

The CS/DS Undergraduate Student Board is recruiting from all grade levels!

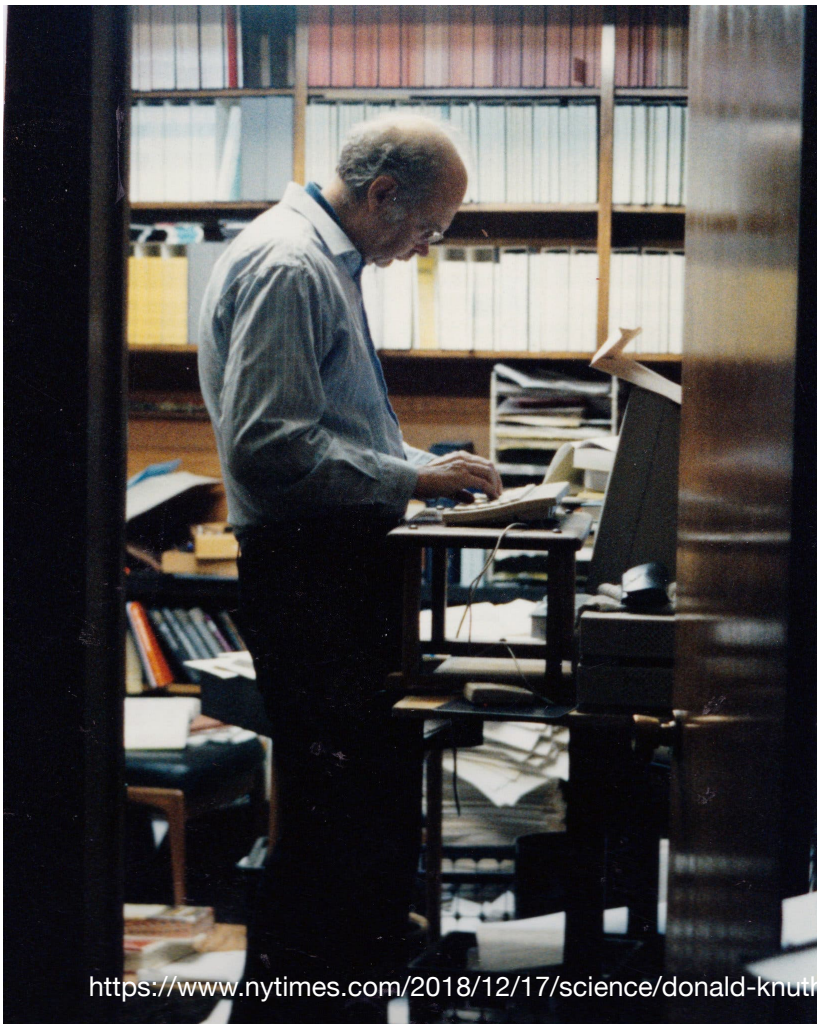
Apply at purdueusb.com/apply by January 31st.

Sequential Optimization





<https://www.quantamagazine.org/computer-scientist-donald-knuth-cant-stop-telling-stories-20200416/>



<https://www.nytimes.com/2018/12/17/science/donald-knuth-computers-algorithms-programming.html>

COMMENT * VALID 00016 PAGES

C REC PAGE DESCRIPTION

C00001 00001

C00004 00002 Preliminary preliminary description of TEX

C00007 00003 In order to explain TEX more fully, I will alternate between very low-level

C00013 00004 1 %Example TEX input related to Seminumerical Algorithms using ACPdefs

C00028 00005 The first thing that must be emphasized about this example is that it is written

C00034 00006 It is time to explain TEX's mechanism for stretching and shrinking.

C00043 00007 Now let's look at more of the example. Lines 21 and 32, etc., are blank lines

C00049 00008 Line 46 begins a "boxinsert", one of the important features needed in page layout.

C00054 00009 The ``big(){...}`` in lines 113, 143, etc. and the ``big/{...}`` in line 152

C00058 00010 The next step in understanding TEX is to go back to the low level and see

C00066 00011 Some tokens are not defined as macros but stand for actions to be carried out.

C00070 00012 At the outermost level, TEX has implicitly preceded the input by "TEXpublish{"

C00078 00013 Now let's consider the page-building routine "nextpage" this gives us a chance

C00086 00014 The nextparagraph routine assembles lines of width hsize, pausing to

C00094 00015 Besides using the permissible breaks, TEX will try to hyphenate words.

C00102 00016 To conclude this memo, I should explain how TEX is going to work on

C00103 ENDMK

C*;

Preliminary preliminary description of TEX

D Knuth, May 13, 1977

In this memo I will try to explain the TEX system for preparing publishable documents and how it is proposed to implement the system. Even though I don't understand TEX very well myself yet, I think the best way for me to get started is by trying to explain it to somebody else.

TEX is for technical text. Insiders pronounce the X as a Greek Chi (cf. the Scottish `ch' sound in `Loch Ness') since the word `technical' stems from a Greek root meaning art as well as technology. I am preparing the system primarily for use in publishing my series The Art of Computer Programming--the initial system will be tuned for my books, but it will not be difficult to extend it for other purposes if anybody wants to do so.

The input to TEX is a file in say TVeditor format at the Stanford AI lab. The output is a sequence of pages, produced in "one pass," suitable for printing on various devices. This report tries to explain how to get from input to output. The main idea is to consider the process as an operation on two-dimensional "boxes"; roughly speaking, the input is a long string of characters culled from a variety of fonts, where each character may be thought of as occupying a small rectangular box, and the output is obtained by gluing these boxes together either horizontally or vertically with various conventions about centering and justification, finally arriving at big rectangular boxes which are the desired pages. While LISP works with one-dimensional character strings, TEX works with two-dimensional box patterns; TEX has both horizontal and vertical `cons' operations. Furthermore, TEX has another important basic concept of elastic glue between boxes, a type of mortar that stretches and shrinks at different specified rates so that box patterns will fit together in flexible ways.

In order to explain TEX more fully, I will alternate between very low-level descriptions of exactly how the processing takes place and very high-level descriptions of what you type to get complex effects.

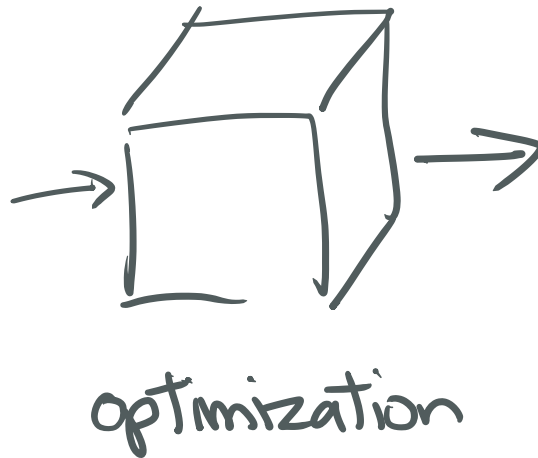
First, at the very lowest level, we must realize that the input to TEX is not really a string of boxes, it is a file of 7-bit characters. This file is called an "external input file". The first thing TEX does is convert it to an "internal input file" by essentially obeying the following rules:

- 1.Delete the first TVeditor directory page, if it exists.
- 2.Delete the first line (title line) of every remaining page, and replace the end-of-page marks ('14) by carriage-returns ('15). Delete all

Latex

```
Title { Brute force on ~ }  
\\section { TEX }  
These notes (and ~)  
⚡
```

.tex source



Brute force on sequences

Kent Quanrud

January 25, 2021

1 TeX

These notes (and hopefully yours too) are typeset using a system called L^AT_EX, which is an extension (due to Leslie Lamport) of an earlier system called T_EX, designed and largely written by Donald Knuth. This is why it looks so nice¹. In the 1960's, Donald Knuth was writing his famous (and still unfinished) series on *The Art Of Computer Programming*. At the same time, the printing industry was shifting from old-fashioned hot metal typesetting to computer driven approaches. Knuth was so unsatisfied with the galley proofs of his book under the new system that he set out to invent his own digital typesetting system. Thus T_EX was born, and he began using it to typeset his books. T_EX was soon widely used in academia, and later a set of user-friendly macros by Lamport was bundled with T_EX to form L^AT_EX.

T_EX takes unstructured text files as input and compiles it into PDF's and other print formats, similar to how programming code is compiled into binary executables. This is in contrast to WYSIWYG editors like Microsoft Word, that immediately updates a visual document as you type, and makes its typesetting decisions immediately. The compilation step of T_EX gives it a chance to optimize various criteria when making its many decisions, and ultimately produce professional quality documents.

Here we will discuss how T_EX does line-breaking (simplified for the sake of discussion), via an algorithm first proposed by Knuth in a 1977 memo² and later published by Knuth and Plass [1]. When typesetting text, a basic decision one has to make is where to wrap the line. These decisions make a big difference on the overall visual product. Having too many words in a line squeezes the white space and makes it harder to delineate words. Too few words leads to too much white space between them, which is distracting and also a waste of paper. For clarity, we restrict ourselves to breaking at white spaces between words or particular points where we can introduce a hyphen, rather than at all points within words. But this chunking makes line wrapping that much clunkier.

One way to approach line-breaking is by the following greedy process. Choose a minimum width for white space between words (e.g., half a centimeter). Each time we consider adding the next word to a line, calculate the amount of white space that would be between each pair of words if the new word were included. If this calculated value is above the minimum, then add the word to the line. If the value is below, then start a new line.

¹Sadly, beautiful typesetting cannot otherwise improve this author's prose.
²<https://www.sdlart.org/TEXOR.ATN261.DKXSD>

.pdf

Line breaking: where to wrap lines?

word₁ word₂ ~ word_k | word_{k+1} ~
wrap here

too many words
⇒ bunched up

too few words
⇒ awkwardly spaced

Greedy

if the word fits on a line, append
else start new line

example from Knuth, Plass (1981)

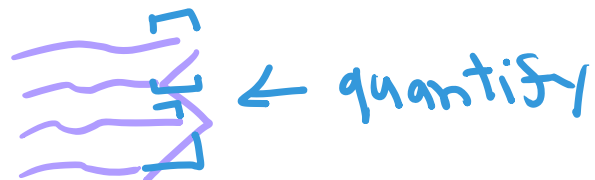
with a hyphen.

Figure 1. An example paragraph that has been typeset by the 'first-fit' method. Small triangles show permissible places to divide words with hyphens; the adjustment ratio for spaces appears at the right of each line.

In olden times when wishing still helped one, there .121
lived a king whose daughters were all beautiful; and .412
the youngest was so beautiful that the sun itself, which ..001
has seen so much, was astonished whenever it shone in ..568
her face. Close by the king's castle lay a great dark .375
forest, and under an old lime-tree in the forest was a .146
well, and when the day was very warm, the king's child ..005
went out into the forest and sat down by the side of the ..006
cool fountain; and when she was bored she took a .965
golden ball, and threw it up on high and caught it; and ..708
this ball was her favorite plaything. .001

Pros: simple, "online"
(good for WYSIWYG)

Cons: choices made on one line effect
subsequent lines,
which is overlooked by greedy



what we want:

"the **Best** linebreaking of document
over all possible ways to break document
into lines"

"Best"

define per-line loss function

$\text{Loss}(w_1, w_2, \dots, w_k)$

= numerical quantity evaluating "badness"
of $w_1, \dots, w_k$ as a line

(treat Loss as black box,
see Knuth-Plass for details)

Total loss = sum of losses over
all the lines

e.g. $\text{loss}(w_1, \dots, w_{10}) + \text{loss}(w_{11}, \dots, w_{22}) + \text{loss}(w_{23}, \dots)$

"Best" = line breaking w/ min total loss

Figure 3. This is the 'best possible' way to break the lines in the paragraph of Figures 1 and 2, in the sense of fewest total 'demerits' as defined in the text.

In olden times when wishing still helped one, there .121
lived a king whose daughters were all beautiful; and .412
the youngest was so beautiful that the sun itself, which -.881
has seen so much, was astonished whenever it shone .444
in her face. Close by the king's castle lay a great dark -.363
forest, and under an old lime-tree in the forest was .703
a well, and when the day was very warm, the king's .381
child went out into the forest and sat down by the side -.614
of the cool fountain; and when she was bored she took -.491
a golden ball, and threw it up on high and caught it; .104
and this ball was her favorite plaything. .001

example from [KP] of "best fit"

Greedy

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Best-Fit

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

How to compute:

"Best" = line breaking w/ min total loss

First line $\rightarrow$ best loss $(w_1, \dots, w_{10})$ ✓

2nd line $\rightarrow$ best ~~loss~~ loss $(w_{10}, \dots, w_{11})$

Recursive:

subproblems?

Best(i) = best loss over all ways to
partition $\{w_i, \dots, w_n\}$



$\leftarrow$ everything bagels

Step 1 define recursion

Six words $w_1, w_2, \dots, w_n$

$BLB(i)$ ("BLB" = "Best Line Breaking")

= min total loss of all line breakings
of $w_i, w_{i+1}, \dots, w_n$

Step 2 code

$BLB(i) =$

$$\left\{ \begin{array}{l} 0 \text{ if } i > n \leq n \quad \text{(base case)} \\ \text{for } j = i, \dots, n \leftarrow n-i \\ \min_{\substack{j \geq i}} \boxed{\text{loss}(w_i, \dots, w_j)} + \underline{BLB(j+1)} \end{array} \right.$$

Naive running time

$$T(n) = \sum_{i=0}^{n-1} T(i) + \underline{O(n)} = \text{big}$$

↑
guess j

Better?

Cache your answers.

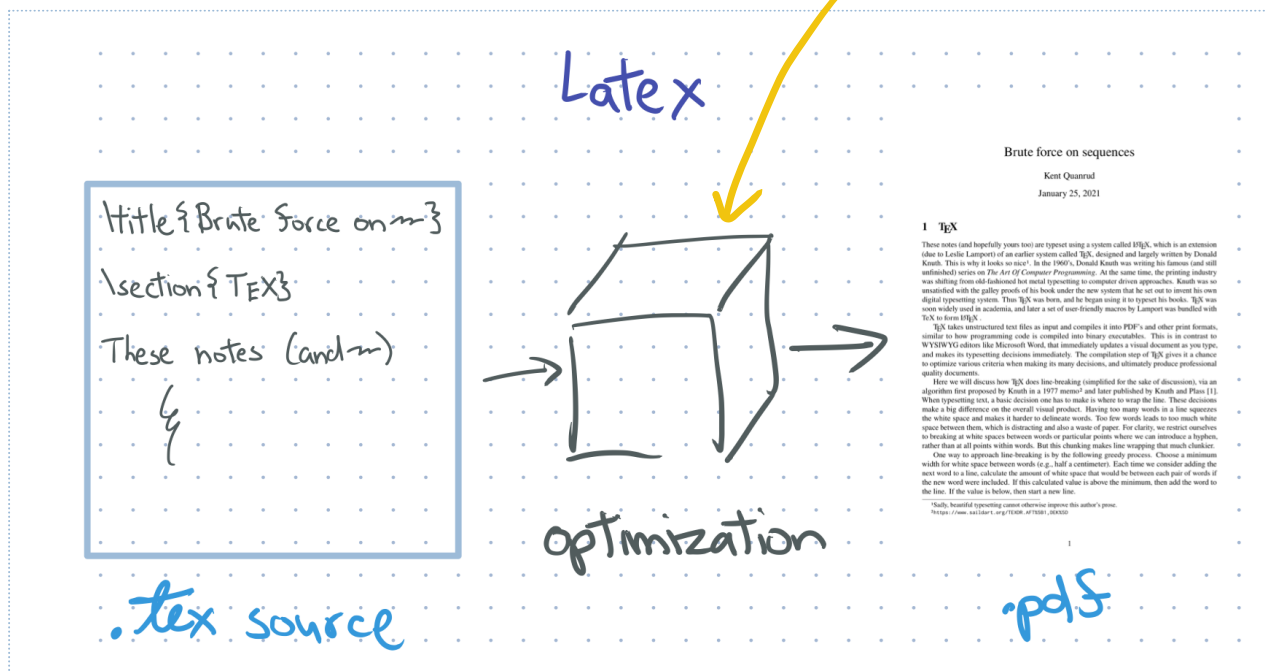
never solve same problem twice.

Running time w/ dynamic programming

$$\frac{(\# \text{ subproblems}) \times (\text{time per subproblem})}{n} = O(n) = O(n^2)$$

also, $O(n)$ space

dynamic programming



II An algorithmic metric on strings

2 homework/homework-6.md			
@@ -7,6 +7,6 @@			
7	2. Design and analyze an algorithm[^2] to compute the even distance between every pair of vertices when the edge weights are real-valued, and there are no negative cycles.	7	2. Design and analyze an algorithm[^2] to compute the even distance between every pair of vertices when the edge weights are real-valued, and there are no negative cycles.
8	2. Design and analyze an algorithm[^2] that, given a directed graph with real-valued edges, detects if there is a negative cycle.	8	2. Design and analyze an algorithm[^2] that, given a directed graph with real-valued edges, detects if there is a negative cycle.
9		9	
10	- [^1]: We pushed back the deadline to Friday because the homework was also prepared late.	10	+ [^1]: We pushed back the deadline to Friday because we were late in getting the homework up.
11	[^2]: As always, the faster the better.	11	[^2]: As always, the faster the better.
12		12	

DIFF(1)

User Commands

DIFF(1)

NAME

diff - compare files line by line

SYNOPSIS

diff [OPTION]... FILES

DESCRIPTION

Compare files line by line.

-i --ignore-case

Ignore case differences in file contents.

--ignore-file-name-case

Ignore case when comparing file names.

--no-ignore-file-name-case

Consider case when comparing file names.

-E --ignore-tab-expansion

Ignore changes due to tab expansion.

-b --ignore-space-change

Ignore changes in the amount of white space.

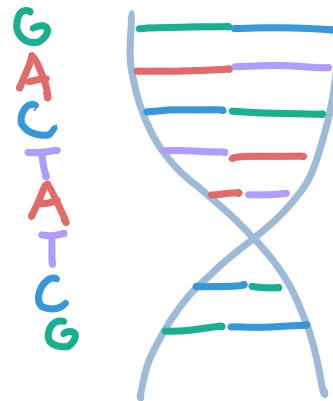
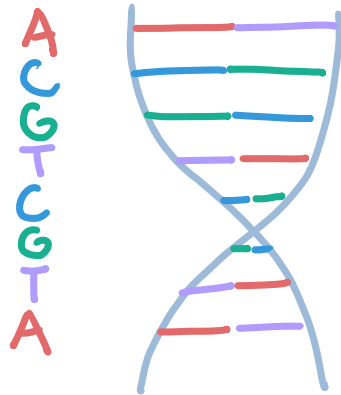
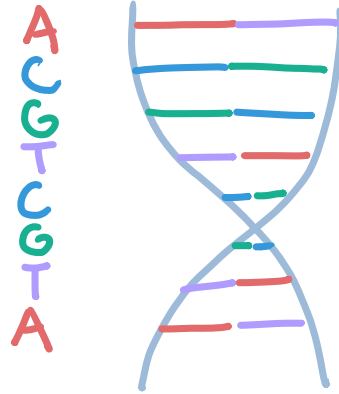
-w --ignore-all-space

Ignore all white space.

-B --ignore-blank-lines

Ignore changes whose lines are all blank.

A, C, G, T



how to measure similarity?

analyzing hereditary traits

distinguishing species

① Hamming distance

- $\text{Hamming}(x, y) = \# \text{ characters differing in } x \text{ and } y$

$x = \text{ABCDEFGG}$
 $x \ x \ x \ x$
 $y = \text{ABCEFGD}$ $\Rightarrow$ Hamming dist. 4

② Edit distance

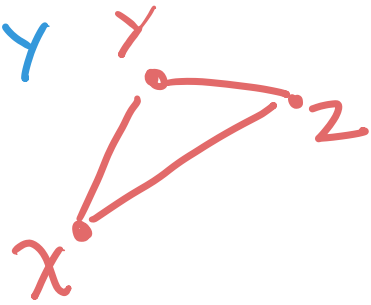
insertions, deletions, substitutions
to convert x into y

$x = \text{ABC} \overset{\text{del.}}{\cancel{\text{D}}} \text{EFG} \overset{\text{ins.}}{\text{D}}$
 $y = \text{ABCEFGD}$ $\Rightarrow$ Edit. distance = 2

Note: $\uparrow$ well-defined

Hamming/edit distance are both metrics

- $\text{Dist}(x, y) = 0 \iff x = y$



- $\text{Dist}(x, y) = \text{Dist}(y, x)$

^{edit}
^{min} $\text{Dist}(x, y) \leq \text{Dist}(x, z) + \text{Dist}(z, y)$

(triangle inequality)

- Edit distance is a better model for many applications

- But also more complicated:
maybe not efficient?

How to compute

② Edit distance

insertions, deletions, substitutions
to convert x into y

Step 1 Recursive spec

Fix strings $x[1 \sim m]$, $y[1 \sim n]$

$\text{edit}(i, j) =$ min edit distance
between $x[i, i+1, \sim, m]$ and $y[j, j+1, \sim, n]$

Step 2 implement

different cases...

Case 1 1 character differs

X = DEFG ~

Y = EFGD ~

⇒ have to do something

① replace "D" w/ "E"

② delete "D"

③ insert "E"

} which one?



$\text{edit}(i, j) =$

$$\min \left\{ \begin{array}{l} \text{subst. } 1 + \text{edit}(i+1, j+1) \\ \text{ins. } 1 + \text{edit}(i, j+1) \\ \text{del. } 1 + \text{edit}(i+1, j) \end{array} \right\}$$

when
 $x[i] \neq y[j]$

Case 2 1st character ~~differs~~ same

X = EFG ~

Y = EFGD ~

⇒ don't
have to do
something

- ① ~~replace "D" w/ "E"~~ ^{do nothing}
- ② delete "D" or "E"
- ③ insert "E"

which one?

~~generally hard to choose.~~

seems obvious

lazy



$\text{edit}(i, j) =$

$$\left\{ \begin{array}{l} \text{min} \left\{ \begin{array}{l} 1 + \text{edit}(i+1, j), \\ 1 + \text{edit}(i, j+1), \\ \text{edit}(i+1, j+1) \end{array} \right\} \end{array} \right\}$$

do nothing

when
 $x[i] = y[j]$

Step 1 Recursive spec

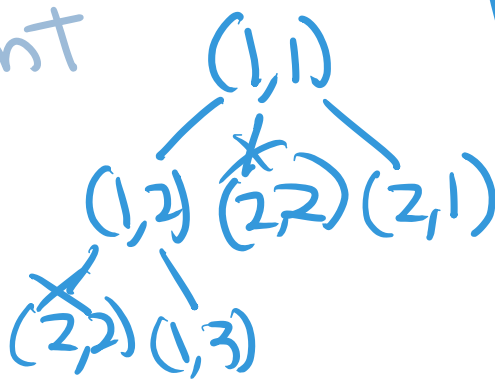
Two strings $x[1 \sim m]$, $y[1 \sim n]$

$\text{edit}(i, j) =$ minimum edit distance

between $x[i, i+1, \sim, m]$ and $y[j, j+1, \sim, n]$

Step 2 implement

$\text{edit}(i, j) =$



$i > m, j > n$

$j > n$

$i > m$

when

$x[i] \neq y[j]$

when

$x[i] = y[j]$

$1 + \text{edit}(i+1, j)$

$1 + \text{edit}(i, j+1)$

$\min \left\{ \begin{array}{l} 1 + \text{edit}(i+1, j), \\ 1 + \text{edit}(i, j+1), \\ 1 + \text{edit}(i+1, j+1), \end{array} \right\}$

$\min \left\{ \begin{array}{l} 1 + \text{edit}(i+1, j), \\ 1 + \text{edit}(i, j+1), \\ \text{edit}(i+1, j+1) \end{array} \right\}$

Initially running time is bad.

But w/ dynamic programming:

(# subproblems) $\times$ (time per subproblem)

$$= m \times n \cdot O(1) = O(mn)$$

also, space is $O(mn)$

hooray for



Multiplying (Many) Matrices

$$A \in \mathbb{R}^{n_1 \times n_2}$$

(rows) (columns)

$$\begin{matrix} \uparrow \\ n_1 \\ \downarrow \end{matrix} \left[\begin{array}{cccc} A_{11} & A_{12} & A_{13} & \dots & A_{1,n_2} \\ A_{21} & & & & \vdots \\ \vdots & & & & \vdots \\ A_{n_1,1} & \dots & & & A_{n_1,n_2} \end{array} \right]$$

$\leftarrow n_2 \rightarrow$

$$A_1 \in \mathbb{R}^{n_1 \times n_2}, \quad A_2 \in \mathbb{R}^{n_2 \times n_3}$$

$$A_1 A_2 \in \mathbb{R}^{n_1 \times n_3}$$

$$\left[\begin{array}{c} \text{---} \\ \text{---} \end{array} \right] \left[\begin{array}{c} | \\ | \end{array} \right]$$

$$(A_1 A_2)_{ij} = \sum_{k=1}^{n_2} A_{1,i,k} A_{2,k,j}$$

$(A_1 A_2)$ takes $O(n_1 n_3 n_2)$ time

Associativity

$$A_i \in \mathbb{R}^{n_i \times n_{i+1}}$$

$$\begin{array}{c} n_1 \times n_3 \\ n_1 n_2 n_3 \\ \rightarrow (A_1 A_2) A_3 = A_1 (A_2 A_3) = A_1 A_2 A_3 \\ \underbrace{\hspace{1.5cm}}_{n_1 n_3 n_4} \quad \underbrace{\hspace{1.5cm}}_{n_2 n_3 n_4} \\ \underbrace{\hspace{2.5cm}}_{n_1 n_2 n_4} \end{array}$$

mathematically.

but computationally:

$(A_1 A_2) A_3$ takes

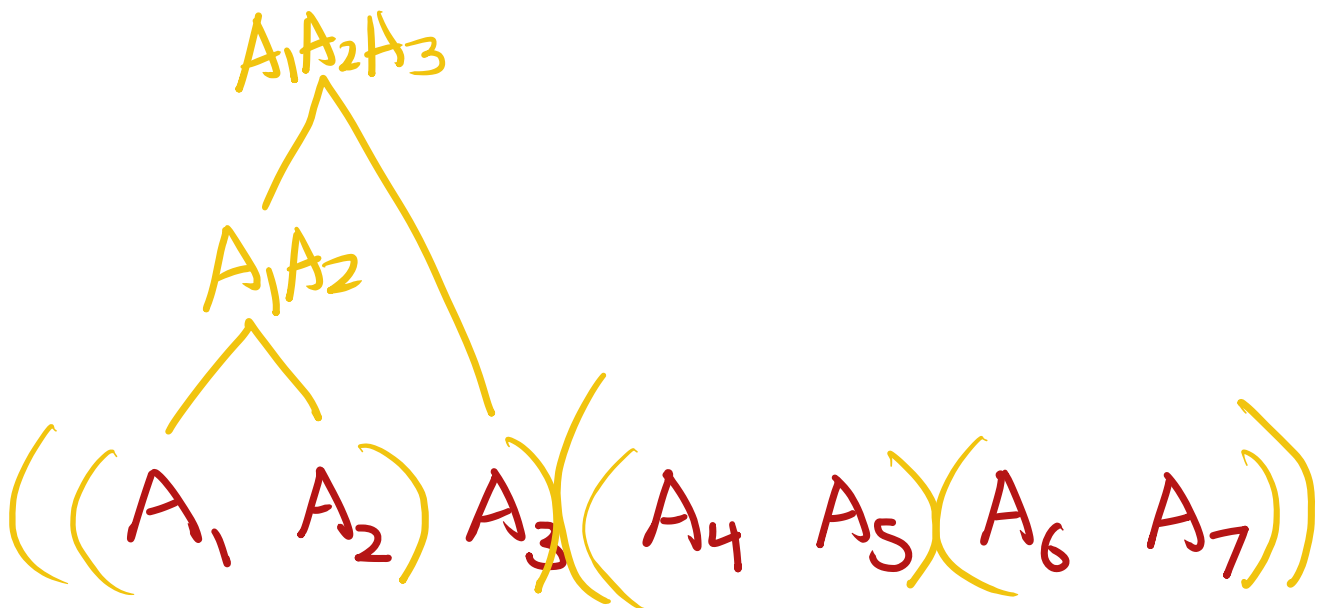
$A_1 (A_2 A_3)$ takes

Problem:

Input: $A_1 \in \mathbb{R}^{n_1 \times n_2}$, $A_2 \in \mathbb{R}^{n_2 \times n_3}$, ..., $A_k \in \mathbb{R}^{n_k \times n_{k+1}}$

Goal: Find fastest order to compute

$$A_1 A_2 \dots A_k \in \mathbb{R}^{n_1 \times n_{k+1}}$$



Problem:

Input: $A_1 \in \mathbb{R}^{n_1 \times n_2}$, $A_2 \in \mathbb{R}^{n_2 \times n_3}$, ..., $A_k \in \mathbb{R}^{n_k \times n_{k+1}}$

Goal: Find fastest order to compute

$$A_1 A_2 \cdots A_k \in \mathbb{R}^{n_1 \times n_{k+1}}$$

Subproblems?

(Recursive Spec)

Best(i, j) =

compute $A_i A_{i+1} \rightsquigarrow A_j$

to

Best(i, j) =

$\left\{ \begin{array}{l} 0 \text{ if } i \geq j \end{array} \right.$

(# subproblems) $\times$ (time per subproblem)

COMMENT * VALID 00016 PAGES

C REC PAGE DESCRIPTION

C00001 00001

C00004 00002 Preliminary preliminary description of TEX

C00007 00003 In order to explain TEX more fully, I will alternate between very low-level

C00013 00004 1 %Example TEX input related to Seminumerical Algorithms using ACPdefs

C00028 00005 The first thing that must be emphasized about this example is that it is written

C00034 00006 It is time to explain TEX's mechanism for stretching and shrinking.

C00043 00007 Now let's look at more of the example. Lines 21 and 32, etc., are blank lines

C00049 00008 Line 46 begins a "boxinsert", one of the important features needed in page layout.

C00054 00009 The ``big(){...}`` in lines 113, 143, etc. and the ``big/{...}`` in line 152

C00058 00010 The next step in understanding TEX is to go back to the low level and see

C00066 00011 Some tokens are not defined as macros but stand for actions to be carried out.

C00070 00012 At the outermost level, TEX has implicitly preceded the input by "TEXpublish{"

C00078 00013 Now let's consider the page-building routine "nextpage" this gives us a chance

C00086 00014 The nextparagraph routine assembles lines of width hsize, pausing to

C00094 00015 Besides using the permissible breaks, TEX will try to hyphenate words.

C00102 00016 To conclude this memo, I should explain how TEX is going to work on

C00103 ENDMK

C*;

Preliminary preliminary description of TEX

D Knuth, May 13, 1977

In this memo I will try to explain the TEX system for preparing publishable documents and how it is proposed to implement the system. Even though I don't understand TEX very well myself yet, I think the best way for me to get started is by trying to explain it to somebody else.

TEX is for technical text. Insiders pronounce the X as a Greek Chi (cf. the Scottish `ch' sound in `Loch Ness') since the word `technical' stems from a Greek root meaning art as well as technology. I am preparing the system primarily for use in publishing my series The Art of Computer Programming--the initial system will be tuned for my books, but it will not be difficult to extend it for other purposes if anybody wants to do so.

The input to TEX is a file in say TVeditor format at the Stanford AI lab. The output is a sequence of pages, produced in "one pass," suitable for printing on various devices. This report tries to explain how to get from input to output. The main idea is to consider the process as an operation on two-dimensional "boxes"; roughly speaking, the input is a long string of characters culled from a variety of fonts, where each character may be thought of as occupying a small rectangular box, and the output is obtained by gluing these boxes together either horizontally or vertically with various conventions about centering and justification, finally arriving at big rectangular boxes which are the desired pages. While LISP works with one-dimensional character strings, TEX works with two-dimensional box patterns; TEX has both horizontal and vertical `cons' operations. Furthermore, TEX has another important basic concept of elastic glue between boxes, a type of mortar that stretches and shrinks at different specified rates so that box patterns will fit together in flexible ways.

In order to explain TEX more fully, I will alternate between very low-level descriptions of exactly how the processing takes place and very high-level descriptions of what you type to get complex effects.

First, at the very lowest level, we must realize that the input to TEX is not really a string of boxes, it is a file of 7-bit characters. This file is called an "external input file". The first thing TEX does is convert it to an "internal input file" by essentially obeying the following rules:

- 1.Delete the first TVeditor directory page, if it exists.
- 2.Delete the first line (title line) of every remaining page, and replace the end-of-page marks ('14) by carriage-returns ('15). Delete all

line-feed symbols ('12). Delete all % marks and the sequences of characters following them up to (but not including) the next carriage-return.

3.Delete all blank spaces following carriage-returns.

4.If two or more carriage returns occur in sequence, replace all but the first by vertical-tab characters ('13). These are used to specify end of paragraphs in TEX; in other words, the user specifies end of paragraph by hitting two carriage returns in a row, or by a page break following a carriage return.

5.Replace all remaining carriage-returns by blank spaces.

6.If two or more blank spaces occur in a row, replace them by a single blank space.

7.Add infinitely many } symbols at the right.

The reason for rule 7 is that TEX uses { and } for grouping, and the trailing }'s will match up with any {'s the user erroneously forgot to match in the external input file. By following the above rules, TEX obtains an internal input file containing no carriage-returns, line-feeds, %'s, or form-feeds (page marks), and with no two blank spaces in a row. Spacing in the output document is controlled by other features of TEX. (Actually TEX changes { and } internally to '14 and '15, but this does not affect the user so I will continue to write this report as if they were { and }.)

Now let's shift to a high level and show how the user can specify complex typesetting requirements to TEX. The following example is rather long and it deserves to be skimmed rather than studied in detail; I devised it mainly to serve as test data during initial development of the system. It is taken from the opening pages of my book *Seminumerical Algorithms*, skipping over the copy when the typesetting presents no essentially new challenges to the system. The reader might find it most useful to have the book in hand for comparison purposes. The line numbers at the left are not part of the TEX input, they are included only for purposes of cross reference so that I can refer to line so-and-so later on in this document.

```
1 %Example TEX input related to Seminumerical Algorithms using ACPdefs
2 ACPpages starting at page 1{
3 titlepage %This tells the page format routine not to put page number at top
4 sectionnumber 3
5 runningleftthead{RANDOM NUMBERS}
6 hexpand 11 pt {fnt cmg11 CHAPTER hskip 10 pt THREE}
7 vskip 1 cm plus 30 pt minus 10 pt
8 rjustline {fnt cmgb20 RANDOM NUMBERS}
9 vskip .5 cm plus 1 pc minus 5 pt
10 quoteformat {Anyone who considers arithmetical \cr
11 methods of producing random digits \cr is, of course,
12 in a state of sin. \cr} author {JOHN VON NEUMANN (1951)}
13 quoteformat {Round numbers are always false.}
14 author{SAMUEL JOHNSON (c. 1750)}
15 vskip 1 cm plus 1 pc minus 5 pt
16 sectionnumber 3.1
17 sectionbegin {3.1. INTRODUCTION}
18 runningrighthead {INTRODUCTION}
19 Numbers which are ``chosen at random'' are useful in a wide variety of
20 applications. For example:
21     %This blank line specifies end of paragraph
22 yskip %This means an extra space between paragraphs
23 textindent{a}){\sl Simulation.}xskip When a computer is used to simulate natural
24 phenomena, random numbers are required to make things realistic. Simulation
25 covers many fields, from the study of nuclear physics (where particles are
26 subject to random collisions) to systems engineering (where people come into,
27 say, a bank at random intervals). \par
28 yskip textindent{b}){\sl Sampling.} xskip It is often impractical to examine all
29 cases, but a random sample will provide insight into what constitutes ``typical''
30 behavior.
31
32 yskip It is not easy to invent a foolproof random-number generator. This fact
33 was convincingly impressed upon the author several years ago, when he attempted
```

```

34 to create a fantastically good random-number generator using the following
35 peculiar method:
36
37 yskip yskip noindent {\bf Algorithm K} xskip({\sl ``Super-random'' number
38 generator}). xskip Given a 10-digit decimal number  $XX$ , this algorithm may be
39 used to change  $XX$  to the number which should come next in a supposedly random
40 sequence. \par
41 algstep K1. [Choose number of iterations.] Set  $Y \leftarrow \lfloor X/10 \sup 9 \rfloor$ , i.e.,
42 the most significant digit of  $XX$ . (We will execute steps K2 through K13  $Y+1$ 
43 times; that is, we will randomize the digits a {\sl random} number of times.)
44
45 algstep K10. [99999 modify.] If  $XX < 10 \sup 5$ , set  $XX \leftarrow X \sup 2 + 99999$ ;
46 otherwise set  $XX \leftarrow X - 99999$ . xskip blacksug
47
48 boxinsert{ctrline{fnt cmgb10 Table 10}
49 ctrline{fnt cm9 A COLOSSAL COINCIDENCE: THE NUMBER 6065038420}
50 ctrline{fnt cm9 IS TRANSFORMED INTO ITSELF BY ALGORITHM K.}
51 blankline 3 pt vskip ruled
52 hjust to 12 pc{blankline vskip 6 pt
53 tabalign{#quad#quad#\$}\cr
54 Stepctr{XX (after)}\cr
55 vskip 10 pt plus 10 pt minus 5 pt
56 K16065038420\cr K121905867781Y=5\cr
57 vskip 10 pt plus 10 pt minus 5 pt blankline}
58 hskip 3 pc ruled align top
59 hjust to 12 pc{blankline vskip 6 pt
60 tabalign{#quad #quad #\$}\cr
61 Stepctr{XX (after)}\cr
62 vskip 10 pt plus 10 pt minus 5 pt
63 K101620063735\cr K111620063735\cr K126065038420Y=0\cr
64 vskip 10 pt plus 10 pt minus 5 pt blankline}
65 vskip ruled blankline} %end of the boxinsert
66 yskip yskip The moral to this story is that {\sl random numbers should not be
67 generated with a method chosen at random}. Some theory should be used.
68
69 exbegin
70 exno tr 1. [20] Suppose that you wish to obtain a decimal digit at random, not
71 using a computer. Shifting to exercise 16, let  $f(x,y)$  be a function such that
72 if  $0 \leq x, y < m$ , then  $0 \leq f(x,y) < m$ . The sequence is constructed by selecting
73  $X \sub 0$  and  $X \sub 1$  arbitrarily, and then letting
74  $X \sub {n+1} = f(X \sub n, X \sub {n-1})$  quad for quad  $n > 0$ .
75 What is the maximum period conceivably attainable in this case?
76
77 exno 17. [10] Generalize the situation in the previous exercise so that
78  $X \sub {n+1}$  depends on the preceding  $k$  values of the sequence.
79 \ff
80 sectionnumber 3.2 sectionbegin{3.2. GENERATING UNIFORM RANDOM NUMBERS}
81 runningrighthead {GENERATING UNIFORM RANDOM NUMBERS}
82 In this section we shall consider methods for generating a sequence of random
83 fractions, i.e., random {\sl real numbers}  $U \sub n$ , uniformly distributed
84 between zero and one. Since a computer can represent a real number with only
85 finite accuracy, we shall actually be generating integers  $X \sub n$  between
86 zero and some number  $m$ ; the fraction  $U \sub n = X \sub n/m$  eqno 1 will
87 then lie between zero and one.
88
89 vskip .4 in plus .2 in minus .2 in
90 sectionnumber 3.2.1 sectionbegin{3.2.1. The Linear Congruential Method}
91 runningrighthead{THE LINEAR CONGRUENTIAL METHOD}
92 By far the most successful random number generators known today are special
93 cases of thhe following scheme, introduced by D. H. Lehmer in 1948. [See
94 {\sl Annals Harvard Comp. Lab.} {\bf 26} (1951), 141-146.] We choose four
95 ``magic numbers'':
96 tabalign{rjust#quad mathoff # quadorjjust #ljust #\cr
97  $X \sub 0$ , the starting value;  $X \sub 0 \geq 0$ . \cr
98  $m$ , the modulus;  $m > X \sub 0$ , quad  $m > a$ , quad  $m > c$ . \cr} eqno 1

```

The desired sequence of random numbers $\langle X_n \rangle$ is then obtained by setting $X_{n+1} = (X_n + c) \bmod m$, $n \geq 0$. This is called a linear congruential sequence.

Let w be the computer's word size. The following program computes $(X_n + c) \bmod (w+1)$ efficiently:

```
LDAN=X\cr
MUL=A\cr
JANN=**+3\cr
ADD=W-1= quad blackslug\cr eqno 2$
{\sl Proof.} xskip We have  $x = 1 + qp$  sup e$ for some integer $q$ which is not a multiple of $p$. By the binomial formula$
equalign{x sup p=1+{p choose 1}qp sup e + cdots +{p choose p-1}q sup {p-1}p sup {(p-1)e}+q sup p p sup {pe}\cr
=1+qp sup {e+1} big(){1 + 1 over p {p choose 2} q p sup e + 1 over p 0p choose 3} q sup 2 p sup {2e} + cdots + 1 over p {p choose p} q sup {p-1} p sup {(p-1)e}.\cr}$
By repeated application of Lemma P, we find that$
equalign{(a sup p sup g - 1)/(a-1)\equiv 0` (modulo p sup g), \cr
(a sup p sup g - 1)/(a-1)\equiv 0` (modulo p sup {g+1}). \cr}eqno 6$
If  $1 < k < p$ ,  $p$  choose  $k$  is divisible by  $p$ . {\biglpren}{\sl Note:} xskip A generalization of this result appears in exercise 3.2.2-11(a).{\bigrpren} By Euler's theorem (exercise 1.2.4-48),  $a^{\varphi(p^{e-f})} \equiv 1 \pmod{p^{e-f}}$ ; hence  $\lambda$  is a divisor of$
lambda(p sub 1 sup e sub 1 ldots p sub t sup e sub t)=lcm paren{lambda(p sub 1 sup e sub 1), ldots, lambda(p sub t sup e sub t)}.$eqno 9$
```

This algorithm in MIX is simply

```
tablsign{hjust to 50 pt{\tt#}*{\tt#}} quad \cr
J6NN=*+2\underline{{it A1. j<0?}}\cr
STA@Zquad quad $-Z$.\cr}eqno 7$
```

That was on page 26. If we skip to page 49, $Y_1 + \cdots + Y_k$ will equal n with probability

$$\sum_{y_1 + \cdots + y_k = n} \prod_{1 \leq s \leq k} \frac{e^{-np_s} (np_s)^{y_s}}{y_s!}$$

This is not hard to express in terms of n -dimensional integrals

$$\int_{\alpha_n}^n dY_n \int_{\alpha_{n-1}}^{Y_n} dY_{n-1} \ldots \int_{\alpha_1}^{Y_2} dY_1 \text{ over } \int_0^n dY_n \int_0^{Y_n} dY_{n-1} \ldots \int_0^{Y_2} dY_1, \text{ quad where } \alpha_j = \max(j-t, 0).$$

eqno 24\$

This together with (25) implies that

$$\lim_{n \rightarrow \infty} \sum_{rtn \leq k \leq n} \binom{n}{k} \frac{k^n}{(n-k)!} = e^{-2s} \sum_{s=0}^\infty \frac{s^k}{k!}, \text{ quad } s \geq 0,$$

eqno 27\$ a limiting relationship which appears to be quite difficult to prove directly.

exbegin exno 17. [HM26] Let t be a fixed real number. For $0 \leq k \leq n$, let

$$P_n(x) = \int_{n-t}^x dx_n \int_{n-1-t}^{x_{n-1}} dx_{n-1} \ldots \int_{k+1-t}^{x_{k+2}} dx_{k+1} \int_0^{x_{k+1}} dx_k \ldots \int_0^{x_2} dx_1;$$

Eq. (24) is equal to

$$\sum_k X'_k Y'_k \bigg/ \sqrt{\sum_k X'_k Y'_k}$$

sectionnumber 3.3.3.3 subsectionbegin{3.3.3.3. This subsection doesn't exist}

runningrighthhead{A BIG MATRIX DISPLAY} Finally, look at page 91.

def diagdots{raise . by 10 pt hskip 4 pt raise . by 5 pt hskip 4 pt .}

$$U = \bigg(\begin{matrix} & & & \\ & & & \\ & & & \\ & & & \\ & & & \end{matrix} \bigg)$$

U sup {-1}=big(){tabalign{ctr#ctr#ctr#ctr#ctr#ctr#\cr l\crodiaqdots\crod1\cr c sub 1 ldots c sub {k-1} c sub {k+1} c sub n\cr 1 ldots ldiaqdots\crod1\cr}} , \cr U sup {-1}=big(){tabalign{ctr#ctr#ctr#ctr#ctr#ctr#\cr l\crodiaqdots\crod1\cr

```
164 -c sub 1oldots@c sub{k-1}01@c sub{k+1}0-c sub n\cr
```

```
165 @@@@1\cr@@@@@diagdots\cr@@@@@1\cr}}$$
```

```
166 This ends the test data, fortunately TEX is working fine.}
```

The first thing that must be emphasized about this example is that it is written in an extension of TEX, not in the basic language itself. For example, "ACPpages" in line 2 is a special routine that calls TEX's basic page-building routine but uses it to prepare pages in the format of The Art of Computer Programming. The words titlepage, sectionnumber, runninglefthead, quoteformat, author, sectionbegin, runningrighthead, xskip, yskip, textindent, algstep, exbegin, exno, and subsectionbegin are specific to my books and they have been defined in terms of lower-level TEX primitives as we shall see below. Furthermore most of the fonts used are embedded in these hidden definitions; for example, "sectionbegin" defines the 10 point fonts used in the text of a section, while "exbegin@" defines the 9 point fonts used in the exercises. Such keywords are chosen so that they do not match English words, since they can appear intermixed with normal text. For example, another definition is that the word MIX always is to be set in the "typewriter type" font; the hidden definition

```
def MIX|{{\tt \{MIX}}}
```

causes this to happen automatically in line 125. We shall study TEX's macro definition mechanism later; three simple examples appear in lines 141, 152, and 157 of the sample program, where a common idiom was given a short definition just to save space. The construction \{string} is used to transmit a string without interpreting its words according to existing definitions; this device is used in the above definition of the keyword MIX. For curious people who want to see a more complex definition, here is the definition of quoteformat:

```
def quoteformat #1 author #2 {lineskip 3 pt plus .5 pt minus 1 pt
  vskip 6 pt plus 5 pt minus 2 pt
  def \rm {fnt cmg8} def \sl {fnt cmgi8}
  {\sl tabalign{rjust##\cr #1}}
  vskip 6 pt plus 5 pt minus 2 pt \rm rjustline #2
  vskip 11 pt plus 5 pt minus 2 pt}
```

Please don't expect to understand this mess now, TEX is really very simple; trust me. The word ``author'' which appears in this definition is given special interpretation only in the context of quoteformat, otherwise it is treated as usual; that is why I didn't use a non-English word like ``quoteauthor'' in lines 12 and 14 of the example.

The specifications of sectionnumber and runninglefthead in lines 4 and 5 are to be used in the top lines of pages in the book. Line 6 contains the first actual text to appear on the page; but look first at line 8, which is simpler: Line 8 says to use font cmgb20 (namely, "Computer Modern Gothic Bold 20 point", which I plan to design shortly) for the words RANDOM NUMBERS, and to right-justify them on the line (rjustline). Line 6 is similar but it uses the 11-point font cmg11; ``hskip 10 pt'' stands for 10 points of horizontal space, and ``hexpand 11 pt'' means to take the line and stretch it 11 points wider than it would ordinarily be. Note that font definitions within {...} disappear outside the braces. So do all other definitions.

It is time to explain TEX's mechanism for stretching and shrinking.

Sizes in TEX can be specified in terms of points, picas, inches, centimeters, or millimeters, using the abbreviations pt, pc, in, cm, mm respectively; these abbreviations are meaningful only in contexts where physical lengths are used, and one of these units must always be supplied in such contexts even when the length is 0. (One inch equals 6 picas equals 72 points equals 2.54 centimeters equal 25.4 millimeters.) The glue between boxes has three components:

```
the fixed component, x
the plus component, y
the minus component, z
```

The x part is "normal" spacing; when expanding a sequence of boxes to more than their normal length, as on line 6, each x part in the glue is increased by some multiple of the y part. When shrinking, each x part is decreased by some multiple of the corresponding z part.

For example, given four boxes bound together by glue of specifications

$(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3),$
 expansion of the line by an amount w is achieved by using the spacing
 $x_1 + y_1 w', x_2 + y_2 w', x_3 + y_3 w',$
 where $w' = w/(y_1+y_2+y_3)$. The expansion is impossible if $w>0$ and $y_1+y_2+y_3=0$.
 The system tries hard to minimize bad-looking expansions, by having a
 reasonably intelligent justification routine (described below). When
 shrinking a line, the maximum amount of contraction is the sum of the z 's, no
 tighter fit will be attempted; thus, one should define the z width so that
 $x-z$ is the minimum space tolerated. The proper setting of y is such that $x+y$
 is a reasonable spacing but $x+3y$ is about at the threshold of intolerability.
 Parameters y and z must be nonnegative, but may be negative (for backspacing
 and overlap) if care is used.

The notation ```hskip 10 pt''` in line 6 means that $x = 10$ pt, $y = 0$, and $z = 0$
 in the horizontal glue between CHAPTER and THREE. If this `hskip` hadn't been
 used, the normal conventions would have applied. Namely, the glue between
 CHAPTER and THREE would have had x equal to the normal spacing for the font,
 and $y = x$, $z = x/2$. The glue between letters has (say) $x = 0$, $y = 1/10$ of the
 normal spacing for the font, $z = 0$; thus the expansion without using `hskip`
 would have gone mostly into the space between CHAPTER and THREE, but by using
`hskip` as shown the expansion spreads out the individual letters. Fonts to
 be used with TEX will have such letter spacing indicated as an additional feature;
 TEX will also recognize things like the end of sentences, giving slightly more
 white space after punctuation marks where appropriate. Symbols like $+$ and $=$
 conventionally are surrounded by spaces, while symbol pairs like `'` and `:=` are
 moved closer together; the symbols $+$ and $=$ are not surrounded by spaces when
 they appear in subscripts. Such things are taken care of in the same way that
 ligatures are handled, by making the fonts a little smarter under TEX's control,
 as described in more detail later.

Much of TEX is symmetrical between horizontal and vertical, although the basic
 idea of building lines first rather than columns first is asymmetrically built
 in to the standard routines (because of the way we normally write). The
 specification ```vskip''` on line 7 specifies vertical glue analogous to
 horizontal glue. When the page justification routine tries to fill the first
 page of the example text (by breaking between pages ideally at the end of a
 paragraph, or in a place where at least two lines of a paragraph appear on
 both pages), this glue will expand or shrink in the vertical dimension using
 $x = 1$ centimeter, $y = 30$ points, $z = 10$ points. Further variable glue is
 specified in the definition of `quoteformat`: `lineskip` is the inter-line spacing,
 and the `vskips` give special additional spacing between quotation and author
 lines. In the main text, the printer would refer to the type as "10 on 12",
 meaning 10 point type with 12 points between corresponding parts of adjacent
 lines; TEX will use a 10 point font with

```
lineskip 2 pt plus .5 pt minus .25 pt
```

so that its line spacing is somewhat more flexible. Additional spacing between paragraphs is given by

```
parskip 0 pt plus 1 pt minus 0 pt
```

and there is other spacing between exercises, steps in algorithms, etc. The definition

```
def yskip {vskip 3 pt plus 2 pt minus 2 pt}
```

is used for special vertical spacing, for example in lines 22 and 37. A horizontal skip

```
def xskip {hskip 6 pt plus 10 pt minus 3.5 pt}
```

is used in lines 23, 35, etc. in contexts where a larger than normal space is psychologically acceptable; for such purposes, flexible glue is especially useful. A larger horizontal space called ```quad''` is used for example in line 74; this is "two ems" of space, a unit frequently used in mathematical typesetting.

Another nice feature of flexible glue occurs when you consider the case

```
def hfill {hskip 0 cm plus 1000 cm minus 0 cm}.
```

In other words, y is essentially infinite (10 meters long). When this symbol appears as an `hskip` at the beginning of a line, it right justifies that line; when it appears at both the beginning and end, it centers the line; when it appears in the middle it neatly partitions the line; and so on. These features of

TEX's variable glue seem to make it superior to existing justification systems, because it provides new features in a rather simple way and at the same time reduces the old features to a common pattern.

Once a list of boxes has been justified, all the glue is permanently set and the overall box becomes rigid. Justification is either horizontal or vertical. Horizontal justification of long lines includes some automatic hyphenation; further details about justification appear later in this memo.

Now let's look at more of the example. Lines 21 and 32, etc., are blank lines indicating the end of a paragraph; another way to specify this is ```\par''` in line 27. Line 124 is a end of a paragraph that ends with a displayed formula. Paragraphs are normally indented; one of the TEX commands subsumed under "sectionbegin" in line 17 is

```
parindent 20 pt
```

which affect the first line of every paragraph unless ```noindent''` appears as on line 37. The "sectionbegin" routine also specifies ```noindent''` on the very first paragraph of the section, since this is the standard format in my books. On line 23 we have ```textindent{a}''`, which creates a box of width parindent containing the characters a) followed by a blank space, right justified in this box.

In line 23 the `"\sl"` means "use slanted font." I have tentatively decided to replace italics by a slanted version of the normal "Roman" typeface, for emphasized words in the text, while variable symbols in math formulas will still be in italics as usual. I will have to experiment with this, but my guess is that it will look a lot better, since typical italic fonts do not tie together into words very beautifully. At any rate I will be distinguishing slanted from italic, in case it becomes desirable to use two different fonts for these different purposes. The `"\bf"` in line 35 stands for boldface. All these fonts are defined in sectionbegin, whose definition is hidden to you.

Mathematical formulas all appear between `$...$` pairs, cf. lines 38 and 41, or between `$$...$$` pairs (displayed equations). A special syntax is used in formulas, modeled on that of Kernighan and Cherry, Comm. ACM 18 (March 1975), 151-157. For example, ```sup 9''` in line 41 specifies a superscript 9, ```sub {n+1}''` in line 74 specifies a subscript n+1. Keywords like sup and sub are active only with `'$'`; the same applies to greek letter names like lambda (line 122) and varphi ("variant phi", the rounded version of this letter, which appears in a superscript in line 120), as well as to words like lim (line 142), max (line 140), lcm (line 122). All letters in formulas are set in italics unless they form part of a recognized word or are surrounded by `"mathoff{...}"` or `"{\rm ...}"`.

All spacing within formulas is chosen by TEX, independent of spaces actually typed; the symbol ``` (cf. line 117) denotes an inserted space, for cases when TEX's rules are unsatisfactory. In line 117, for example, extra space is wanted before `"(modulo...)"` since space before parentheses is usually omitted but not here. The later parts of the example text are largely concerned with complicated formulas, which will appear as shown in the corresponding parts of volume 2. The code `"eqno 24"` (cf. line 140) will insert `"(24)"` at the right margin, vertically centered on the corresponding displayed formula, if there is room, otherwise the `"(24)"` is stuck on a new line at the bottom right.

The algorithm-step-format keyword "algstep" used on lines 41 and 45 is defined as follows:

```
def algstep #1. [#2] {vskip 3 pt plus 1 pt minus 1 pt
noindent rjust in 20 pt{#1.} [#2] xskip hangindent 20 pt}
```

This sets vertical spacing glue before the text for the algorithm step, and it also set up an appropriate "textindent", etc. The new feature here is the hanging indent, which affects all but the first line of the following paragraph.

The keyword "exno" used on lines 70, 77, etc. has a definition somewhat similar to algstep; such definitions of format that are needed frequently in my books will help to ensure consistency. The ```tr''` in line 70 will insert

a triangle in the left margin, using negative glue spacing so that the character actually appears to the left of the box it is in. Line 46 begins a "boxinsert", one of the important features needed in page layout. The box defined within the {...} after boxinsert is set aside and placed on top of either the present page or the following page (followed by vertical glue specified by

boxskip 20 pt plus 10 pt minus 5 pt, this being another thing subsumed by "sectionbegin"). This is used for figures and tables, in this case a table. The the table caption appears in lines 48-50; the table itself (cf. page 6 of the book) is rather complicated, so we will defer explanation of lines 52-65 until after we have studied the simpler example of tabalign in lines 96-98.

In general, a tabalign command takes the form

```
tabalign{ u1#v1 @ ... @ un#vn \cr
          x11 @ ... @ x1n \cr
          . . . . .
          xm1 @ ... @ xmn \cr}
```

Here the @'s represent <TAB>'s on the keyboard and in the input file, but they are shown here explicitly as printed characters to make their appearance plain. The "\cr" is not a carriage-return, however, it is the sequence of three characters \, c, r. The meaning is to form the mn boxes

```
u1{x11}v1 ... un{x1n}vn
. . . . .
u1{xm1}v1 ... un{xmn}vn
```

and, for each k, to determine the maximum width of box $u_k\{x_{ik}\}v_k$ for $i = 1, \dots, m$. Then each $u_k\{x_{ik}\}v_k$ is justified to the size of this maximum width, and each line $x_{i1} @ \dots @ x_{in} \backslash cr$ is replaced by the horizontal concatenation of the resulting boxes, separated by

tabskip 0 pt plus 1 pt minus 0 pt.

If less than n entries appear on any line before the \cr, the remaining entries are left blank.

In the example of tabalign on lines 96-98 we have n=4; the first column is to be right justified, the second is to be treated as "mathoff" and surrounded by quad spaces, the third again is right justified, the fourth is simply left-justified. The result is shown on page 9 of the book, although with TEX the formula number "(1)" will be centered. Note: Eventually I will put in an "omittab" feature which will allow portions of a line to span several columns when desired.

Now let's look at lines 52-65 and compare with Table 1 on page 6 of the book. A box of width 12 picas is built up using tabalign, and placed beside another such box. The words "ruled" modifying vskip or hskip mean that the glue between boxes also appears with a horizontal or vertical ruled line in its center.

The ``eqalign'' feature (cf. lines 111, 116) is used to line up operators in different displayed formulas. Actually this is simply a special case of tabalign:

```
def eqalign #1 {tabalign{rjust##@ljust##\cr #1}}.
```

Note that line 113 begins with <TAB>.

The ``big(){...}'' in lines 113, 143, etc. and the ``big/{...}'' in line 152 is used to choose suitably large versions of parentheses and slashes, forming ``(...)' and ``/...', respectively; the size of the symbols is determined by the height of the enclosed formula box. This type of operation is available for [, <, ||, and left and right braces signified by \[and \]. TEX will provide the best size available in its repertoire. Parenthesis, brackets, braces, and vertical lines will be made in arbitrarily large sizes, but slashes will not, at least not in this year's TEX. Some very large parentheses will be generated for the big matrices in lines 158ff.

The ``biglpren'' and ``bigrpren'' on lines 118-119 are not really so big, they are simply 12-point parentheses instead of 10-point ones, used to set off the enclosed normal-size parentheses in the text's "(a)".

The summation signs produced by ``sum over...' in lines 131, 143, will be large, since these summations occur in displayed formulas; but between \$...\$ a smaller sign will be used and the quantity summed over will be attached as a subscript. Similarly, the size of an integral sign will change, and fractions ``...over...' do too, as does the binomial coefficient (cf. "\$p\$ choose \$k\$" in line 118). The keyword ``textsize' in lines 136 and 138 says to use smaller integral signs in this formula even though it is displayed.

The \ff in line 79 means to eject a page (top justified) before continuing. This again is part of the format of my books, a major section always should begin on a new page.

I think the above comments on the example give the flavor of TEX. The example was intended to show a variety of changing constructions of unusual complexity; in general, most of a book will be comparatively routine, and it is only the occasional formula or table which proves intricate. Such intricacies were purposely emphasized in the example.

The next step in understanding TEX is to go back to the low level and see what happens to an internal input file as it is being read in. What happens is that it is massaged once again, converted to a sequence of "tokens", which are either single characters or words. In general, adjacent letters and digits are grouped together to form words; also periods and commas immediately followed by digits are treated as letters, as are \ signs that are not preceded by letters or digits. More precisely, the following finite-state machine defines the token-building process.

```

State 0. Let x be the next character of the internal input file.
  If x is \ or one of the 52 letters or 10 digits, set w←x and go to state 1.
  If x is period or comma, set y←x, set w←null, go to state 2.
  Otherwise output x as the next token.
State 1. Let x be the next character of the internal input file.
  If x is a letter or digit, set w←w&x (w followed by x).
  If x is period or comma, set y←x and go to state 2.
  If x is \, output w as the next token, set w←x.
  Otherwise output w as the next token, then output x as the next token,
    and go to state 0.
State 2. Let x be the next character of the internal input file.
  If x is a digit, set w←w&y&x, go to state 1.
  Otherwise if w is nonempty, output w as the next token; then output y
    as the next token; then go to state 0 but without resetting x at the
    beginning of the instructions for that state.
```

For example, the sequence

```
Abc\de 5,000 plus-or-minus ``2.5 \per cent'' are ... here.
```

would be transformed into a sequence of 28 tokens:

```

Abc      \de      <space> 5,000    <space> plus    -        or        -
minus    <space> ` `      2.5      \per    <space> cent    '        '
<space> are      <space> .        .        .        <space> here    .
```

TEX works on sequences of tokens. If a token has been defined (using ``def''), a <space> before this token is removed. The remaining portion of the definition is plugged in; the same process is repeated on the resulting sequence of tokens. Definitions disappear after the } or \$ or \$\$ which closes the group they are in; they also are inoperative between \{ and its matching }. If the keyword MIX had been defined as

```
def MIX{{\tt MIX}}
```

instead of

```
def MIX{{\tt\{MIX}}},
```

TEX would have looped endlessly while emitting {\tt {\tt {\tt {\tt

By now the reader will probably understand everything or nothing about the definition process; but anyway I will try to spell out its rules more carefully. Once writes in general

```
def token string0 #1 string1 #2 string2 ... #k stringk {right-hand side}
```

where "token" is any token except { or }, and where string0 ... stringk are strings of zero or more tokens not including { or } or # or |; spaces are

ignored in these strings. I am describing the general case; the above definition of MIX is the simplest case of the general form, where `k` is zero and `string0` is empty.

When the defined token is recognized, a matching process ensues until the left-hand side is completely matched: tokens in `string0...stringk` must match exactly with corresponding tokens of the input text (removing all spaces), with error messages if a failure to match occurs. The `#j`'s are matched to tokens or to `{...}` groups. No expansion is done on the `{...}` groups, they are merely treated as an uninterrupted sequence of tokens. There is at most one definition active per token at any time. Once the matching process is complete, TEX will have found the ingredients to be substituted for `#1` through `#k` in the right-hand side. These are simply copied into the positions occupied by `#1 ... #k` in the right-hand sequence. And one further change is made: the first `#` is removed from any sequence of two or more `#`'s. This is done so that definitions can be made in the right-hand side without causing confusion with the current definition.

For example, consider what happens when the following definition is active:

```
def A #1 B C {def E ##1{##1 #1 #1}}.
```

If the internal input file contains

```
A {X-y} B C D
```

the resulting sequence after expanding the definition will be

```
def E #1{#1{X-Y}{X-Y}} D
```

(note the spacing).

If the character `|` appears just before the `{` which introduces the right-hand side of a definition, the space if any after the last token matched will be preserved. Thus, in the above example if we had written

```
def A #1 B C|{def E ##1{##1 #1 #1}}
```

the result would have been

```
def E #1{#1{X-Y}{X-Y}}D.
```

This feature was used in our definition of MIX on a previous page, because it will preserve the space that comes after MIX in mid-sentence; on the other hand, most definitions (for example, `xskip`) would not use this feature.

The above is not the most general sort of macro definition facility, but I have applied Occam's razor. A much more general capability is available via TEX "routines" which are explained later; routines are for the more experienced users. The reader should now be able to look back at the definitions given above for `quoteformat`, `algstep`, and `eqalign`, discovering that they are really quite simple after all.

Some tokens are not defined as macros but stand for actions to be carried out. For example, "`fnt`" means means the following non-space token is to be the name of the current font of type, and "`def`" means that TEX is supposed to absorb a new definition. Thus we have four mutually exclusive sorts of tokens:

```
defined tokens
action tokens
{ and }
other tokens.
```

The `{` and `}` denote grouping and separate levels of context; definitions and assignment actions made inside `{...}` do not have any effect outside.

Assignment actions affect TEX's transformation of the input. They are:

<code>fnt</code> token	to change the current font
<code>hsize</code> length	normal width of generated lines of text
<code>vsize</code> length	normal height of generated pages of text
<code>hmargin</code> length	distance from left edge of paper to generated lines
<code>vmargin</code> length	distance from top edge of paper to generated pages
<code>lineskip</code> glue	spacing between generated lines
<code>parskip</code> glue	additional spacing between paragraphs (added to <code>lineskip</code>)
<code>dispskip</code> glue	additional spacing above and below displayed formulas
<code>boxskip</code> glue	additional spacing below an inserted box
<code>noteskip</code> glue	additional spacing above footnotes
<code>tabskip</code> glue	horizontal spacing between tabaligned columns

parindent length indentation on first line of paragraphs
hangindent length indentation on all but first line of paragraph
(hangindent reset to zero after every paragraph)

All of these quantities will have default values, which I will choose after TEX is up and running; the default values will apply whenever a quantity has not been respecified. In the above, ``length'' is of the form

token unit or - token unit

where the token is a digit string or a digit string containing a period (decimal point), and where ``unit'' is either pt, pc, in, cm, or mm. Furthermore ``glue'' is of the form

length plus length minus length

where the plus and minus parts cannot be negative; any of the three lengths can be omitted if zero.

For example, standard XGP conventions at the moment are to produce 8 1/2 by 11 inch pages with one-inch margins and interline spacing of 4 pixels; this would be specified by

hsize 6.5 in vsize 9 in hmargin 1 in vmargin 1 in lineskip .02 in

At the outermost level, TEX has implicitly preceded the input by "TEXpublish{", where TEXpublish is a routine that repeatedly calls on the page-builder, the page-builder is a routine that repeatedly calls on the paragraph-builder, and the paragraph-builder is a routine that reads the input and emits strings of justified lines of text. TEXpublish simply prints each page that it gets; our example text input substitutes the more sophisticated routine ACPpages for TEXpublish.

In the first implementation of TEX, routines like ACPpages will be written in SAIL code and loaded with TEX. I have had some notion of adding an interpretive system to TEX and a mini-programming language so that such extensions are easier to make without penetrating into TEX's innards, but on second thought it seems much better to assume that TEX's code will be sufficiently clean that it will best be extended using SAIL code. This will save the implementors considerable time and make the system run faster.

Let's continue the outside-in approach by sketching the SAIL code for ACPpages. Remember that this is not basic TEX, but the code resembles the internal program of TEX. It is at the moment only an initial approximation to the real SAIL code, since I will have to think more about how TEX will look inside before I can be more specific.

```
if scan("starting") then
    begin scanreqd("at"); scanreqd("page"); spage-nextnonspacetoken;
    end else spage←"1";
```

{Here "scan" is a TEX routine which looks at the next nonspace token from the input; if it equals the specified token, scan returns true and discards the token, otherwise the current input token is not passed over and false is returned. The routine "scanreqd" is similar, but instead of returning false it will produce an error message like "Syntax error, I will insert missing #". The net result of the above code is to set spage to the string representing the specified starting page, or by default to set it to "1".}

```
if spage = "r" then
    begin rnumeral←true; lop(spage);
    end;
```

```
pageno←intscan(spage, brchar);
```

{Roman numeral page numbers are indicated by r1, r2, etc. Now pageno is the integer number of the desired first page and rnumeral is true if and only if roman numeral page numbers are desired.}

```
scanreqd(leftbrace);
put_on_stack_something_to_make_matching_right_brace_end_the_input;
while true do
```

```
    begin cpage←nextpage;
    if not cpage then done;
```

{Here cpage is a pointer to a box or the null record pointer when the input has terminated.}

```
    if rnumeral then spage←cvrom(pageno) else spage←cvs(pageno);
    if omithead then {this would be set by "titlepage" routine}
```

```

begin omithead←false;
output_cpage_with_blanks_for_top_headline_and_with_spage_in_
9_point_type_centered_as_a_new_line_at_the_bottom;
end
else begin if pageno land 1 then {right-hand page}
begin texta←the_running_right_head;
textb←attr(sectno,cpage);
{texta and textb are record pointers to sequences of tokens; "attr" gets the
sectno attribute of box cpage, as explained below}
line←pointer_to_box_which_TEX_would_make_from_the_input
"{textb} hfill {texta} rjust to .375 in {spage}"
end else
begin texta←the_running-left-head;
textb←attr(sectno,first(cpage));
line←pointer_to_box_which_TEX_would_make_from_the_input
"ljust to .375 in {spage} texta hfill textb"
end;
place_line_above_cpage_with_14_pt_glue_and_output_the_result;
end;
pageno←pageno+1;
end;

```

{In other words, odd-numbered pages get the sectionnumber attribute of cpage and the running right headline followed by a right-justified page number, as the title line; even-number pages get a left-justified page number followed by the running left headline followed by the sectionnumber attribute of the first component of cpage. Note that the section number is supposed to represent the top of a left-hand page but the bottom of a right-hand page}

The above example indicates that we need to discuss another basic feature of TEX, namely the "attributes" which can be attached to its boxes. One such attribute is the section number; another is the height, another is the width; still others, used in math formulas, are the amounts of space available inside the box, if any, that may be used to attach subscripts and superscripts. For each attribute, two routines are supplied, explaining how to produce the attribute of a box formed from two boxes placed together horizontally or vertically. For example, the height attribute is replaced by $\max(h_1, h_2)$ when two boxes are joined horizontally but by $h_1 + h_2$ when they are joined vertically. The section-number attribute is s_1 if s_2 is null, otherwise it is s_2 ; and so on.

Now let's consider the page-building routine "nextpage"; this gives us a chance to study the process TEX uses for vertical justification, which introduces some of the concepts we will need in the more complicated routine used for horizontal justification.

The first idea is the concept of "badness." This is a number computed on the basis of the amount of stretching or shrinking necessary when setting the glue. Suppose we have a list of n boxes (either a horizontal list or a vertical list), separated by $n-1$ specifications of glue. The w be the desired total length of the list (i.e., the desired width or height, depending on which dimension we are justifying); let x be the actual total length of boxes and glue; and let y, z be the total amount of glue parameters for expansion and contraction. The badness of this situation is defined to be

```

infinite, if  $x > w - z + \epsilon$ , where  $\epsilon$  is a small tolerance to compensate
for floating-point rounding;
 $100((x-w)/z) \uparrow 3$ , if  $w - z + \epsilon \geq x > w$ ;
0, if  $x = w$ ;
 $100((w-x)/3y) \uparrow 3$ , if  $w > x$ ;

```

plus penalties charged for breaking lines in comparatively undesirable places.

According to these formulas, stretching by y has a badness rating of $100/27$, or about 3.7; stretching by $2y$ is rated about 30; stretching by $3y$ is rated 100 units of badness, and so is shrinking by the maximum amount z . I plan to charge a penalty of something like 80 units for breaking a paragraph or displayed formula in such a way that only one line comes by itself on a page; thus, for instance, a five-line paragraph gets a penalty of 80 if we break after the first or fourth line, but no penalty if we break after two or three

lines. I will of course be tuning these formulas to make them correspond as well as I can to me aesthetic perceptions of bad layout. The user will be able to specify his own additional penalty points for undesirable breaking between specific lines (e.g. in a MIX program to break before an instruction that refers to *-1). A penalty is also made for break at "ruled" vertical glue.

The nextpage routine forms a list a lines (and the connecting vertical glue) which it receives from the nextparagraph routine; the nextparagraph routine returns a sequence g1 h1 ... gk hk alternating between vertical glue and boxes representing horizontally justified lines of text. The individual h's are not broken up or examined by nextpage, except to look at their attributes. Also \ff and end-of-input codes will be transmitted by nextparagraph to the nextpage routine.

The nextpage routine accumulates lines until its previous accumulation plus the new paragraph is greater than or equal to the specified page height, vsize. Then it breaks the new paragraph just after the jth line, for $0 \leq j \leq k$, whichever value of j has the minimum badness; if this minimum occurs for more than one j, the largest such j is used. Then the glue g(j+1) is discarded, and the remaining k-j lines are carried over to the next page. (They are immediately checked to ensure that they don't already overfill the new page, and they are broken in the same way if necessary.)

A boxinsert interrupts this otherwise straightforward procedure. The box to be inserted is computed, off to the side, and then an attempt is made to place it over the current accumulated page. If it fits, well and good, we leave it there. If not, it is carried over to the next page, and placed in front of any carryover from the present page. Additional box-inserts are inserted below the first one, in a natural but hard-to-explain manner.

Footnotes: I have used footnotes only three times in over 2000 pages of The Art of Computer Programming, and personally I believe they should usually be avoided, so I am not planning an elaborate footnoting mechanism (e.g. to break long footnotes between pages or to mark the first footnote per page with an asterisk and the second with a dagger, etc.). They can otherwise be satisfactorily handled by attaching a footnote attribute to any line referring to a footnote, and by requiring the nextpage routine to put the footnote on the page containing that line. This, together with the badness ratings, will solve the problem about footnote references on the bottom line preempting the space needed for the footnote itself, etc. A user will be able to get fancier footnotes if he doesn't mind rewriting a few of TEX's routines.

On \ff or end-of-input, the nextpage routine simulates "vfill blankline" and clears its page buffers, unless they were already empty. After end-of-input it returns a null page, to signal the end of its activity; after \ff it asks nextparagraph for more.

The nextparagraph routine assembles lines of width hsize, pausing to transmit them

- a) at end of paragraph ('13);
- b) at beginning of display formula (\$\$);
- c) at end of display formula (\$\$);
- d) just before vskip operation;
- e) just before and after blankline, ctrlline, ljustline, rjustline, hjustline;
- f) at \ff or end-of-input.

The operations in (e) produce independent lines of width hsize that are not part of a paragraph, and such lines are merely transmitted without change. Display formulas are also passed without change, except that appropriate glue is attached above and below them. (The glue above a displayed equation is reduced from dispskip to lineskip if the text on the preceding line does not overhand the displayed formula after centering.)

The text of a paragraph, or of the portion of a paragraph beginning and/or ending at \$\$, is indented (if appropriate) and passed to the hjust routine. The hjust routine attempts to break the text into lines of length hsize in the least bad way, taking proper account of hangindent. The "least bad" way is a way which

minimizes the maximum badness of any of the breaks made. At this point in the process, the text consists of a possibly long sequence of boxes, separated by glue, and these boxes will not be changed by the hjust routine. The hjust routine has a new feature which makes it superior to existing justification systems, besides the idea of variable-weight glue, namely it effectively "looks ahead" so that the situation in the later lines of a paragraph can influence the breaks in the earlier lines. I have noticed quite a few places where such a justification routine will provide substantially better output. This lookahead is accomplished by applying the principles of "dynamic programming," as I will try to explain below.

First let's understand what the boxes precessed by hjust usually look like. They might be large complicated boxes, which are to be treated as black boxes essentially as the nextpage routine treats its lines; but mostly the individual boxes at this point will be individual letters from one or more fonts. When a text token comes along, the current font indicator is attached to it (so that the 7-bit code becomes a 12-bit code), and the token is broken into its individual characters. Pairs of adjacent characters are examined, from left to right, using tables associated with the font; this is used to compute changes in the glue between characters, and to produce ligatures if they are present, etc. For example, on most fonts the sequence `` will want to have specially narrow glue between the two characters. We get ligatures by replacing

f f	by	<ff>
f i	by	<fi>
f l	by	<fl>
<ff> i	by	<ffi>
<ff> l	by	<ffl>.

Such ligature formation and glue adjustment takes place when the character boxes are being appended to the current line, before hyphenation etc.; this means that we lose the chance to break "shuffling" into "shuff-ling", but so what.

The <space> token is changed into glue between boxes, with $y=x$ and $z=x/2$ as explained earlier. The hskip and quad actions also produce variable glue. Whenever this glue has x or y greater than or equal to the spacing width of the current font, it is an acceptable place to break the line with no penalty. Another acceptable place is after an explicit hyphen. (Some hskips, used for backspacing, have negative x ; they are, of course, unacceptable breaks.) TEX will give double y glue to the first space that follows a period, exclamation point, question mark, or colon, unless letters or digits intervene before this space. A semicolon and comma are treated similarly, but with 1.5 and 1.25 as the relative amounts of y glue.

The math formula routine which processes $\$...\$$ will yield a sequence of boxes in format compatible with the text strings outside of formulas; acceptable places to break in formulas will be marked just after binary operators and relations. A penalty is attached to such breaks; I will have to tune the parameters, but the following idea appears to be best: Relations like $=$, $\leq$, $\equiv$, etc. have only a small penalty, operations like $+$, $-$, $\times$, $/$ have a larger penalty, with $-$ larger than the others. Superscripts and subscripts and function arguments and the like will be attached unbreakably to their boxes.

There are three "discretionary" symbols used to provide or inhibit breaks:

\-	OK to hyphenate this word here;
\+	do not hyphenate here;
*	OK to break here, but insert a times sign not a hyphen.

The last of these would be used in a long product like $\$(n+1)*(n+2)*(n+3)*(n+4)\$$. Besides using the permissible breaks, TEX will try to hyphenate words.

It will do this only in a sequence of lower-case letters that is preceded and followed by anything other than a letter, digit, $-$, $\backslash-$, or $\backslash+$. Note that, for example, already-hyphenated compound words will not be broken. If a permissible hyphenation break is discovered, a penalty of say 30 units of badness will be paid, but this may be better than not hyphenating at all. An additional 20 units of badness is charged for breaking a word or formula in the last line of a paragraph.

There is no point in finding all possible places to hyphenate. For one thing, the problem is extremely difficult, since e.g. the word "record" is supposed to be broken as "rec-ord" when it is a noun but "re-cord" when it is a verb. Consider the word "hyphenation" itself, which is rather an exception:

hy-phen-a-tion vs. co-or-di-na-tion

Why does the n go with the a in one case and not the other? Starting at letter a in the dictionary and trying to find rigorous rules for hyphenation without much knowledge, we come up against a-part vs. ap-er-ture, aph-o-rism vs. a-pha-sia, etc. It becomes clear that what we want is not an accurate but ponderously slow routine that consumes a lot of memory space and processing time, instead we want a set of hyphenation rules that are

- a) simple;
- b) almost always safe;
- c) powerful enough to find say 80% of the words already hyphenated in The Art of Computer Programming.

To justify point (c), I find that there are about 2 hyphenated words per page in the books, and the places where the rules I shall propose do not find the identical hyphenation only very rarely would cause a really bad break. The time needed to handle the remaining exceptions is therefore insignificant by comparison with what I already do when proof-reading.

So here are the rules I came up with.

1. Consider only breaks in which both parts contain a vowel other than final e. (Vowels are a,e,i,o,u,y.)
2. Subject to 1, break before the following "safe" suffixes:
-cious -gion -ly -ment -mial -nary -ness -nomial -sion -tion -ture -vious
and also -tive preceded by a vowel, -ed preceded by d or t.
Break before -ing except in -bling -pling or when preceded by a double letter other than ll or ss or zz; for other double letters, break between them.
If the word ends in s or ly, strip off this ending and apply these rules again.
Suffixes recognized by this rule must not be further broken except vi-ous.
3. Subject to 1 and 2, break after the following "safe" prefixes:
algo- equi- ex- hyper- ini- intro- lex- lexi- math- mathe- max- maxi- mini- multi-
out- over- pseudo- semi- some- sub- super- there- under-
Also be- followed by c,h,s,w; dis- followed by anything but h,y; trans- followed by a,f,g,l,m; tri- followed by a,f,p,u.
4. Subject to 1 and 2, combine an h with the previous letter if it is a consonant, treating the combination as a consonant; then it's OK to break between the two consonants in the pattern vc-cv except when the pair of consonants is
bl ck cl cr dr ght gl gr lk ll nd ng pl rch rd rm rt sch sp ss st thr zz
(I may have to revise this list.)

There will be rare exceptions (e.g., equivocate, minister, somersault, triphammer), but these will be so infrequent as to be unimportant. Looking through quite a few pages of volume 3, I found 48 hyphenations, and the above rules were satisfactory except in the three cases

de-gree hap-hazard re-placement.

Of course, these rules are biased toward my vocabulary and subject matter, but a few extensions should make it work well enough for nearly everybody.

Now for the dynamic programming algorithm which minimizes the maximum badness of breaks. The badness formula is the same as before; thus for each break after a given point we can compute the resulting badness. In this way, for each permissible break position, we will minimize the maximum badness that can occur counting this and previous breaks. The running time will be approximately linear in the number of possible breaks.

However, this will be rather slow since it tries breaking all words when in practice only a few candidate words really need to be considered. Thus the following approximate method should run about ten times faster: Given the best three ways to break the kth line, use these to find the best three ways to break the (k+1)st line. The breaks found will almost certainly be optimum unless the situation is very bad anyway.

To conclude this memo, I should explain how TEX is going to work on

math formulas. However, I will have to sketch out the code in more detail and it is only fuzzy in my mind at the moment. My plan is to use the top-down precedence parsing approach introduced by Vaughan Pratt in his 1973 POPL paper, and used so successfully by him in CGOL (for example). Further details will be forthcoming; I hope to do the box pasting in one pass, but it may be necessary to build the parse tree first as Kernighan and Cherry do.