Problem Set 8 Solutions

MIT students: This problem set is due in lecture on Wednesday, November 14.

SMA students: This problem set is due after the video-conferencing session on Wednesday, November 14.

Reading: Chapter 15, 16, 23

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 by the exercises.

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

MIT students: Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.

SMA students: Each problem should be done on a separate sheet (or sheets) of two-hole punched paper.

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 your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
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. Graders will be instructed to take off points for convoluted and obtuse descriptions.
Exercise 8-1. Do exercise 15.4-4 on page 356 of CLRS

Solution:

When computing a particular row of the c-table, only the previous row is needed. Only two rows need to be kept in memory at a time.

\[ m = \text{length}[X], \ n = \text{length}[Y]. \] If \( m < n \) compute the LCS-LENGTH\((Y, X)\) otherwise compute LCS-LENGTH\((X, Y)\). In either case, use the method given in Exercise ?? to use \( O(\min(m, n)) \) space.

Actually only one row need be kept in memory at a time. When an element of the current row is computed, it should be stored in a temporary variable. After the next element is computed, the previous element may be placed in the row.

Exercise 8-2. Do exercise 16.2-2 on page 384 of CLRS.

Solution:

\( m_{i,j} \) represents the total value that can be taken from the first \( i \) items when the knapsack can hold \( j \).

Exercise 8-3. Do exercise 16.2-3 on page 384 of CLRS.

Solution:

Go greedy. Take the item with maximum value. This item also has the smallest weight and therefore, the largest \( v_i/w_i \) ratio. One wouldn’t ever take something besides the optimal choice because the optimal choice weighs at most as much and is worth as least as much as any other item.

Exercise 8-4. Do exercise 23.2-2 on page 573 of CLRS.

Solution:

Use an array (or a field) to store the distance of each node from the growing tree. Simply search the entire list to find the node with least distance. There are \( V \) iterations each taking \( \Theta(V) \) time (for the search) for a total time of \( \Theta(V^2) \).

Problem 8-1. Typesetting

In this problem you will write a program (real code that runs!) to solve the following typesetting scenario. **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. In addition, any remaining space at the end of the line is padded with blanks. Thus, if words $i$ through $j$ are printed on a line, the number of extra space characters at the end of the line (after word $j$) is

$$M - j + i - \sum_{k=i}^{j} \ell_k .$$

There are many ways to divide a paragraph into multiple lines. To produce nice-looking output, we want a division that fills each line as much as possible. A heuristic that has empirically shown itself to be effective is to charge a cost of the cube of the number of extra space characters at the end of each line. 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 $\text{linecost}(i, j)$ for printing words $i$ through $j$ on a line is given by

$$\text{linecost}(i, j) = \begin{cases} \infty & \text{if words } i \text{ through } j \text{ do not fit on a line,} \\ 0 & \text{if } j = n \text{ (i.e., last line),} \\ (M - j + i - \sum_{k=i}^{j} \ell_k)^3 & \text{otherwise.} \end{cases}$$

The total cost for typesetting a paragraph is the sum of the costs of all lines in the paragraph. An optimal solution is an division 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:**

Note that $\text{linecost}(i, j)$ is defined to be $\infty$ 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 $\text{linecost}(i, j)$ 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.

Now, 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. Also note 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 $k$th line, where $2 \leq k \leq n$. Therefore this problem displays optimal substructure.

(b) Recursively define the value of an optimal solution.

**Solution:**
Let $c(j)$ be the optimal cost of printing words 1 through $j$. From part (a), it is clear that given the optimal $i$ (i.e., the index of the first word printed on the last line of an optimal solution), we have

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

However, 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

$$c(j) = \min_{1 \leq i \leq j} \left\{ c(i - 1) + \text{linecost}(i, j) \right\}.$$

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

(c) Give an efficient algorithm to compute the cost of an optimal solution. Analyze the running time and storage space of your algorithm.

**Solution:**

We calculate the values of the array $c$ from index 1 to $n$, bottom up, which can be done efficiently since each $c(k)$ for $1 \leq k < j$ 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.

A good optimization can be obtained by noticing that computing $\text{linecost}(i, j)$ takes in general $O(j - i + 1)$ time because of the summation in the formula. However, it is possible to do this computation in $O(1)$ time with some additional pre-processing.

We create an auxiliary array $L[0\ldots n]$, where $L[i]$ is a cumulative sum of lengths of words 1 through $i$.

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

Filling in this array takes $O(n)$ time using recursion. In order to then compute $\text{linecost}(i, j)$ in $O(1)$ time, we modify the formula as follows:

$$\text{linecost}(i, j) = \begin{cases} 
\infty & \text{if words } i \text{ through } j \text{ do not fit into a line}, \\
0 & \text{if } j = n \text{ (i.e. last line)}, \\
(M - j + i - (L[j] - L[i - 1]))^{3} & \text{otherwise.}
\end{cases}$$
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 $\lceil (M + 1)/2 \rceil$ words can fit on a single line, we can reduce running time to $O(nM)$—another significant improvement—by considering only those $i$ for which $j - \lceil (M + 1)/2 \rceil + 1 \leq i \leq j$ when calculating each $c(j)$.

(d) Write code (in any programming language you wish\textsuperscript{1}) to print an optimal division of the words into lines. For simplicity, assume that a word is any sequence of characters that are not blanks. Thus, a word is a substring strictly between two space characters or bounded by the beginning or end of the input.

**Solution:**

The following code is written in C. It is a straightforward implementation of the $O(nM)$ algorithm described in part (c). The program takes as arguments the name of the input file and a value for $M$, reads the input words from the file, computes the optimal cost, and then reconstructs an optimal solution, which is printed out.

```c
/* 6.046 Spring 1998 Problem 6-1(d) */
/* 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))

define linecost(int n, int M, int i, int j); 

define dynamic_typeset(int n, int M, int p[]); 

/* 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);
lดm dynamic_typeset(int n, int M, int p[]);
```

\textsuperscript{1}The solution will be written in C.
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 auxiliary 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*extras; /* assumes result fits in a long */
}

Here is sample output. Please note that for some samples and values of $M$ there exist several different outputs with optimal cost. Below we have included each sample for each for the two values of $M$.

Sample 1, Line length = 40
COST = 2588
1:[40] Typesetting is defined as the production
2:[38] of type, and is therefore a mechanical
3:[38] and creative craft. There are numerous
4:[34] ways to set type, the early way of
5:[40] doing it being called "hot type" because
6:[36] it was done by heating molten metal.
7:[37] Originally type was set by hand, this
8:[33] involved arranging the individual
9:[40] letters on a tray, line by line. Because
10:[35] these letters could be arranged and
11:[39] moved about they were known as "movable
12:[35] type". Up until the 1880s, this was
13:[34] the only method of composition and
14:[38] the number of typefaces and fonts were
15:[39] few. Johann Gutenberg is a famous name,
16:[37] and contrary to popular belief he did
17:[37] not invent the printing press, or the
18:[36] ink or movable type. He did however,
19:[35] perfect the art of handsetting with
20:[39] movable type. It is odd why Gutenberg’s
21:[34] name is so well known, considering
22:[36] that he was not thought to be one of
23:[38] the leading contributers to the art of
24:[37] communication. There were many others
25:[39] who invented and created much more than
26:[38] Gutenberg, but it is his name which is
attached to so many of these historic events, many of which he didn’t do. In Typographic Milestones, by Allan Haley, he tells of a theory which states that Gutenberg was not the first to invent typography. This theory tells of a man named Lourens Coster who was from Holland. Supposedly one day during a walk in the woods he busied himself by cutting letters out of beech bark, and when he returned home he decided to put them next to each other to form words and sentences. For entertainment purposes he inked them and put them on paper to amuse his grandchildren. The rumour has it that he later used letters cast from metal in place of the wooden ones and actually printed entire books, until Gutenberg stole the idea and became rich off his "invention", while Coster got nothing.

Typesetting is defined as the production of type, and is therefore a mechanical and creative craft. There are numerous ways to set type, the early way of doing it being called "hot type" because it was done by heating molten metal. Originally type was set by hand, this involved arranging the individual letters on a tray, line by line. Because these letters could be arranged and moved about they were known as "movable type". Up until the 1880s, this was the only method of composition and the number of typefaces and fonts were few. Johann Gutenberg is a famous name, and contrary to popular belief he did not invent the printing press, or the ink or movable type. He did however, perfect the art of handsetting with movable type. It is odd why Gutenberg’s name is so well known, considering that he was not thought to be
the leading contributors to the art of communication. There were others who invented and created much more than Gutenberg, but it is his name which is attached to so many of these historic events, which he didn’t do. In Typographic Milestones, by Allan Haley, he tells of a theory which states that Gutenberg was not the first to invent typography. This theory tells of a man named Lourens Coster who was from Holland. Supposedly one day during a walk in the woods he busied himself by cutting letters out of beech bark, and when he returned home he decided to put them next to each other to form words and sentences. For entertainment purposes he inked them and put them on paper to amuse his grandchildren. The rumour has it that he later used letters cast from metal in place of the wooden ones and actually printed entire books, until Gutenberg stole the idea and became rich off his "invention", while Coster got nothing.

Sample 2, Line length = 40
COST = 2026
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.
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.

The first practical mechanized type casting machine was invented in 1884 by Ottmar Mergenthaler. His invention was called the "Linotype," and it produced solid lines of text cast from rows of matrices. Each matrix 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.

Throughout his life, Knuth had been intrigued by the mechanics of printing and graphics. As a boy at Wisconsin summer camp in the 1940s, he wrote a guide to plants and illustrated the flowers with a stylus on the blue ditto paper that was commonly used in printing at that time. In college, he recalls admiring the typeface used in his math textbooks. But he was content to leave
the mechanics of designing and setting type to the experts. "I never thought I would have any control over printing. Printing was done by typographers, hot lead, scary stuff. Then in 1977, I learned about new printing machines that print characters made out of zeros and ones, just bits, no lead. Suddenly, printing was a computer science problem. I couldn’t resist the challenge of developing computer tools using the new technology with which to write my next books." Knuth designed and implemented TeX, a computer language for digital typography. He explored the field of typography with characteristic thoroughness. For example, he wrote a paper called "The letter S" in which he dissects the mathematical shape of that letter through the ages, and explains his several day effort to find the equation that yields the most pleasing outline.

Sample 3, Line length = 72
COST = 1191
1: Throughout his life, Knuth had been intrigued by the mechanics of printing and graphics. As a boy at Wisconsin summer camp in the 1940s, he wrote a guide to plants and illustrated the flowers with a stylus on the blue ditto paper that was commonly used in printing at that time.
2: In college, he recalls admiring the typeface used in his math textbooks. But he was content to leave the mechanics of designing and setting type to the experts. "I never thought I would have any control over printing. Printing was done by typographers, hot lead, scary stuff. Then in 1977, I learned about new printing machines that print characters made out of zeros and ones, just bits, no lead. Suddenly, printing was a computer science problem. I couldn’t resist the challenge of developing computer
12: [66] tools using the new technology with which to write my next book.
13: [67] Knuth designed and implemented TeX, a computer language for digital
typography. He explored the field of typography with characteristic
thoroughness. For example, he wrote a paper called "The let-
ter S" in
16: [67] which he dissects the mathematical shape of that letter through
17: [67] ages, and explains his several day effort to find the equa-
tion that
18: [33] yields the most pleasing outline.

For part (d), you should turn in a printout of the code you have written and the output of your pro-
gram on the input samples which are provided at the end of the problem set. Use two values of $M$
the maximum number of characters per line), namely $M = 72$ and $M = 40$, on each input sample.
You can download these input samples from the 6.046 webpage http://theory.lcs.mit.edu/classes/
If, for some reason, you need to type the samples in yourself, please be careful about reproducing
text accurately. For example, substituting double quotes in place of two single quotes will result
in a different layout. Remember that collaboration, as usual, is allowed to solve problems, but you
must write your program by yourself.

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

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

and that the total cost is still the sum of the costs of all lines in the paragraph. Give 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. The algorithm clearly runs in linear time.
We need to 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 8-2. Scheduling to minimize average completion time

Suppose you are given a set \( S = \{a_1, a_2, \ldots, a_n\} \) of tasks, where task \( a_i \) requires \( p_i \) units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let \( c_i \) be the completion time of task \( a_i \), that is, the time at which task \( a_i \) completes processing. Your goal is to minimize the average completion time, that is, to minimize \( \frac{1}{n} \sum_{i=1}^{n} c_i \). For example, suppose there are two tasks, \( a_1 \) and \( a_2 \), with \( p_1 = 3 \) and \( p_2 = 5 \), and consider the schedule in which \( a_2 \) runs first, followed by \( a_1 \). Then \( c_2 = 5 \), \( c_1 = 8 \), and the average completion time is \( (5 + 8)/2 = 6.5 \).

(a) Suppose there are two tasks with \( p_1 = 3 \) and \( p_2 = 5 \). Consider (1) the schedule in which task 1 runs first, followed by task 2 and (2) the schedule in which task 2 runs first, followed by task 1. In each case, state the values of \( c_1 \) and \( c_2 \) and compute the average completion time.

Solution:

For (1) the schedule in which task 1 runs first, followed by task 2:
\[
\begin{align*}
c_1 &= p_1 = 3 \\
c_2 &= p_1 + p_2 = 8 \\
\text{average completion time} &= \frac{c_1 + c_2}{2} = \frac{11}{2} = 5.5
\end{align*}
\]
For (2) the schedule in which task 2 runs first, followed by task 1:
\[
\begin{align*}
c_2 &= p_2 = 5 \\
c_1 &= p_2 + p_1 = 8 \\
\text{average completion time} &= \frac{c_1 + c_2}{2} = \frac{13}{2} = 6.5
\end{align*}
\]

(b) Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task \( i \) is started, it must run continuously for \( p_i \) units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Solution:

Run tasks in shortest processing time order. This can be done by sorting the elements using heap sort or merge sort and then scheduling them in the order of increasing scheduling times. This algorithm takes \( O(n \log n) \).

This algorithm uses a greedy strategy. It is shown to be optimal as follows: \( c_{avg} = \frac{c_1 + c_2 + \cdots + c_n}{n} \). This cost can also be expressed as \( \frac{p_1 + (p_1 + p_2) + (p_1 + p_2 + p_3) + \cdots + (p_1 + p_2 + \cdots + p_n)}{n} \). \( p_1 \) is added in the most times, then \( p_2 \), etc. As a result, \( p_1 \) should have the shortest processing time, then \( p_2 \), etc. otherwise, you could cut and paste in a shorter processing time and produce a faster algorithm. As a result, the greedy property holds and our algorithm is correct.
Suppose now that the tasks are not all available at once. That is, each task has a **release time** $r_i$ before which it is not available to be processed. Suppose also that we allow **preemption**, so that a task can be suspended and restarted at a later time. For example, a task $i$ with processing time $p_i = 6$ may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task $i$ has run for a total of 6 time units, but its running time has been divided into three pieces.

(c) Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

**Solution:**

This problem also exhibits the greedy property which can be exploited by running tasks in shortest remaining processing time order. Use a priority queue which prioritizes based on the task with the shortest time remaining. Each time a new task comes up, insert it into the queue and if it would take less time to do that task then the one you are on, do the shorter task. Each time you finish a task, the next task you should do is the one with the least remaining time until completion. The priority queue can be maintained in $O(n \log n)$ time.

This algorithm minimizes the average completion time and the proof is similar to part $b$. If we do not schedule using the greedy algorithm based on remaining processing time, then we will be able to swap two time slots which would then improve the sum of the completion times and thus result in a contradiction. For example, assume you have two tasks at time $t$, where task $i$ has $x$ processing time remaining and $j$ has $y$ processing time remaining where $x > y$. Assume for the purposes of contradiction that the optimal answer has task $i$ running before task $j$. If $i$ is done before $j$ then $c_i = t + x$ and $c_j = t + x + y$. The average completion time is $\frac{2t + 2x + y}{2}$. However if $j$ were done before $i$, then $c_i = t + y + x$ and $c_j = t + y$. The average completion time is now $\frac{2t + 2y + x}{2}$ which is less the average completion time for the “optimal” solution since $x > y$. As a result, the task with the lowest time remaining should be done first.