Problem Set 5 Solutions

Reading: Chapters 15, 16

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

Three-hole punch your paper on submissions.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct algorithms which are *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 4-1. Do Exercise 15.2-1 on page 338 in CLRS.

Exercise 4-2. Do exercise 15.3-4 on page 350 in CLRS.

Exercise 4-3. Do exercise 15.4-4 on page 356 in CLRS and show how to reconstruct the actual longest common subsequence.

Exercise 4-4. Do exercise 16.1-3 on page 379 in CLRS.

Exercise 4-5. Do exercise 16.3-2 on page 392 in CLRS.

Problem 4-1. Typesetting

In this problem you will write a program (real code that runs!!!) to solve the following typesetting problem. **Because of the trouble you may encounter while programming, we advise you to START THIS PROBLEM AS SOON AS POSSIBLE.**

You have an input text consisting of a sequence of n words of lengths $\ell_1, \ell_2, \ldots, \ell_n$, where the length of a word is the number of characters it contains. Your printer can only print with its built-in Courier 10-point fixed-width font set that allows a maximum of M characters per line. (Assume that $\ell_i \leq M$ for all $i = 1, \ldots, n$.) When printing words i and $i+1$ on the same line, one space character (blank) must be printed between the two words. Thus, if words i through j are printed on a line, the number of extra space characters at the end of the line—that is, after word j—is $M - j + i - \sum_{k=i}^{j} \ell_k$.

To produce nice-looking output, the heuristic of setting the cost to the square of the number of extra space characters at the end of the line has empirically shown itself to be effective. To avoid the unnecessary penalty for extra spaces on the last line, however, the cost of the last line is 0. In other words, the cost *linecost*(i, j) for printing words i through j on a line is given by

$$\textit{linecost}(i,j) = \begin{cases} \infty & \text{if words } i \text{ through } j \text{ do not fit into a line,} \\ 0 & \text{if } j = n \text{ (i.e. last line),} \\ \left(M - j + i - \sum_{k=i}^{j} \ell_k\right)^2 & \text{otherwise.} \end{cases}$$

The total cost for typesetting a paragraph is the sum over all lines in the paragraph of the cost of each line. An optimal solution is an arrangement of the n words into lines in such a way that the total cost is minimized.

(a) Argue that this problem exhibits optimal substructure.

Solution: First, notice that $linecost(i, j)$ is defined to be ∞ if the words i through j do not fit on a line to guarantee that no lines in the optimal solution overflow. (This relies on the assumption that the length of each word is not more than M.) Second, notice that $linecost(i, j)$ is defined to be 0 when $j = n$, where n is the total number of words; only the actual last line has zero cost, not the recursive last lines of subproblems, which, since they are not the last line overall, have the same cost formula as any other line.

Consider an optimal solution of printing words 1 through n. Let i be the index of the first word printed on the last line of this solution. Then typesetting of words $1, \ldots, i-1$ must be optimal. Otherwise, we could paste in an optimal typesetting of these words and improve the total cost of solution, a contradiction. Please notice that the same *cut-and-paste* argument can be applied if we take i to be the index of the first word printed on the kth line, where $2 \leq k \leq n$. Therefore this problem displays optimal substructure.

(b) Define recursively the value of an optimal solution.

Solution: Let $c(j)$ be the optimal cost of printing words 1 through j. From part (a), we see that given the optimal i (i.e., the index of the first word printed on the last line of an optimal solution), we have $c(j) = c(i-1) + linecost(i, j)$. But since we do not know what i is optimal, we need to consider every possible i, so our recursive definition of the optimal cost is

$$c(j) = \min_{1 \le i \le j} \{c(i-1) + linecost(i,j)\} \ .$$

To accommodate this recursive definition, we define $c(0) = 0$.

(c) Describe an efficient algorithm to compute the cost of an optimal solution.

Solution: We calculate the values of an array for c from index 1 to n, which can be done efficiently since each $c(k)$ for $1 \le k < j$ will be available by the time $c(j)$ is computed. To keep track of the actual optimal arrangement of the words, we record an array p, where $p(k)$ is the i (in the recursive definition of c) which led to the optimal $c(k)$. Then, after the arrays for c and p are computed, the optimal cost is $c(n)$ and the optimal solution can be found by printing words $p(n)$ through n on the last line, words $p(p(n)-1)$ through $p(n)-1$ on the next to last line, and so on.

Notice that computing $linecost(i,j)$ takes in general $O(j-i+1)$ time because of summation in the formula. However, it is possible to do this computation in $O(1)$ time with some additional pre-processing. We create an auxillary array $L[0\ldots n]$, where $L[i]$ is a cumulative sum of lengths of words 1 thru i.

$$\begin{aligned} L[0] &= 0 \\ L[i] &= L[i-1] + \ell_i = \sum_{k=1}^{i} \ell_k \end{aligned}$$

Filling in this array takes $O(n)$ time using recursion. Using the auxillary array we compute $linecost(i,j)$ in $O(1)$ time according to this formula:

$$\textit{linecost}(i,j) = \begin{cases} \infty & \text{if words } i \text{ through } j \text{ do not fit into a line,} \\ 0 & \text{if } j = n \text{ (i.e. last line),} \\ (M - j + i - (L[j] - L[i-1]))^2 & \text{otherwise.} \end{cases}$$

This algorithm uses $\Theta(n)$ space for the arrays and runs in $O(n^2)$ time, since each value of c takes up to n calculations as each value of i is considered. By noticing that at most $\lfloor (M+1)/2 \rfloor$ words can fit on a single line, we can reduce running time to $O(nM)$—a significant improvement—by considering only those i for which $j - \lfloor (M+1)/2 \rfloor + 1 \le i \le j$ when calculating each $c(j)$.

(d) Write code (in any language you wish—even Visual Java ++ :-)[1]) to print an optimal arrangement of the words into lines. For simplicity, assume that a word is any sequence of characters not including blanks—so a word is everything included between two space characters (blanks).

[1]The solution will be written using C.

(d) requires 5 parts: you should turn in the code you have written, and the output of your program on the two input samples using two values of M (the maximum number of characters per line), namely $M = 72$ and $M = 40$, on each input sample.

Sample 1 is from *A Capsule History of Typesetting* by Brown, R.J. Sample 2 is from *Out of Their Minds*, by Shasha, Lazere. Remember that collaboration, as usual, is allowed to solve problems, but you must write your program by yourself.

```
/* NOTE: This is an implementation of the O(nM) algorithm. */
/* DISCLAIMER: No effort has been made to streamline memory */
/* management or micro-optimize performance. */

/* standard header files */
#include <stdio.h>
#include <limits.h>

/* arbitrary data size limits, so no dynamic allocation needed */
#define WORD_NUM 1024 /* arbitrary max for number of input words */
#define WORD_LENGTH 32 /* arbitrary max for length of input words */
#define LINE_LENGTH 80 /* arbitrary max for length of output lines */

/* macros */
#define max(A, B) ((A) > (B) ? (A) : (B))

/* global array of words */
char words[WORD_NUM+1][WORD_LENGTH]; /* array for input words */
int auxL[WORD_NUM+1];    /* auxillary array for computing lengths
                                     of lines - MM*/

/* function prototypes */
long linecost(int n, int M, int i, int j);
long dynamic_typeset(int n, int M, int p[]);

/* main expects two arguments: the input file name and M */
int main (int argc, char *argv[]) {
  FILE *ifile; /* input file */
  int p[WORD_NUM]; /* array of how to get min costs */
  char lines[WORD_NUM+1][LINE_LENGTH]; /* buffer for output lines */
  int M; /* output line length */
  int n; /* number of input words */
  char read_word[WORD_LENGTH]; /* for use during reading */
  int i, j, k, l; /* aux vars used during construction of solution */

  /* verify arguments */
  if(argc != 3) /* verify number of arguments */
    exit(1);
  if(!(ifile = fopen(argv[1], "r"))) /* open input file */
    exit(2);
  if(!sscanf(argv[2], "%d", &M)) /* get length of output line */
    exit(3);

  /* read input words */
  n = 1;
  while(!feof(ifile)) {
    if(1 == fscanf(ifile, "%s", read_word)) { /* assumes input word fits */
      strcpy(words[n++], read_word);
      if(n == WORD_NUM)
        break; /* no more room for words */
    }
  }
  n--;
  /*fill in auxillary array of word lengths */
  auxL[0] = 0;
  for(k = 1; k <= n; k++)
    auxL[k] = auxL[k-1] + strlen(words[k]);


  /* compute and output min cost */
  printf("COST = %ld\n", dynamic_typeset(n, M, p));

  /* construct and output the actual solution */
  j = n; /* start at last line and work backwards */
  l = 0;
  do {
    l++;
    lines[l][0] = 0; /* line starts off as empty string */
    for(i = p[j]; i <= j; i++) { /* words i..j make up a line */
      strcat(lines[l], words[i]);
      strcat(lines[l], " "); /* space in between words */
    }
    j = p[j] - 1; /* recurse ... */
```

```
    /* ... and construct next line */
  }
  while(j != 0); /* just finished first line */

  for(i = l; i > 0; i--) /* output lines in right order */
    printf("%d:[%d]\t%s\n", l-i+1, strlen(lines[i])-1, lines[i]);
}

/**** algorithmic part *****/

/* returns min cost and a min solution in p[] */
long dynamic_typeset(int n, int M, int p[]) {
  int i, j;
  /* need an extra space for c[0], so c is indexed from 1 to n, */
  /* instead of from 0 to n-1 (like p) */
  long c[WORD_NUM+1];
  c[0] = 0; /* base case */
  for(j = 1; j <= n; j++) { /* fill in c[] bottom-up */
    c[j] = LONG_MAX;
    /* find min i (only look at the O(M/2) possibilities) */
    for(i = max(1, j+1-(M+1)/2); i <= j; i++) {
      long lc = linecost(n, M, i, j), cost = c[i-1] + lc;
      if(lc > -1 && cost < c[j]) {
        c[j] = cost; /* record the cost (c indexed from 1) */
        p[j] = i; /* record the min i (p indexed from 0) */
      }
    }
  }
  return c[n]; /* min cost of all n words */
}

/* compute cost of a single line, i and j indexed from 0 */
long linecost(int n, int M, int i, int j) {
  int k;
  long extras = M - j + i; /* compute number of extra spaces */
  extras -= ( auxL[j] - auxL[i-1] );
  if(extras < 0)
    return -1; /* signal infinity */
  else if(j == n) /* last line (non-recursive defn of last line) */
    return 0;
  else
    return extras*extras; /* assumes result fits in a long */
```

```
}
```

Solutions:

```
sample1 72
COST = 160
1:[67]  The first practical mechanized type casting machine was invented in
2:[69]  1884 by Ottmar Mergenthaler. His invention was called the "Linotype".
3:[72]  It produced solid lines of text cast from rows of matrices. Each matrice
4:[70]  was a block of metal -- usually brass -- into which an impression of a
5:[69]  letter had been engraved or stamped. The line-composing operation was
6:[72]  done by means of a keyboard similar to a typewriter. A later development
7:[64]  in line composition was the "Teletypewriter". It was invented in
8:[70]  1913. This machine could be attached directly to a Linotype or similar
9:[66]  machines to control composition by means of a perforated tape. The
10:[70] tape was punched on a separate keyboard unit. A tape-reader translated
11:[70] the punched code into electrical signals that could be sent by wire to
12:[71] tape-punching units in many cities simultaneously. The first major news
13:[56] event to make use of the Teletypewriter was World War I.
sample1 40
COST = 360
1:[35]  The first practical mechanized type
2:[36]  casting machine was invented in 1884
3:[37]  by Ottmar Mergenthaler. His invention
4:[38]  was called the "Linotype". It produced
5:[37]  solid lines of text cast from rows of
6:[37]  matrices. Each matrice was a block of
7:[36]  metal -- usually brass -- into which
8:[34]  an impression of a letter had been
9:[39]  engraved or stamped. The line-composing
10:[32] operation was done by means of a
11:[35] keyboard similar to a typewriter. A
12:[37] later development in line composition
13:[32] was the "Teletypewriter". It was
14:[36] invented in 1913. This machine could
15:[37] be attached directly to a Linotype or
16:[39] similar machines to control composition
17:[39] by means of a perforated tape. The tape
18:[40] was punched on a separate keyboard unit.
19:[36] A tape-reader translated the punched
20:[39] code into electrical signals that could
21:[38] be sent by wire to tape-punching units
22:[40] in many cities simultaneously. The first
23:[35] major news event to make use of the
24:[31] Teletypewriter was World War I.
sample2 72
COST = 229
1:[65]  Throughout his life, Knuth had been intrigued by the mechanics of
2:[70]  printing and graphics. As a boy at Wisconsin summer camp in the 1940s,
3:[71]  he wrote a guide to plants and illustrated the flowers with a stylus on
```

```
4:[69]  the blue ditto paper that was commonly used in printing at that time.
5:[71]  In college, he recalls admiring the typeface used in his math texbooks.
6:[71]  But he was content to leave the mechanics of designing and setting type
7:[72]  to the experts. "I never thought I would have any control over printing.
8:[71]  Printing was done by typographers, hot lead, scary stuff. Then in 1977,
9:[71]  I learned about new printing machines that print characters made out of
10:[69] zeros and ones, just bits, no lead. Suddenly, printing was a computer
11:[71] science problem. I couldn’t resist the challenge of developing computer
12:[66] tools using the new technology with which to write my next books."
13:[67] Knuth designed and implemented TeX, a computer language for digital
14:[67] typography. He explored the field of typography with characteristic
15:[68] thoroughness. For example, he wrote a paper called "The letter S" in
16:[67] which he dissects the mathematical shape of that letter through the
17:[67] ages, and explains his several day effort to find the equation that
18:[33] yields the most pleasing outline.
sample2 40
COST = 413
1:[35]  Throughout his life, Knuth had been
2:[38]  intrigued by the mechanics of printing
3:[35]  and graphics. As a boy at Wisconsin
4:[36]  summer camp in the 1940s, he wrote a
5:[35]  guide to plants and illustrated the
6:[39]  flowers with a stylus on the blue ditto
7:[40]  paper that was commonly used in printing
8:[36]  at that time. In college, he recalls
9:[38]  admiring the typeface used in his math
10:[37] texbooks. But he was content to leave
11:[38] the mechanics of designing and setting
12:[37] type to the experts. "I never thought
13:[39] I would have any control over printing.
14:[34] Printing was done by typographers,
15:[36] hot lead, scary stuff. Then in 1977,
16:[37] I learned about new printing machines
17:[33] that print characters made out of
18:[35] zeros and ones, just bits, no lead.
19:[33] Suddenly, printing was a computer
20:[38] science problem. I couldn’t resist the
21:[38] challenge of developing computer tools
22:[38] using the new technology with which to
23:[36] write my next books." Knuth designed
24:[40] and implemented TeX, a computer language
25:[39] for digital typography. He explored the
26:[39] field of typography with characteristic
27:[37] thoroughness. For example, he wrote a
28:[39] paper called "The letter S" in which he
29:[39] dissects the mathematical shape of that
30:[37] letter through the ages, and explains
31:[34] his several day effort to find the
32:[38] equation that yields the most pleasing
33:[8]  outline
```

Here is what Sample 1 should look like when typeset with $M = 50$. Feel free to use this output to debug your code.

```
The first practical mechanized type casting
machine was invented in 1884 by Ottmar
Mergenthaler. His invention was called the
"Linotype". It produced solid lines of text
cast from rows of matrices. Each matrice was a
block of metal -- usually brass -- into which
an impression of a letter had been engraved or
stamped. The line-composing operation was done
by means of a keyboard similar to a typewriter.
A later development in line composition was
the "Teletypewriter". It was invented in
1913. This machine could be attached directly
to a Linotype or similar machines to control
composition by means of a perforated tape. The
tape was punched on a separate keyboard unit.
A tape-reader translated the punched code into
electrical signals that could be sent by wire to
tape-punching units in many cities simultaneously.
The first major news event to make use of the
Teletypewriter was World War I.
```

(e) Suppose now that the cost of a line is defined as the number of extra spaces. That is, when words i through j are put into a line, the cost of that line is

$$\textit{linecost}(i,j) = \begin{cases} \infty & \text{if words } i \text{ through } j \text{ do not fit into a line,} \\ 0 & \text{if } j = n \text{ (i.e. last line),} \\ M - j + i - \sum_{k=i}^{j} \ell_k & \text{otherwise;} \end{cases}$$

and that the total cost is still the sum over all lines in the paragraph of the cost of each line. Describe an efficient algorithm that finds an optimal solution in this case.

Solution: We use a straightforward greedy algorithm, which puts as many words as possible on each line before going to the next line. Such an algorithm runs in linear time.

Now we show that any optimal solution has the same cost as the solution obtained by this greedy algorithm. Consider some optimal solution. If this solution is the same as the greedy solution, then we are done. If it is different, then there is some line i which has enough space left over for the first word of the next line. In this case, we move the first word of line $i+1$ to the end of line i. This does not change the total cost, since if the length of the word moved is l, then the reduction to the cost of line i will be $l + 1$, for the word and the space before it, and the increase of the cost of line $i + 1$ will also be $l + 1$, for the word and the space after it. (If the moved word was the only word on line $i + 1$, then by moving it to the previous line the total cost is reduced, a contradiction to the supposition that we have an optimal solution.) As long as there are lines with enough extra space, we can keep moving the first words of the next lines back without changing the total cost. When there are no longer any such lines, we will have changed our optimal solution into the greedy solution without affecting the total cost. Therefore, the greedy solution is an optimal solution.

Problem 4-2. Manhattan Channel Routing

A problem that arises during the design of integrated-circuit chips is to hook components together with wires. In this problem, we'll investigate a simple such problem.

In ***Manhattan routing***, wires run on one of two layers of an integrated circuit: vertical wires run on layer 1, and horizontal wires run on layer 2. The height h is the number of horizontal tracks used. Wherever a horizontal wire needs to be connected to a vertical wire, a ***via*** connects them. Figure 1 illustrates several ***pins*** (electrical terminals) that are connected in this fashion. As can be seen in the figure, all wires run on an underlying grid, and all the pins are collinear.

In our problem, the goal is to connect up a given set of pairs of pins using the minimum number of horizontal tracks. For example, the number of horizontal tracks used in the routing channel of Figure 1 is 3 but fewer might be sufficient.

Let $L = \{(p_1, q_1), (p_2, q_2), \ldots, (p_n, q_n)\}$ be a list of pairs of pins, where no pin appears more than once. The problem is to find the fewest number of horizontal tracks to connect each pair. For example, the routing problem corresponding to Figure 1 can be specified as the set $\{(1, 3), (2, 5), (4, 6), (8, 9)\}$.

(a) What is the minimum number of horizontal tracks needed to solve the routing problem in Figure 1?

Solution: You can verify that the wire connecting pins 4 and 6 could be at the same height as the wire connecting pins 1 and 3, making the number of horizontal track needed 2. Note that this is the minimum possible. Otherwise the wire connecting pins 2 and 3 and the wire connecting 1 and 3 would be on the same track, violating the problem specifications.

(b) Give an efficient algorithm to solve a given routing problem having n pairs of pins using the minimum possible number of horizontal tracks. As always, argue correctness (your algorithm indeed minimizes the number of horizontal tracks), and analyze the running time.

Algorithm description

The following algorithm routes pin pairs greedily into available horizontal tracks in order of the smaller pin of a pair.

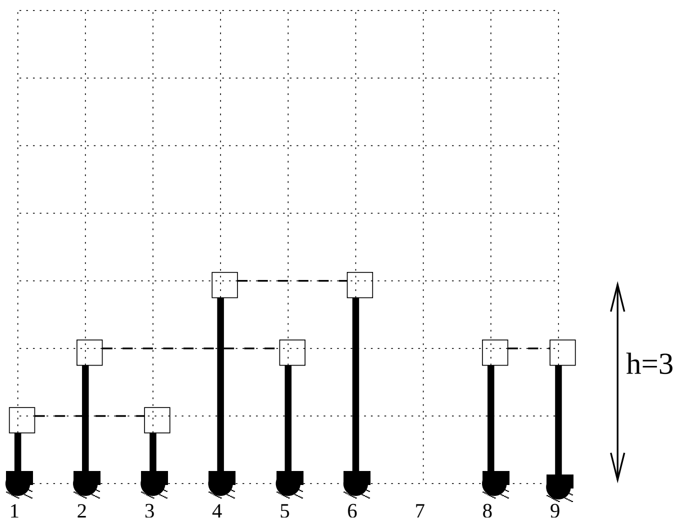

Figure 1: Pins are shown as circles. Vertical wires are shown as solid. Horizontal wires are dashed. Vias are shown as squares.

1. Go through L and if $q_i > p_i$ swap them. For each pair, we call the smaller pin, start, and the larger, end. Sort the start and end pins of the pairs in an increasing order. The resulting list contains $2n$ values.
2. Place all available horizontal tracks in a stack S.
3. Go through the list in sorted order.
   - If the current pin is a start pin, pop the first available horizontal track from S and route it in that horizontal track. If S is empty, then it is not possible to route all of the pin pairs using the given number of horizontal tracks. Report an error in this case.
   - If the current pin is an end pin, look up in which horizontal track it has been routed and push that horizontal track back onto S.

Correctness

Suppose that the algorithm terminates with a routing requiring m horizontal tracks. Let k denote the first pair of pins routed in the mth horizontal track. Let s_k denote the start pin and f_k denote the end pin of this pair. Let f_l be the earliest finish pin appearing in the sorted list after s_k. Necessarily, $f_l > s_k$. The closest routing in each of the $m - 1$ horizontal tracks already in use starts before s_k. Each routing terminates after f_l. Thus there are m routings in between $[s_k, f_l]$, i.e., any routing must use at least m vertical tracks. Thus the routing returned by the algorithm is optimal.

The above argument shows that given an infinite supply of horizontal tracks, our algorithm will always produce a routing that uses the fewest number of horizontal tracks. Thus, if the algorithm terminates with an error, it means that a given number of horizontal tracks is less than the number of horizontal tracks in an optimal routing, and hence it is impossible to route all the pin pairs.

Analysis

This algorithm runs in $O(n \lg n)$ because it is necessary to sort $2n$ items (which can be accomplished using heapsort or mergesort). Notice that scanning through the list and assigning horizontal tracks takes $O(1)$ time per connection, for a total of $O(2n) = O(n)$ time.

This is also known as the ***interval-graph coloring problem***. We can create an interval graph whose vertices are the given pairs of pins and whose edges connect incompatible pairs of pins. The smallest number of colors required to color every vertex so that no two adjacent vertices are given the same color corresponds to finding the fewest number of horizontal tracks needed to connect all of the pairs of pins.)