

c) Show by a counterexample that given only this information it is impossible to decide whether e is a "tree edge" or a "forward edge" with respect to F.

Now assume that without knowing G you are given a vertex $v \in V$ together with the set $N_v = \{w \mid (v,w) \in E\}$. Along with each vertex $w \in N_v$ you are also given the ordered pair of integers $(prenum[w], postnum[w])$; the pair $(prenum[v], postnum[v])$ is available as well.

d) Give an algorithm that based on only this information determines which members of N_v are children of v in F.

Remark: In all of the above problems assume that the information given to you is correct.

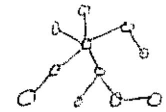
4. Give an algorithm to determine the length of the longest directed path in a directed acyclic graph G. Assume an adjacency list representation for G. Your algorithm should be as efficient as possible. What is its running time?
(The *length* of a path is meant to be the number of arcs along the path.)

↳ linear (7.38)

Problem: an instance of CNF-SAT where each variable appears twice, once in regular form, once in negated form.
show SAT-CNF can be polynomially reduced to this (or else find polynomial soln to it!)

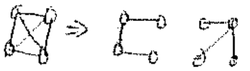
$(X + \text{expr1}) \cdot (\bar{X} + \text{expr2})$; $\overline{(X + \text{expr1})} = (\bar{X} \cdot \overline{\text{expr1}})$ no big deal.

Problem 1: $G = (V, E)$ and an integer K determine whether G contains a spanning tree T s.t. each vertex in T has degree $\leq K$



SAT
Solo

Reduction from K colorability



degree guaranteed to be less than $|V|$!
$\Sigma(\text{degrees}) = 2 \times (|V| - 1)$

G is K colorable $\Leftrightarrow$ G has spanning tree FALSE

Suet Chu

$(X_1 \vee X_2) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee X_3) \wedge \bar{X}_3$

Im dead!
Adrian

$(X_1 \vee \text{expr1}) \cdot (\bar{X}_1 \vee \text{expr2}) \cdot (\text{EXPR3})$
$= (X_1 \text{expr2} \vee \bar{X}_1 \vee \text{expr3}) \cdot (\text{EXPR3})$

one prob of n var $\Rightarrow 2$ eq $n-1$
$\therefore O(2^n) \Rightarrow$ Bad!

SAT $\rightarrow$ Little SAT

whenever variable repeated, replace with X_1'
SAT has an assignment $\Rightarrow$ new SAT (little) has assgn
But the converse is not true(!)

DARPA

$(X_1 \vee X_2) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee X_3)\ X_? (X_3 \vee X_5) \wedge (\bar{X}_5)$

Reduce 3SAT to little SAT:

$(X_1 \vee X_2 \vee X_3)(X_1 \vee X_4 \vee X_5)$

$(X \vee \text{expr1}) \wedge (X \vee \text{expr2}) = (X \vee X \wedge (\text{expr1} \vee \text{expr2}) \vee (\text{expr1}) \wedge (\text{expr2}))$

Hey, where are we?

$(X \vee \text{expr1}) \cdot (X \vee \text{expr2}) = (X \vee \text{expr3}) \cdot (?)$

11.13 Take $G = (V, E)$ & int K; G ? contains a ~~subset of K vertices which~~ clique of size K and an independent set of size K.

Take any $G' = (V', E')$ does it have a ~~K clique~~? VTX cover of size K?

construct $G = (V, E)$ where G is G' and $(G')^{\text{complement}}$

CS170 65/25 Assignment 6 Adnan Aziz

Q1 { connected graph $G = (V, E)$ / spanning tree T of G / a vertex v } INPUT

Objective: to design an algorithm to if T is a valid DFS tree rooted at v, complexity should be $O(|V| + |E|)$

Answer 1: Algorithm: (i) Do a DFS on T starting at v. Assign a labeling to nodes based on their prenumber in the DFS. (Do DFS of T based on any arbitrary order)

(ii) Do a DFS on the graph G starting at v. (The order of the DFS should be that assigned by the labeling done in (i)) Let the new tree be T'

(iii) If $T = T'$ then T is a valid DFS tree. If $T \neq T'$ then T is not a valid DFS tree

20

Running Time: (i) DFS is linear $O(|V| + |E|)$

(ii) DFS again (linear) $O(|V| + |E|)$

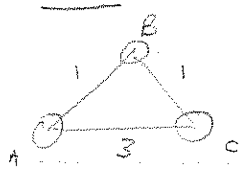
(iii) Testing for $T = T'$ is also linear ($\because$ we have labeled nodes & the edges in the adjacency list representation are in proper sequence if they are equal. If they aren't then the algo terminates right off)

$\Rightarrow O(|V| + |E|)$

Correctness: By doing the DFS on T we obtain an ordering on V. If T is a DFS then running DFS on G using the induced ordering must generate T. Thus if $T' \neq T$, T could not be a DFS tree.

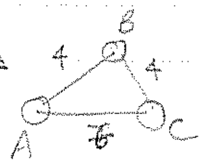
Q2. $G = (V, E)$ undirected weighted graph
T is shortest paths tree rooted at v.
All weights in G are increased by a const c
If T still shortest paths tree rooted at v?

Answer2: <u>No</u> consider the counter-example.

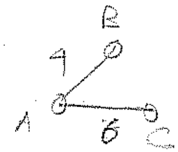


$\Rightarrow$ the shortest path tree is

add 3 to each edge $\Rightarrow$



$\Rightarrow$ the shortest path tree is



10

The trees are not the same. Hence the conjecture is <u>false</u>

Ans 3(a) $e = (v, w)$ is a "back edge" w.r.t $\mathcal{F}$
$$\Leftrightarrow \left[(\text{Prenum}(v) > \text{Prenum}(w)) \cdot (\text{Postnum}(v) < \text{Postnum}(w)) \right]$$ ← logic AND

5

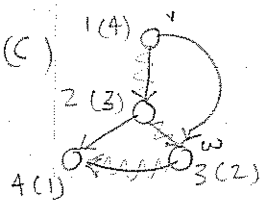
∵ for e to be a back-edge ⇔ v & w must lie in the same tree AND v must be visited after w ⇔ Postnum(v) < Postnum(w) AND Prenum(v) > Prenum(w)

(b) $e = (v, w)$ is a "cross edge" w.r.t $\mathcal{F}$
$$\Leftrightarrow \left[(\text{Prenum}(v) > \text{Prenum}(w)) \cdot (\text{Postnum}(v) > \text{Postnum}(w)) \right]$$

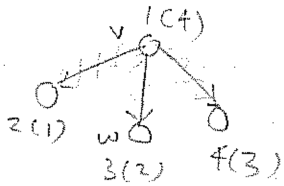
5

∵ for e to be a cross edge ⇔ v & w must lie in diff trees AND v must be visited after w ⇔ Postnum(v) > Postnum(w) AND prenum(v) > prenum(w)

(c)



(i)



(ii)

The # not in () is prenum of vertex
The # in () is postnum of vertex
~~~~ indicates tree edge

5

Clearly in (i) (v,w) is a forward edge
in (ii) (v,w) is a tree edge
But in both Prenum(v) = 1 Prenum(w) = 3
Postnum(v) = (4) Postnum(w) = (2)

So this information is insufficient to distinguish between tree edges & forward edges!

(d) We are given $N_v = \{ w \mid (v,w) \in E \}$
also $\forall w \in N_v$ we have (prenum(w), postnum(w))
and we have (prenum(v), postnum(w))

To test w for being a child of v, we need only to check to see if (v,w) is a tree edge or a forward edge!!
(This is a nec & suff cond)

(v,w) is a tree edge OR a forward edge
⇔ (v,w) is not a back edge OR cross edge
⇔ Prenum(v) < Prenum(w)

Initial appl. of logic calculus.

descendant, not child 0

∴ Our algo. is : For all $w \in N_v$ if Prenum(v) < Prenum(w)
then w is a child of v.

But actually figuring out the children is trickier.

I second the grade
~~~~

Ans 4. We use a modified topological sort:

Let $G = (V, E)$ be the DAG & K be the maximal # of edges in a path of G. Initialize COUNT to zero

The idea is that we know the longest path must start at a source and end at a sink. (For if not we could extend the path (& there would be no cycles ∵ graph is DAG)) The path goes through K edges. ALGO: We first isolate the sources. Increment COUNT by unity.

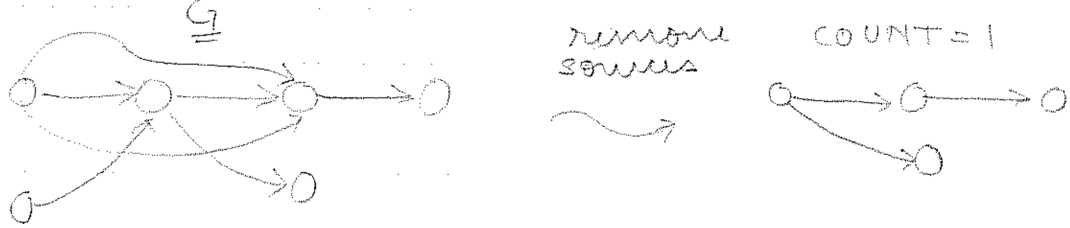
Now for all the vertices on the fringe of the sources, decrement the indegree appropriately (ie by the # of input edges originating from the sources) Now examine these vertices for zero indegree. These vertices which have zero indegree correspond to a new set of sources. Increment COUNT by 1.

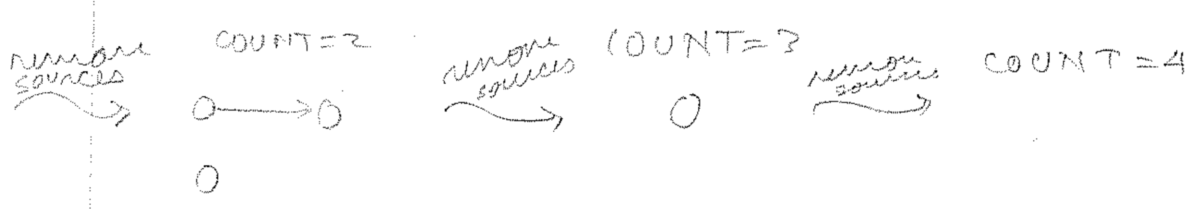
Repeat above procedure until you have exhausted all vertices Then the value of COUNT will be $K+1$

Graphically -

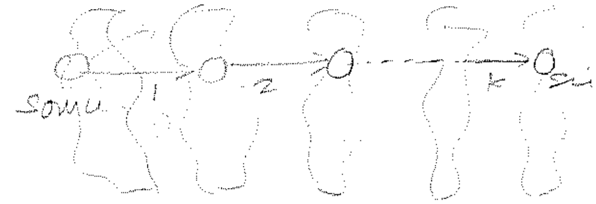
20

G





Proof of Algo's Correctness – By conjecture the longest path is of length k. (k is not known) Also as demonstrated earlier it is from a source to a sink. The algorithm partitions the graph into a sequence of sets of vertices such that the sets preserve the topological ordering. Clearly there must be exactly $k+1$ sets to exhaust all the nodes in the graph.



Proof of linearity of Algo: Initialization takes $O(|V|+|E|)$ as in topological sorting.

After that we take constant action on nodes no more than $O(|V|+|E|)$ times (corresponding to updating indegrees $\Rightarrow |E|$ & looking for nodes of in-degree 0 ($\Rightarrow |E|+|V|$))

So the running time is $O(|V|+|E|)$ i.e. linear.

QED

Solutions to Assignment 6:

1. Input: connected undirected graph $G=(V,E)$; Spanning tree T of G, and vertex v. Determine whether T is a valid DFS tree of G rooted at v.

Claim: A Spanning tree T is a DFS tree iff the graph contains no cross edges with respect to T.

Pf: $\Leftarrow$: In Manber

$\Rightarrow$: Suppose T is a spanning tree of G s.t. none of the edges of G are cross edges. We can use T to find a DFS tree of G by starting at v (on both) and every time we need to pick an edge of G to extend the DFS tree we choose an edge of T if we can. If ever we can extend the DFS tree of G with some edge where a DFS on T would be forced to back track then we have identified a cross edge w.r.t. T, which is a contradiction. $\therefore$ we can always choose the edges of T for our DFS tree.

To design an $O(|V|+|E|)$ algorithm we can use problem 3c (proven later!) to determine whether G contains cross edges w.r.t. T.

ALG: Perform DFS on T and prenumber + postnumber vertices.

For each edge in the graph $e=(u,v)$:

if (prenumber(u) < prenumber(v) and postnumber(u) < post#(v))

or (pre#(u) > pre#(v) and post#(u) > post#(v))

then output "NOT DFS." + stop.

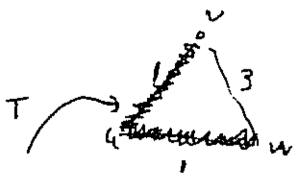
output "VALID DFS."

If we pass the test for all edges then there is no cross edge and the tree is a valid DFS tree.

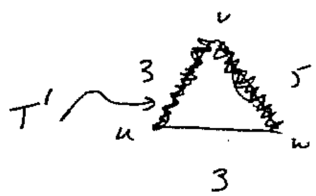
Step one takes $O(|V|)$ + step 2 takes $O(|E|)$ so we have a linear alg.

2. $G = (V, E)$ undirected + wtd. Let T be shortest paths tree rooted at v.
If we increase all wts by c is T still shortest paths tree?

No. Consider



If we increase all the weights by 2 we get



3. Given prenum(v) postnum(v) prenum(u) and postnum(u),

a) How can you tell if (u,v) is a back edge wrt DFS forest F?

(u,v) is a back edge iff v is ancestor of u

$\iff$ we see v before we see u + after the last time we visit u

$\iff$ prenum(v) < prenum(u)

<u>and</u> postnum(v) > postnum(u).

b) How can you tell if (u,v) is a cross edge wrt F?

Suppose prenum(v) < prenum(u). Then u cannot be an ancestor of v. $\therefore$ Either u is a descendant of v or (u,v) is a cross edge. But postnum(u) < postnum(v) iff u is a descendant

$\Rightarrow$ postnum(v) < postnum(u) iff (u,v) is a cross edge.

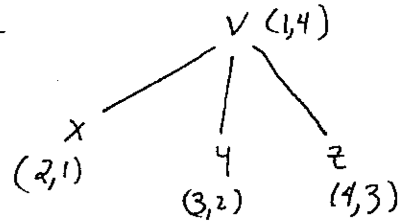
The other case is symmetrical and we find that (u,v) is a cross edge iff

(prenum(v) < prenum(u) and postnum(v) < postnum(u))

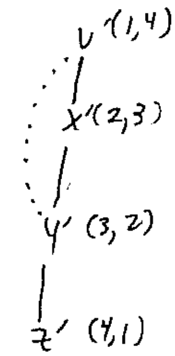
<u>or</u> (prenum(u) < prenum(v) and postnum(u) < postnum(v)).

c) Show this isn't enough info to distinguish tree + forward edge.

Consider



and



where the ordered pairs are (prenum, postnum).
v and v' have the same ordered pair (1,4), and y + y' both have (3,2).
Yet (v,y) is a tree edge and (v',y') is a forward edge.

d) Give an alg to find which neighbors of v (N_v) are children of v in T.

<u>Alg:</u> $C_v = \emptyset$.
All descendants of v will have prenum > prenum(v)
so we make a set D_v of vertices which satisfy this condition.
Examine the vertices of D_v in order, from lowest prenum to highest.
Say w is the vtx s.t. prenum(w) = prenum(v) + 1. Put w in C_v
and set presentchild = w.
For all remaining vertices u (in prenum order!)
if postnum(u) < postnum(w) go to next vtx (i.e. u is a descendan[t] of w + ∴ not a child of v)
else put u in C_v and set presentchild = u.

C_v is the set of children of v.

∴ This algorithm will work since we are identifying the descendants of v which are not ~~childs~~ descendants of children of v and the ~~vertex w~~ descendant with the lowest postnum must be a child

4. Give an alg to find the longest directed path in a dir. acyclic graph G.

We make a graph $G' = (V', E')$ by adding two nodes "source" and "sink". We add edges from source to all vertices in G and from all vtcs in G to sink. $|V'| = |V| + 2$. $|E'| = |E| + 2|V|$.

Topologically sort G'. This takes time $O(|V'| + |E'|) = O(3|V| + |E| + 2)$
$= O(|V| + |E|)$.

We can now run the single source longest path algorithm starting at the source to find the longest path from source to sink (notice source is the only vtx with indegree $= 0$.)

Single Source Longest Path: (number vtcs in top. sort order) (See Manber pg 203

```
SSLP (G, source, n)
begin
    Let z be vtx labeled n.
    if z ≠ source then
        SSLP (G - z, source, n-1)
        for all w s.t. (w,z) ∈ E do
            if w.length + 1 > z.length then
                z.length = w.length + 1
    else source.length = 0.
end.
```

This gives the longest path in G' which is 2 more than the longest path in G. The running time is $O(|V'| + |E'|) = O(|V| + |E|$

CS 170
R. Seidel

Assignment 7

April 17th, 1990
Due: April 24th, 1990

1. Show an implementation of the algorithm discussed in class to find a perfect matching in a graph with $2n$ vertices, each with degree at least n. Your algorithm should run in time $O(|V|+|E|)$.

2. Let $G=(V,E)$ be an undirected weighted graph. Prove or disprove the following statements:

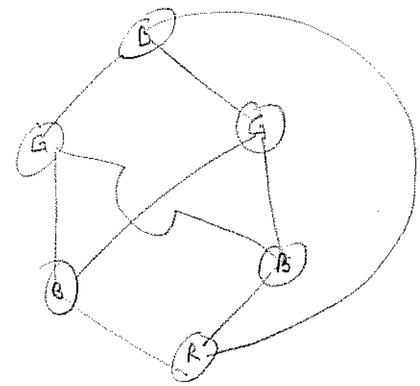
   (a) If all the edge weights of G are distinct, then the minimum cost spanning tree is unique.

   (b) If the minimum cost spanning tree of G is unique, then all the edge weights of G must be distinct.

3. The input is a directed graph $G=(V,E)$ with a positive cost $c(w)$ associated with every vertex w. Let v be a distinguished vertex of G. For a vertex $u \in V$ the cost of the directed path $v, x_1, x_2, \ldots, x_k, u$ is defined as $\sum_{1 \le i \le k} c(x_i)$. (Thus the costs of the two endpoints v and u are ignored, and so if $(v,u) \in E$, then the cost of getting from v to u is 0.)

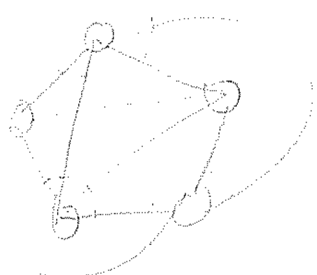
   Design an efficient algorithm to find the minimum-cost paths from v to all other vertices.

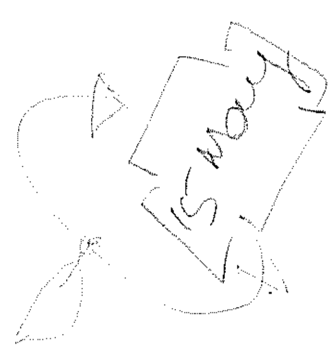
4. Let $G=(V,E)$ be a connected weighted undirected graph, and let T be a minimum cost spanning tree of G. Suppose the cost of one edge e in G is changed. Discuss the conditions under which T is no longer a minimum cost spanning tree. Design an efficient algorithm that either determines that T is still a minimum cost spanning tree or, if it is not, finds a new one. (Note that e may or may not belong to T.)



(3 colorable)

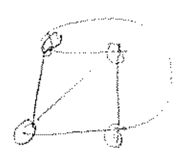
① feel good
② End pairings
③ Didn't have any real adverse effect on studies

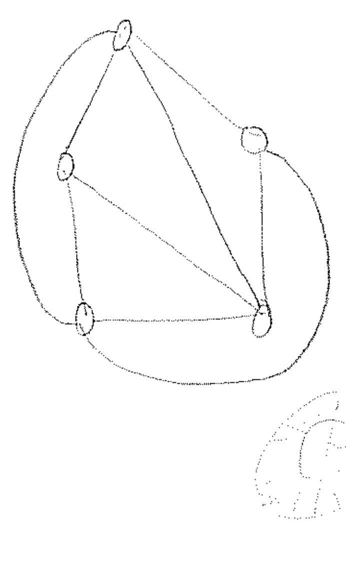


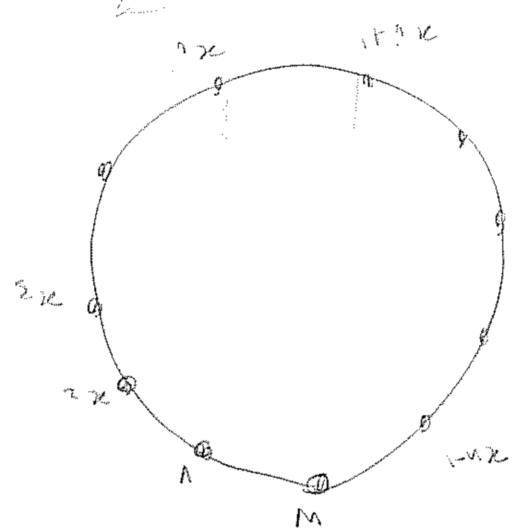


[5 way]

[3 Draw]







CS 170
HW 7
solutions

1.) Input: graph $G=(V,E)$, represented via adjacency lists, i.e. for each $v \in V$ $Neighbors[v]$ is the list of v's neighbors.
It is assumed that $|V| = 2n$, and for each $v \in V$: $|Neighbors[v]| \geq n$.

Output: perfect matching for G, i.e. an array $PARTNER[v]$
s.t. for each $v \in V$: $PARTNER[v] \neq v$ and
$PARTNER[\,PARTNER[v]\,] = v$.

Besides the array PARTNER[ ] (which initially has all elements set to "undefined"), the algorithm uses

MATCH#[v], an integer array, and
MULTIPLICITY[1..n], an integer array
} all entries initially 0.

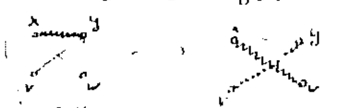
(i) (* Find a maximal matching greedily *)

```
UNMATCHED = ∅
m = 0;

for each v ∈ V do
   if MATCH#[v] = 0 then
      for each w ∈ Neighbors[v] do
         if MATCH#[w] = 0 then   MATCH#[v] = MATCH#[w] = m = m+1
                                 ~~goto ⊛~~ PARTNER[w] = v
                                 PARTNER[v] = w
                                 goto ⊛
      insert v into UNMATCHED
⊛ next
```

Running time analysis: each iteration of for loop in (i) takes $O(n)$ time, thus (i) takes $O(n^2)$ time.
each iteration of while loop in (ii) takes $O(n)$ time; there are $\leq n$ such iterations; thus (ii) also takes $O(n^2)$ time.
Entire algorithm takes $O(n^2)$ time, which is $O(|V|+|E|)$, as $|V| = 2n$, and $n^2 \leq |E| < 4n^2$.

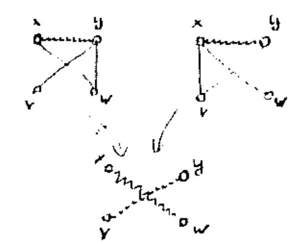
(ii) (* Deal with unmatched vertices *)

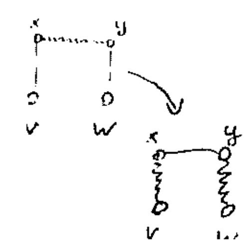
```
while UNMATCHED ≠ ∅ do
   pick and delete two vertices v and w from UNMATCHED
   for j = 1 to m do
      MULTIPLICITY[j] = 0

   for each x ∈ Neighbors[v] do
         MULTIPLICITY[ MATCH#[x] ]++

   for each x ∈ Neighbors[w] do
         MULTIPLICITY[ MATCH#[x] ]++
         if MULTIPLICITY[ MATCH#[x] ] = 3 then break

   y = PARTNER[x]

   (* now v and w are two unmatched vtcs, that have at
      least 3 edges to the matching edge {x,y}; *)

   if x ∉ Neighbors[v] then
   or y ∉ Neighbors[w]
```



```
        PARTNER[y] = v, PARTNER[v] = y;
        PARTNER[x] = w; PARTNER[w] = x;
        MATCH#[w] = MATCH#[v];  MATCH#[v] = MATCH#[y] = m = m+1;
```

```
   if x ∉ Neighbors[v] or y ∉ Neighbors[w] then
        PARTNER[y] = v; PARTNER[v] = y;
        MATCH#[v] = MATCH#[y];
        PARTNER[x] = w; PARTNER[w] = x;
        MATCH#[w] = MATCH#[x] = m = m+1;
   else
        PARTNER[v] = x; PARTNER[x] = v,
        MATCH#[v] = MATCH#[x],
        PARTNER[w] = y; PARTNER[y] = w
        MATCH#[w] = MATCH#[y] = m = m+1;
end while loop.
```



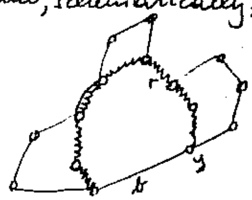


2. (a) Cla. : If the edgeweights of an undirected graph $\mathcal{G}$ are distinct, then the MST is unique.

Pf: Assume the edgeweights are distinct, but there are two different minimum spanning trees R and $\mathcal{B}$ (a "red" and a "blue" one).

Since R and $\mathcal{B}$ are different there must be an edge $b = \{x,y\}$ in $\mathcal{B}$ that is not in R. Let P be the unique path in R that connects x and y. Each vertex v on path P can be classified to be either an "x-vtx" iff ~~it can be~~ the "blue" path in $\mathcal{B}$ from x to v does not contain b, or a "y-vtx" iff the blue path in $\mathcal{B}$ from y to v does not contain b.

Since clearly x is an x-vtx and y is a y-vtx, there must be some edge r on P for which an endpoint v is an x-vtx and the other endpoint w is a y-vtx. Clearly r is not contained in $\mathcal{B}$; but the blue path in $\mathcal{B}$ that connects v and w contains b. Thus, schematically, we have the following situation.



By assumption all edgeweights are distinct, in particular the ones of r and b.

Assume $wt(r) < wt(b)$: Replacing b in tree $\mathcal{B}$ by r yields a spanning tree of smaller total weight, i.e. $\mathcal{B}$ was not minimal

Assume $wt(b) < wt(r)$: Replacing r in tree R by b yields a spanning tree of smaller total weight, i.e. R was not minimal.

CONTRADICTION

QED.

2 cont'd: (b) Let $\mathcal{G}$ be a tree, with all edges having weight 1.

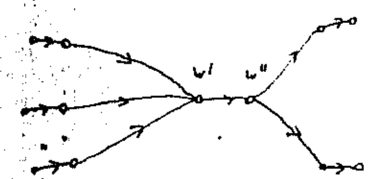
But $\mathcal{G}$'s spanning tree is unique, namely $\mathcal{G}$ itself.

QED.

3. Let $\mathcal{G} = (V,E)$ be a directed graph with positive vertex weights $c(w)$.

We will transform (or "reduce") this kind of shortest path problem to the usual single source shortest path problem on a graph $\mathcal{G}'$. Intuitively, $\mathcal{G}'$ is formed from $\mathcal{G}$ by splitting each vtx $w \in V$ into two vertices w' and w'', with a directed edge from w' to w'', and all edges ending at w now ending at w', all edges leaving w now leaving from w''.

$\mathcal{G}$



$\mathcal{G}'$

Formally, $\mathcal{G}' = (V', W')$ is defined by

$$V' = \{w', w'' \mid w \in V\}$$

$$E' = \{(w', w'') \mid w \in V\} \cup \{(v'', w') \mid (v,w) \in E\}$$

The edge costs for $\mathcal{G}'$ are given by $c'(w', w'') = c(w)$
$c'(v'', w') = 0$.

Thus the "length" of a path $v, x_1, x_2, \ldots, x_k, u$ in $\mathcal{G}$ corresponds to the usual length of the path $v'', x_1', x_1'', x_2', x_2'', \ldots, x_k', x_k'', u'$ in $\mathcal{G}'$.

Now apply the usual single source shortest path algorithm to G' with start vertex v''. For each $w \in V$, the shortest path from v to w (in G) can now be easily recovered from the shortest path from v'' to w' (in G').

G' can be constructed from G in time $\Theta(|V| + |E|)$.

G' has $\underbrace{2|V|}_{n}$ vtcs and $\underbrace{|E| + |V|}_{e}$ edges.

The single source shortest path algorithm requires time $\Theta((n+e)\log n)$,

which in this case is $\Theta\left(\left(2|V| + |E| + |V|\right)\log(2|V|)\right) =$

$$= \Theta\left((|V| + |E|)\log|V|\right).$$

4. Let e be the edge of G whose weight is changed and let T be the current minimum cost spanning tree of G.

case 1: e not in T

subcase 1.1: e's weight is increased $\longrightarrow$ T does not change

subcase 1.2: e's weight is decreased

can be done in $\Theta(|V|)$ time $\left\{ \begin{array}{l} \text{let } b \text{ be the (an) edge of largest weight on the unique} \\ \text{path in } T \text{ that joins the endpoints of } e \\ \text{if } wt(e) < wt(b) \text{ then replace } e \text{ by } b \text{ in } T \\ \qquad\qquad \text{else } T \text{ does not change} \end{array} \right.$

case 2: e is in T

subcase 2.1: e's weight is decreased $\longrightarrow$ T does not change

subcase 2.2: e's weight is ~~decreased~~ increased

can be done in $\Theta(|E|)$ time $\left\{ \begin{array}{l} \text{remove } e \text{ from } T \text{, which yields two trees } T_1 \text{ and } T_2 \\ \text{join } T_1 \text{ and } T_2 \text{ by the shortest edge that} \\ \qquad \text{connects a vtx in } T_1 \text{ with a vtx of } T_2 \end{array} \right.$

CS170 $\quad$ 60/60 Awesome! $\quad$ ASSIGNMENT #7 $\quad$ Adnan Aziz

Ans3 $\quad$ I/p $\quad G = (V, E)$ a directed graph
$c(w)$ (+)ve cost associated with every vertex w
v : a distinguished vertex of G.

Cost of $v, x_1, \dots x_K, u = \sum_{1 \le i \le K} c(x_i)$

An efficient algorithm to find the min^m cost paths from v to all other vertices:

$O(|V|+|E|)$ → (i) Transform G to G' where $G' = (V', E')$;
$V' = V$; $E' = E$ & cost is assigned to each edge $e = (x, y)$ by $w(e) = c(y)$ $\forall y \ne v$; $c(e) = 0$ for $e = (x, v)$ (all x.)

$O[(|E|+|V|)\log|V|]$ → (ii) Solve the single source shortest paths from v in G' (eg by Dijkstra's algo.); this can be done in $O(|E| + |V|\log|V|)$ too cf Fredman & Tarjan [1987] if graph dense ie $|E| = \Theta(|V|^2)$ then could do w/o heap in $O(|V|^2)$

$O(|V|)$ → (iii) The paths obtained in (ii) are min^m cost for G; the cost of the path for G is that computed for G' minus the cost of the final node.

15

Total Running time is $O[(|E| + |V|)\log|V|)]$ or $O(|V|^2)$ without heap
↳ could do in $O[(|E|) + (|V|\log|V|)]$ if we used Fredman & Tarjan [1987] algo.

Proof of correctness:

Let $p = v, x_1, x_2, \dots x_K, u$ be an optimal path from v to u as obtained from G'.

Suppose $\exists$ a different path $p' = v, x_1', x_2', \dots x_{K'}', u$ s.t. this path p' is better for G

ie $\quad c(x_1') + c(x_2') + c(x_3') + \dots c(x_K')$

p' better than $p \Rightarrow \sum_{i=1}^{K'} c(x_i') < \sum_{i=1}^{K} c(x_i)$

Now for G' cost of p' is $\sum_{i=1}^{K'} c(x_i') + c(u)$
for G' cost of p is $\sum_{i=1}^{K} c(x_i) + c(u)$

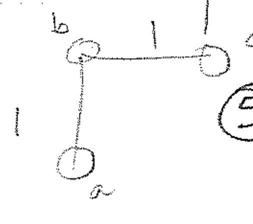
$\Rightarrow$ for G' cost of p' is $<$ cost of p
$\Rightarrow p$ is not optimal for G' $\quad$ ie CONTRADICTION

page 2

Adnan
A823

Ans 2 : $G = (V, E)$ undirected, weighted graph

(b) MIN^{M} cost spanning tree of G unique $\Rightarrow$ all edge weights distinct FALSE
for consider the counter example:

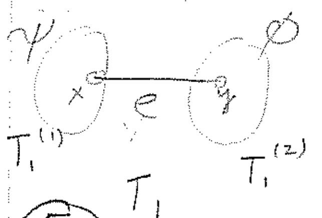
$G =$ 

(5) There is only one tree () & all edge weights are same

(a) Edge wts are unique $\Rightarrow$ MST unique TRUE

Lemma: In every subset of edges of E $\exists$ a unique minimal element (For if not the edge wts would not be unique)

Proof: Let there be two distinct MSTs T_1 & T_2 as they they are distinct there is an edge 'e' which is in T_1 & not in T_2



'e' connects the subtrees $T_1^{(1)}$ & $T_1^{(2)}$ which correspond say to the vertex subsets $\psi^{(\cdot)}$ & ϕ

(5) now in T_2 we again partition the tree into the vertex sets ψ & ϕ. As T_2 is a tree $\exists$ a unique path from every vertex in T_2 to another vertex in T_2. $\therefore$ there is a unique edge e' connecting a vertex V in ψ to a vertex W in ϕ which lies on every path connecting vertices in ψ to those in ϕ. By hypothesis $e \notin T_2$. If $c(e) < c(e')$ then we can replace e by e' in T_2 to form a Spanning Tree which has lesser cost. $e' \notin T_1$ (else T_1 no longer a tree

so if $c(e') < c(e)$ we can replace e in T_1 by e' so T_1' is still spanning tree & $c(T_1') < c(T_1)$ In either case we force a contradiction ($\because T_1$ & T_2 were distinct MSTs) The 3rd case $c(e') = c(e)$ is never possible by the conditions of uniqueness

page 3

Sivan
S8?

Q4. $G = (V, E)$ connected weighted undirected graph. T is an MST cost of an edge e in G is changed.

4 possibilities exist –

(i) $e \in T$ & e is decreased

(ii) $e \notin T$ & e is increased

(iii) $e \notin T$ & e is decreased

(iv) $e \in T$ & e is increased

In the first two cases (i) & (ii) there is no way T can change, so we needn't check anything

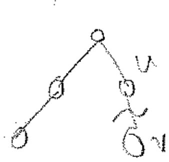
In case (iii) $e \notin T$ & e is decreased $\Rightarrow$ e may form part of a new MST. Add e to T. This creates a cycle in T. Examine this cycle and remove the edge with maximum weight. This will give us the new MST.

COMPLEXITY: Finding the cycle in the new set $T + \{e\}$ is done by DFS. Complexity is $O(|V|+|E|)$ $= O(|V|)$ $(\because |E| = |V|-1+1)$ Finding maximal weight edge in cycle is again $O(|V|)$ $\therefore$ the problem is solved in $O(|V|)$

20

In case (iv) $e \in T$ & e is increased $\Rightarrow$ e may have to be replaced. Break the tree T by removing e & forming the sets T_1 & T_2. finding a minimal cost edge from $a \in T_1$ to $b \in T_2$ will give us a new MST.

COMPLEXITY: Say (u, v) is the relevant edge. w/o loss of generality assume u is father of v. When we break the edge, T_2 will contain the descendant of v. A scan of the tree to find the min along the back edges is $O(|E|)$



On the avg we could do better than $O(|E|)$ by using some more complex data structures (minheap of edges) but in the worst case the algo will always be $O(|E|)$ as any one of $O(|E|)$ edges could be the new one in the tree (There are as many as $O(|E|)$ edges between T_1 & T_2 on removing 'e').

Ans1: To derive an implementation of algo showed in class to find a perfect matching in a graph with $2n$ vertices each with degree at least n.
Running time $O(|V| + |E|)$

Assume vertices are $v_1, v_2, \ldots v_{2n}$ & each vertex has a linked list of all the vertices its adjacent to.

(1) Construct a maximal matching M
Initially mark each vertex as unmatched
Start with vertex v_1 and match it to first (if there is one) unmatched vertex its adjacent to. (by walking down its adjacency list) and mark both as being matched. Maintain a pointer from each node to the other in the matched pair. Proceed to the next unmatched node and repeat the above procedure. Do this till all vertices have been examined.

COMPLEXITY: Clearly we do no more than $|E|$ constant operations (we look at each edge at most once!)

Now we have a maximal matching (for if not then $\exists$ v_k, v_l st. v_k, v_l are unmatched & $\exists$ an edge between them. But in the procedure we could have matched v_k & v_l when we came to one of them $\Rightarrow\Leftarrow$)

(2) We extend the matching as was explained in class. Examine vertices $v_1, v_2, \ldots$ one by one to see if there is an unmatched vertex. If there are none, we are done. If there is one there must be one more. Continue looking at vertices along the same sequence until you come to another. Let this pair of unmatched vertices be v_x & v_y. By invoking the pigeonhole principle argument, we know

~~There must be a pair attached to either end of an included edge, but not **all** ~~edges~~ pairs of vertexes have to work~~

C

there exists an edge in the matching which is connected to $\{V_x, V_y\}$ by at least 3 edges. We show how to find this edge in time $O(|V|)$

Construct an array [1..2n] which has true in A[k] if there is an edge from v_x to V_k; false otherwise. construct another similar array for V_Y. Proceed along both arrays untill there is a true in both places. This is at say: V_s. The vertex V_s *might* be on a suitable matching edge. Look at the position in the arrays corresponding to V_p, the edge that V_s is matched to. (We know V_p from v_s immediately ∵ we are maintaining pointers from vertexes to their matched vertexes) If there is a true in either position its good. (3 edges to this edge) Else continue to proceed along the arrays. repeat above. We are guaranteed in this fashion to reach our desired edge. Note that constructing the arrays takes $O(|V|)$ time; ~~traversing~~ the array elements takes a constant amount of time for each index ⇒ total is also $O(|V|)$)

C

~~$O(|V|)$ time~~ thus $O(|V|^2)$

With this edge we can extend the matching to include V_X & V_Y by removing this edge from the matching & revising pointers to V_x & V_Y & setting matched on V_X & V_Y appropriately. This takes constant time.

We repeat (2) starting where we left off i.e. from vertex V_{Y+1} onwards. The complexity for (2) is as follows:

Go through $|V|$ vertices

at each vertex we do no more than

$O(|V|)$ work (finding this edge & extending the matching)

thus (2) is $O(|V|^2)$

$\therefore$ (1) + (2) is $O(|E|) + O(|V|^2)$

but each vertex has at least n edges

$\Rightarrow |E| \geq n \times 2n$

also $|E| \leq (2n)^2$

$\therefore 2n^2 \leq |E| \leq 4n^2 \quad |E|$ is $O(n^2)$

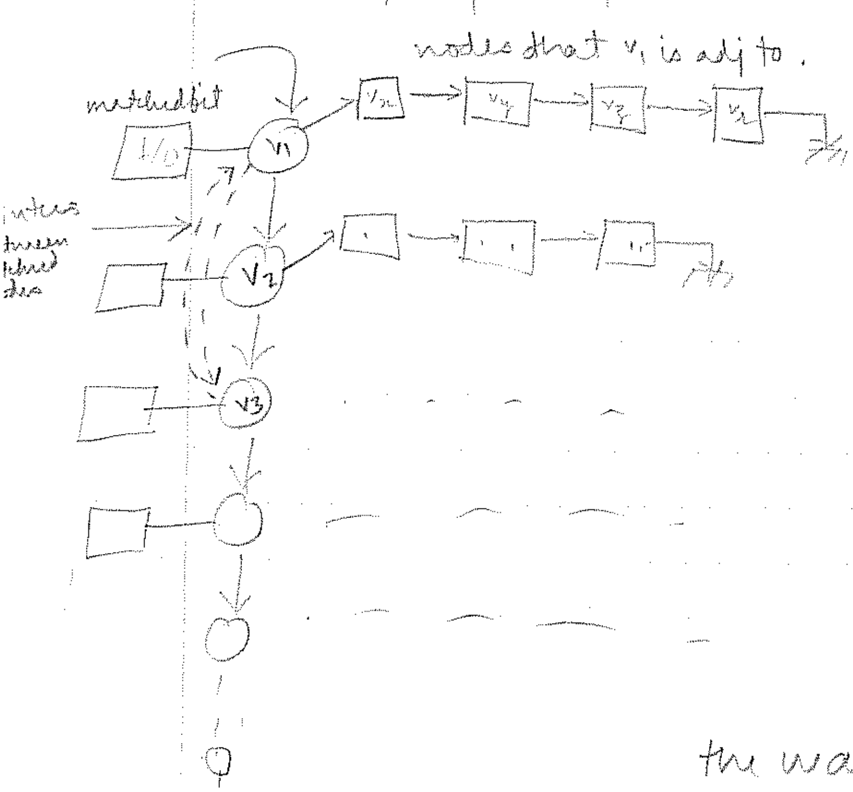
$|V|^2 = 4n^2 \;\therefore\; |V|$ is $O(n^2)$

$\therefore O(|E|) + O(|V^2|) = O(n^2)$

$= O(|V| + |E|)$

$\therefore$ algo is $O(|V| + |E|)$ 15

Graph Representation



Arrays used

	1						2n
A1	F	T	T	F	.	.	.
A2	T	F	T	.	.	.	.

A smarter way of doing this (not the way in class) is given next!

Salman Aziz

Ans 1 Graph with $2n$ vertices; each has degree $\geq n$. Find perfect matching in $O(|V|+|E|)$
$O(|V|+|E|) = O(n + 2n \times n) = O(n^2) = O((2n)^2)$
$= O(|V|^2)$

We will do this by obtaining a Hamiltonian cycle in G and then trivially extracting a perfect matching.

First some preliminary material about Hamiltonian cycles in very dense graphs.
[Adapted from Manber p. 245]
Prob: Given connected undirected graph $G=(V,E)$ with $n \geq 3$ vertices s.t. for each pair of non adjacent vertices v and w $d(v)+d(w) \geq n$ to obtain a Hamiltonian cycle (H.C) in G in $O(n^2)$ time.

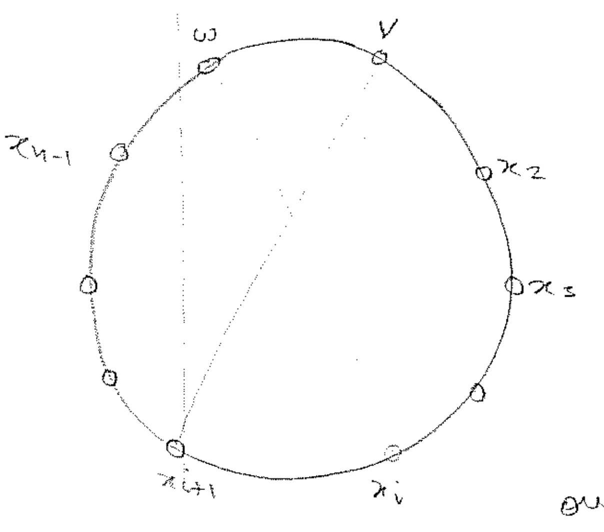
We use reversed induction argument to exhibit the existence of the H.C. This suggests the algorithm.

Base case: completely connected graph. This trivially has an H.C. (any circuit passing thru each vertex)
Induction hypothesis: We can find an H.C. in graphs satisfying the conditions of Problem with $\geq$ M edges.

(Next Page)

We now show how to find an HC in a graph with $(n-1)$ edges which satisfies the conditions stipulated in the problem. Let $G = (V,E)$ be such a graph. Take any pair of non adjacent vertices v & w in G & consider the graph G' which is the same as G except that (v,w) has been added. By the induction hypothesis, we can find an HC in G'. Let $x_1 x_2 x_3 \ldots x_n x_1$ be such a cycle in G'. (See Fig.) If (v,w) isnt included in the cycle then the same cycle is contained in G and we are done. Otherwise, without loss of generality we can assume that $v = x_1$ & $w = x_n$. We know $d(v) + d(w) \geq n$ (Condition on G) We now exhibit a new HC

Consider all the edges in G coming out of v & w. There are at least n of them. G contains $(n-2)$ other vertices. $\therefore$ there must exist at least two vertices x_i & x_{i+1} which are neighbors in the cycle s.t. there is an edge from v to x_{i+1} and an edge from w to x_i. Using these edges (v, x_{i+1}) & (w, x_i) we can construct a new H.C. which doesnt use (v,w) It is the cycle $V (= x_1)\, x_{i+1}\, x_{i+2} \ldots W (= x_n), x_i, x_{i-1}, \ldots v$



We have demonstrated the existence of an H.C. in any graph which satisfies the conditions of the problem. We now describe an algorithm that uses the existence argument outlined above to construct the HC in $O(n^2)$

page 5

Adnan Aziz

Constructing the HC:

Take the i/p graph G, find a large path (say by doing DFS from an arbitrary node until a back edge is encountered). Add edges (not in G) to complete this path to an HC. ∴ we have a larger graph G' which contains an H.C. In the very worst case (n-1) edges will have to be added. We can apply the proof iteratively, removing each of the added edges (not in G) until they are all removed at which the HC will be entirely in G. The total # of steps to replace an edge is O(n). [We look for edges from v & w to nodes of the form x_{i+1} & x_i on the cycle. This could be done by writing out the cycle (O(n)) then writing out the edges from v & w till a match is reached (O(n))] We need to remove at most n-1 edges (ie O(n) edges)

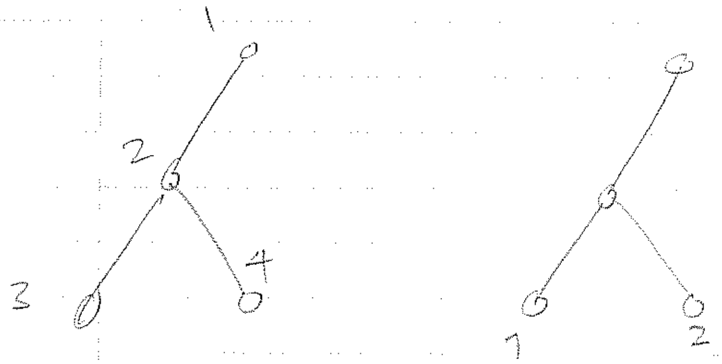
∴ algo is $O(n^2)$

Going from HC to perfect matching is trivial: Let the cycle be $x_1 x_2 \dots x_{2n} x_1$ then the matching $\{(x_1x_2), (x_3x_4), \dots (x_{2n-1}x_{2n})\}$ is perfect. It takes O(n) to traverse the H.C.

Coming to the problem given to us ie of finding a perfect matching in a graph with 2n nodes each node having degree $\geq n$, it is clear that this meets the requirements laid down earlier. Thus we can run the H.C. algo ($O(n^2)$) to get the H.C. from which we get a perfect matching O(n).

⇒ We get perf. matching in $O(n^2)$ (which

CS 170 — Algorithms — 16 April



pre # : number it the 1st time you see it
post # : number it the last time you see it

back edge : from node to ancestor
tree edge : in tree
forward edge : node to descendant

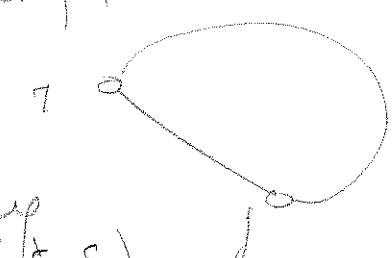
COUSIN?

$c(i,j)$ = cost for "distance between i + j"

$c(i,i) = 0$ $\quad$ $c(i,j) > 0$ $\quad$ $c(i,j) = \infty \Leftrightarrow$ can't go from i to j

TSP : whats the min cost of a round trip ~~traversal~~ through all cities

1. Naive : try all possibilities $(n!)$
2. Dynamic Programming:
   WLOG : node 1 is start & end pt
   $S \subseteq V$
   $g(1, S, i)$ = min path
   starting at 1
   ending at i, & hitting
   all the nodes in S. $(i \notin S)$



$g(1, V \setminus \{1\}, 1)$

~~Not~~ directed

she changed to notation

$$g(1, V\setminus\{1\}, 1) = \min_{j \in V\setminus\{1\}} \left(c(1,j) + g(j, V\setminus\{1,j\}, 1) \right)$$

$$i \in S \quad g(i, S, 1) = \min_{j \in S} \left(c(i,j) + g(j, S\setminus\{j\}, 1) \right)$$

Base case $g(1, \emptyset, j) = c(1,j)$

$$\sum_{k=1}^{n} k \binom{n}{k}$$

$\leftarrow O(n^2 2^{n-1})$ time

$(1+x)^n$

$O(n 2^{n-1})$ space

CS 170 **Assignment 8** **April 26th, 1990**

R. Seidel **Due: May 8th, 1990**

1. Prove that the following problem is NP-complete: Given an undirected graph G and an integer k, determine whether G contains a spanning tree T such that each vertex in T has degree at most k.

2. Prove that the following problem is NP-complete: Given an undirected graph G and an integer k, determine whether G contains a clique of size k **and** an independent set of size k.

3. Let E be a CNF expression such that each variable x appears exactly once as x and exactly once as $\bar{x}$. Either find a polynomial time algorithm to determine whether such an expression is satisfiable or prove that this problem is NP-complete.

4. Prove that the following problem is NP-complete: Given an undirected graph G and an integer $k > 3$, determine whether G is k-colorable.

Your reductions should only involve problems of this homework and problems proved NP-complete in class.

$(X \vee Y)($

$X(\bar{X} \vee Y)$

$(X \vee Y \vee Z)(\bar{X} \vee$

$\bar{X}(X \vee Y)(\bar{Y} \vee \bar{Z})Z$

1 0 0 1 0 1

N variables

⇒ can't have more than 2N clauses!

once you have forced all the X·( )

& resulting you have at least 2 things in each clause? & there are say ~~xx~~ m variables at this point?

Also You can't have more than 2

(·:) at most

$X(\bar{X} \vee Y)(\bar{Y} \vee Z)\bar{Z}$

$X(X \vee Y \vee Z)(\bar{Y} \vee \bar{Z})$

1 0 1 0 0 1

CS 170
HWK 8
Solutions

1. DEGREE-CONSTRAINED-SPANNING-TREE

Instance: undirected graph $G' = (V', E')$, integer k'

Question: Does G' have a spanning tree, where each vertex has degree $\leq k'$?

<u>Claim</u>: This problem is NP-complete.

Pf: (i) It is in NP:

non-deterministic algorithm "guesses" a subset T of edges in E' and then checks that

(i) $|T| = |V'| - 1$
(ii) T is connected
} implies T is spanning tree

(iii) every vtx $v \in V'$ has at most k' edges in T incident

in polynomial time

(ii) some NP-complete problem polynomially reduces to DEGREE-CONSTRAINED-SPANNING-TREE, namely

HAMILTONIAN PATH

Instance: undirected graph $G = (V, E)$

Question: Does there exist a hamiltonian path in G, i.e. is there an ordering of all the vertices in V, $x_1, x_2, x_3, \ldots, x_n$ s.t. for $1 \leq i < n$ $\{x_i, x_{i+1}\} \in E$?

Now note that a hamiltonian path is nothing but a spanning tree where every vertex has degree at most 2.

Thus an instance G of HAMILTONIAN PATH can be transformed into an instance (G', k') of DEGREE-CONSTRAINED-SPANNING-TREE s.t. G has a hamiltonian path iff G' has a spanning tree with each vtx degree $\leq k'$, by simply setting $G' = G$ and $k' = 2$.

Clearly this transformation only needs polynomial time.

QED.

(A proof of the NP-completeness of HAMILTONIAN PATH is appended).

From Garey & Johnson

$\bar{u}_i \in V'$. To see that this truth assignment satisfies each of the clauses $c_j \in C$, consider the three edges in E_j''. Only two of those edges can be covered by vertices from $V_j' \cap V'$, so one of them must be covered by a vertex from some V_i that belongs to V'. But that implies that the corresponding literal, either u_i or $\bar{u}_i$, from clause c_j is true under the truth assignment t, and hence clause c_j is satisfied by t. Because this holds for every $c_j \in C$, it follows that t is a satisfying truth assignment for C.

Conversely, suppose that $t: U \rightarrow \{T,F\}$ is a satisfying truth assignment for C. The corresponding vertex cover V' includes one vertex from each T_i and two vertices from each S_j. The vertex from T_i in V' is u_i if $t(u_i) = T$ and is $\bar{u}_i$ if $t(u_i) = F$. This ensures that at least one of the three edges from each set E_j'' is covered, because t satisfies each clause c_j. Therefore we need only include in V' the endpoints from S_j of the other two edges in E_j'' (which may or may not also be covered by vertices from truth-setting components), and this gives the desired vertex cover. ■

3.1.4 HAMILTONIAN CIRCUIT

In Chapter 2, we saw that the HAMILTONIAN CIRCUIT problem can be transformed to the TRAVELING SALESMAN decision problem, so the NP-completeness of the latter problem will follow immediately once HC has been proved NP-complete. At the end of the proof we note several variants of HC whose NP-completeness also follows more or less directly from that of HC.

For convenience in what follows, whenever $<v_1, v_2, \ldots, v_n>$ is a Hamiltonian circuit, we shall refer to $\{v_i, v_{i+1}\}$, $1 \leqslant i < n$, and $\{v_n, v_1\}$ as the edges "in" that circuit. Our transformation is a combination of two transformations from [Karp, 1972], also described in [Liu and Geldmacher, 1978].

Theorem 3.4 HAMILTONIAN CIRCUIT is NP-complete

Proof: It is easy to see that HC $\in$ NP, because a nondeterministic algorithm need only guess an ordering of the vertices and check in polynomial time that all the required edges belong to the edge set of the given graph.

We transform VERTEX COVER to HC. Let an arbitrary instance of VC be given by the graph $G = (V,E)$ and the positive integer $K \leqslant |V|$. We must construct a graph $G' = (V',E')$ such that G' has a Hamiltonian circuit if and only if G has a vertex cover of size K or less.

Once more our construction can be viewed in terms of components connected together by communication links. First, the graph G' has K "selector" vertices $a_1, a_2, \ldots, a_K$, which will be used to select K vertices from the vertex set V for G. Second, for each edge in E, G' contains a "cover-testing" component that will be used to ensure that at least one endpoint of that edge is among the selected K vertices. The component for

$e = \{u,v\} \in E$ is illustrated in Figure 3.4. It has 12 vertices,

$$V_e' = \{(u,e,i),(v,e,i): 1 \leqslant i \leqslant 6\}$$

and 14 edges,

$$E_e' = \{\{(u,e,i),(u,e,i+1)\},\{(v,e,i),(v,e,i+1)\}: 1 \leqslant i \leqslant 5\}$$
$$\cup \{\{(u,e,3),(v,e,1)\},\{(v,e,3),(u,e,1)\}\}$$
$$\cup \{\{(u,e,6),(v,e,4)\},\{(v,e,6),(u,e,4)\}\}$$

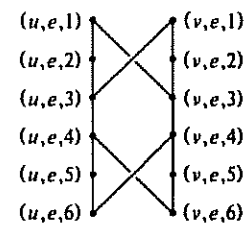


Figure 3.4 Cover-testing component for edge $e = \{u,v\}$ used in transforming VERTEX COVER to HAMILTONIAN CIRCUIT.

In the completed construction, the only vertices from this cover-testing component that will be involved in any additional edges are $(u,e,1)$, $(v,e,1)$, $(u,e,6)$, and $(v,e,6)$. This will imply, as the reader may readily verify, that any Hamiltonian circuit of G' will have to meet the edges in E_e' in exactly one of the three configurations shown in Figure 3.5. Thus, for example, if the circuit "enters" this component at $(u,e,1)$, it will have to "exit" at $(u,e,6)$ and visit either all 12 vertices in the component or just the 6 vertices (u,e,i), $1 \leqslant i \leqslant 6$.

Additional edges in our overall construction will serve to join pairs of cover-testing components or to join a cover-testing component to a selector vertex. For each vertex $v \in V$, let the edges incident on v be ordered (arbitrarily) as $e_{v[1]}, e_{v[2]}, \ldots, e_{v[deg(v)]}$, where $deg(v)$ denotes the *degree* of v in G, that is, the number of edges incident on v. All the cover-testing components corresponding to these edges (having v as endpoint) are joined together by the following connecting edges:

$$E_v' = \{\{(v,e_{v[i]},6),(v,e_{v[i+1]},1)\}: 1 \leqslant i < deg(v)\}$$

As shown in Figure 3.6, this creates a single path in G' that includes exactly those vertices (x,y,z) having $x = v$.

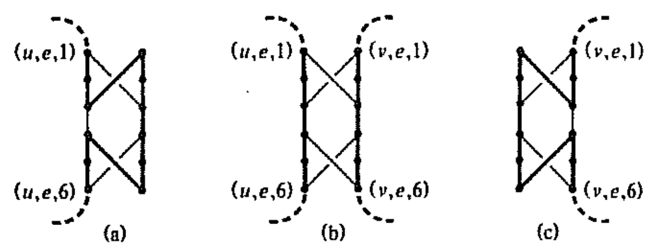


Figure 3.5 The three possible configurations of a Hamiltonian circuit within the cover-testing component for edge $e = \{u,v\}$, corresponding to the cases in which (a) u belongs to the cover but v does not, (b) both u and v belong to the cover, and (c) v belongs to the cover but u does not.

The final connecting edges in G' join the first and last vertices from each of these paths to every one of the selector vertices $a_1, a_2, \ldots, a_K$. These edges are specified as follows:

$$E'' = \{\{a_i,(v,e_{v[1]},1)\},\{a_i,(v,e_{v[deg(v)]},6)\}: 1 \leqslant i \leqslant K,\ v \in V\}$$

The completed graph $G' = (V',E')$ has

$$V' = \{a_i: 1 \leqslant i \leqslant K\} \cup (\bigcup_{e \in E} V'_e)$$

and

$$E' = (\bigcup_{e \in E} E'_e) \cup (\bigcup_{v \in V} E'_v) \cup E''$$

It is not hard to see that G' can be constructed from G and K in polynomial time.

We claim that G' has a Hamiltonian circuit if and only if G has a vertex cover of size K or less. Suppose $< v_1, v_2, \ldots, v_n >$, where $n = |V'|$, is a Hamiltonian circuit for G'. Consider any portion of this circuit that begins at a vertex in the set $\{a_1, a_2, \ldots, a_K\}$, ends at a vertex in $\{a_1, a_2, \ldots, a_K\}$, and that encounters no such vertex internally. Because of the previously mentioned restrictions on the way in which a Hamiltonian circuit can pass through a cover-testing component, this portion of the circuit must pass through a set of cover-testing components corresponding to exactly those edges from E that are incident on some one particular vertex $v \in V$. Each of the cover-testing components is traversed in one of the modes (a), (b), or (c) of Figure 3.5, and no vertex from any other cover-testing component is encountered. Thus the K vertices from $\{a_1, a_2, \ldots, a_K\}$ divide the Hamiltonian circuit into K paths, each path

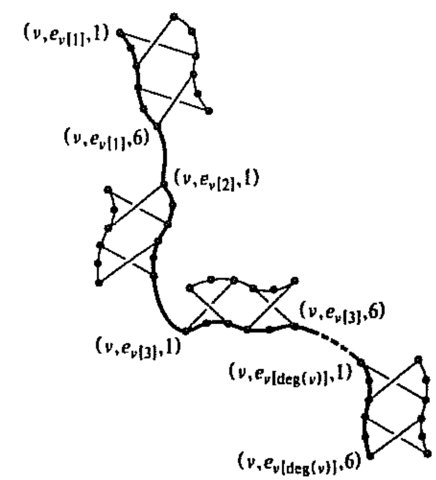


Figure 3.6 Path joining all the cover-testing components for edges from E having vertex v as an endpoint.

corresponding to a distinct vertex $v \in V$. Since the Hamiltonian circuit must include all vertices from every one of the cover-testing components, and since vertices from the cover-testing component for edge $e \in E$ can be traversed only by a path corresponding to an endpoint of e, every edge in E must have at least one endpoint among those K selected vertices. Therefore, this set of K vertices forms the desired vertex cover for G.

Conversely, suppose $V^* \subseteq V$ is a vertex cover for G with $|V^*| \leqslant K$. We can assume that $|V^*| = K$ since additional vertices from V can always be added and we will still have a vertex cover. Let the elements of V^* be labeled as $v_1, v_2, \ldots, v_K$. The following edges are chosen to be "in" the Hamiltonian circuit for G'. From the cover-testing component representing each edge $e = \{u,v\} \in E$, choose the edges specified in Figure 3.5(a), (b), or (c) depending on whether $\{u,v\} \cap V^*$ equals, respectively, $\{u\}$, $\{u,v\}$, or $\{v\}$. One of these three possibilities must hold since V^* is a vertex cover for G. Next, choose all the edges in E'_{v_i} for $1 \leqslant i \leqslant K$. Finally, choose the edges

$$\{a_i,(v_i, e_{v_i[1]},1)\},\ 1 \leqslant i \leqslant K$$

$$\{a_{i+1}, (v_i, e_{v_i[deg(v_i)]}, 6)\}, 1 \leq i < K$$

and

$$\{a_1, (v_K, e_{v_K[deg(v_K)]}, 6)\}$$

We leave to the reader the task of verifying that this set of edges actually corresponds to a Hamiltonian circuit for G'. ■.

Several variants of HAMILTONIAN CIRCUIT are also of interest. The HAMILTONIAN PATH problem is the same as HC except that we drop the requirement that the first and last vertices in the sequence be joined by an edge. HAMILTONIAN PATH BETWEEN TWO POINTS is the same as HAMILTONIAN PATH, except that two vertices u and v are specified as part of each instance, and we are asked whether G contains a Hamiltonian path beginning with u and ending with v. Both of these problems can be proved NP-complete using the following simple modification of the transformation just used for HC. We simply modify the graph G' obtained at the end of the construction as follows: add three new vertices, a_0, a_{K+1}, and a_{K+2}, add the two edges $\{a_0, a_1\}$ and $\{a_{K+1}, a_{K+2}\}$, and replace each edge of the form $\{a_1, (v, e_{v[deg(v)]}, 6)\}$ by $\{a_{K+1}, (v, e_{v[deg(v)]}, 6)\}$. The two specified vertices for the latter variation of HC are a_0 and a_{K+2}.

All three Hamiltonian problems mentioned so far also remain NP-complete if we replace the undirected graph G by a directed graph and replace the undirected Hamiltonian circuit or path by a directed Hamiltonian circuit or path. Recall that a directed graph $G = (V, A)$ consists of a vertex set V and a set of *ordered* pairs of vertices called *arcs*. A Hamiltonian path in a directed graph $G = (V, A)$ is an ordering of V as $<v_1, v_2, \ldots, v_n>$, where $n = |V|$, such that $(v_i, v_{i+1}) \in A$ for $1 \leq i < n$. A Hamiltonian circuit has the additional requirement that $(v_n, v_1) \in A$. Each of the three undirected Hamiltonian problems can be transformed to its directed counterpart simply by replacing each edge $\{u, v\}$ in the given undirected graph by the two arcs (u, v) and (v, u). In essence, the undirected versions are merely special cases of their directed counterparts.

4. k-COLORING

Instance: graph $G' = (V', E')$
integer $k > 3$

Question: Can G' be k-colored ?

<u>Claim</u>: k-COLORING is NP-complete

Pf: (i) k-COLORING is in NP:
"guess" a color for each vertex $v \in V$
and check for each edge $\{v, w\} \in E$ that
v and w have different colors

(ii) some NP-complete problem polynomially reduces to k-COLORING, namely

3-COLORING

Instance: graph $G = (V, E)$

Question: Can G be 3-colored?

Given an instance G of 3-COLORING one has to produce an instance (G', k) of k-COLORING s.t. G is 3-colorable iff G' is k-colorable.

given G let G' be G plus one new vtx, which is connected by an edge to every other vtx

let k' be 4

Clearly G is 3-colorable iff G' is 4-colorable.
Obviously G' and k' can be obtained from G in polynomial ti[me]

CS 170
HWK 8
Solutions

- CLIQUE-INDEP-SET

  Instance: undirected graph $G'=(V',E')$
  integer k'

  Question: Does G' have a clique of size k' and an independent set of size k', i.e. does there exist $C \subset V'$ and $I \subset V'$ s.t.

  $|C| = |I| = k'$ and

  $x,y \in C \implies \{x,y\} \in E$

  $x,y \in I \implies \{x,y\} \notin E$?

<u>Claim</u>: CLIQUE-INDEP-SET is NP-complete.

Pf: (i) It is in NP: a non-deterministic algorithm could "guess" sets C and I, and then check that for each $x,y \in C$, $\{x,y\}$ is in E and that for each $x,y \in I$, $\{x,y\}$ is not in E; all this in polynomial time.

(ii) some NP-complete problem polynomially reduces to CLIQUE-INDEP-SET, namely

CLIQUE

Instance: undirected graph $G=(V,E)$
integer k

Question: Does G have a clique of size k?

Need to show how to transform an instance (G,k) of CLIQUE into an instance (G',k') of CLIQUE-INDEP-SET, s.t. G has a clique of size k iff G' has a clique of size k' and an independent set of size k'.

Let $G=(V,E)$. Let M be a set of cardinality k, s.t. $M \cap V = \emptyset$.

Let $G' = (V', E')$ with $V' = V \cup M$ and $E' = E$.
Let $k' = k$

Clearly G' has an independent set of size $k'=k$, namely M.

Clearly G' has a k'-clique iff G has a k-clique.

Thus G' has a k'-clique and an independent set of size k' iff G has a k-clique.

Obviously G' and k' can be obtained from G and k in polynomial time.

QED.

3. Let E be a boolean formula in CNF s.t. each variable x appears exactly once as x and exactly once as $\bar{x}$.

<u>Claim</u>: Satisfiability of E can be decided deterministically in polynomial time.

Pf: Assume n variables $x_1, \dots, x_n$ appear in E and $E = C_1 \wedge C_2 \wedge \dots \wedge C_k$. Clearly $k \leq 2n$.

REDUCTION ALGORITHM

For $i = 1$ to n do
  if x_i still appears in the formula E then
    case 1: x_i appears as single literal clause and
            $\bar{x}_i$ appears as single literal clause
              return (E is <u>not</u> satisfiable)
    case 2: x_i (or $\bar{x}_i$) appears as single literal clause C, but
            $\bar{x}_i$ (or x_i) appears in a multi-literal clause C'
              set x_i (or $\bar{x}_i$) to "true"
              remove C from E and remove $\bar{x}_i$ (or x_i) from C'
    case 3: x_i and $\bar{x}_i$ appear in the same clause C
              set x_i to "true" and remove C from E
    case 4: x_i and $\bar{x}_i$ appear in two different clauses, say C and C'
              replace C and C' in E by the new clause
              $C \vee C'$ with x_i and $\bar{x}_i$ removed
end for loop
return (E is satisfiable)

Clearly the reduction alg works in polynomial time.

For proof of correctness the only interesting step to consider is case 4 which replaces

$$C' = (\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_{\ell'} \vee x_i) \text{ and}$$
$$C = (\beta_1 \vee \beta_2 \vee \dots \vee \beta_\ell \vee \bar{x}_i)$$

by

$$\bar{C} = \alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_{\ell'} \vee \beta_1 \vee \beta_2 \vee \dots \vee \beta_\ell$$

But now note that if for some truth assignment of the variables (except for x_i) the clause $\bar{C}$ evaluates to "true", then x_i can be assigned a value s.t. $C' \wedge C$ evaluates to true.

On the other hand, if for some truth assignment the clause $\bar{C}$ evaluates to "false" (i.e. all literals α_j and β_h evaluate to "false"), then $C' \wedge C$ evaluates to "false" no matter what truth value is assigned to x_i.

QED.