Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer

3. December 2018

Datenstrukturen & Algorithmen   Blatt P11   HS 18

Please remember the rules of honest conduct:

- Programming exercises are to be solved alone
- Do not copy code from any source
- Do not show your code to others

Hand-in: Sunday, 16. December 2018, 23:59 clock via Online Judge (source code only).
Questions concerning the assignment will be discussed as usual in the forum.

Those who cannot remember the past are condemned to repeat it.
— Dynamic Programming

Exercise P11.1 *Parenthesis.*

Input

You are given an arithmetic expression containing n numbers and $n-1$ operators, each either $+$ or $-$. Your goal is to perform the operations in an order that maximizes the value of the expression. That is, insert parentheses into the expression so that its value is maximized.

For example, for the expression $6-3-2+5$, the optimal pacing of parhenthisis is: $6-(3-2)+5=10$.

Input The first line of the input is an integer N, $1 \leq N \leq 10^5$, denoting the number of integers in the expression. The next line contains an expression of the form $v_1\ op_1\ v_2\ op_2\ v_3\ op_3 \ldots op_{N-1}\ v_N$ where $op_i \in \{+,-\}$ for $i \in [1 \ldots N-1]$ and $1 \leq v_j \leq 10^9$ for $j \in [1 \ldots N]$

Output The output contains a single line that represents the maximum value of the expression that can be obtained by placing zero or more pairs of parenthesis.

Grading You get 3 bonus points if your program works for all inputs. Your algorithm must complete the solution in $O(N)$ time complexity, and $O(N)$ space complexity. Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=25012&problem=AD8H11P1`. The enrollment password is "`asymptotic`".

Example

Input:

```
4
6 - 3 - 2 + 5
```

Output:

```
10
```

Notes

For this exercise we provide an archive containing a program template available at `https://www.cadmo.ethz.ch/education/lectures/HS18/DA/uebungen/AD8H11P1.Parenthesis.zip` The archive also contains additional test cases (which differ from the ones used for grading). Importing any additional Java class is **not allowed** (with the exception of the already imported ones `java.io.{InputStream, OutputStream}` and `java.util.Scanner` class.

Solution

The problem can be solved using the main idea of the matrix chain multiplication algorithm.

To solve the subexpression $v_i \ldots v_j$, we can split it into two problems at the k-th operator, and recursively solve the subexpressions $v_i \ldots v_k$ and $v_{k+1} \ldots v_j$. In doing so, we must consider all combinations of the minimizing and maximizing subproblems.

Let $M[i, j]$ be the maximum value obtainable from the subexpression beginning at v_i and ending at v_j (i.e., v_i $op_i \ldots op_{j-1}$ v_j), and let $m[i, j]$ be the minimum value obtainable from the subexpression beginning at v_i and ending at v_j. The base cases are $M[i, i] = m[i, i] = v_i$, for all $i \in [1 \ldots N]$.

The recursive function is then defined as:

$$M[a,b] = \max_{a \le k < b}(\max(M[a,k]\ op_k\ M[k+1,b], M[a,k]\ op_k\ m[k+1,b], m[a,k]\ op_k\ M[k+1,b], m[a,k]\ op_k\ m[k+1,b])) \tag{1}$$

$$m[a,b] = \min_{a \le k < b}(\min(M[a,k]\ op_k\ M[k+1,b], M[a,k]\ op_k\ m[k+1,b], m[a,k]\ op_k\ M[k+1,b], m[a,k]\ op_k\ m[k+1,b])) \tag{2}$$

Using dynamic programming, this strategy will result in a solution with complexity of $O(n^3)$, as there are $O(n^2)$ subproblems, and at each level we have to consider $b - a$ of them i.e. $O(n)$.

However, we can do better, optimizing the solution even further. Instead of m and M lets consider min_k which will represent $m[k, n]$ and max_k that will represent $M[k, n]$ for some $k \in [1 \ldots n]$ where n is the number of the sequence. We can observe the following:

1. If all signs are "+", then we can calculate min_k and max_k by iterating k from n down to 1:

   $$min_k = min_{k+1} + v_k \text{ and } max_k = max_{k+1} + v_k$$

2. If there is only one "−", such that the sign is before v_k, then min_k can be calculated by placing the parenthesis after the "−" sign and at the end of the expression. This essentially means that we are negating all positive numbers after the negative sign.

   $$pos_{k+1} = \sum_{i=k+1}^{n} v_i \text{ and } min_k = -pos_{k+1} - v_k$$

   The calculation for max_k remains the same.

3. If there is more than one "$-$", placed before $\{v_{k_1}, v_{k_2}, \dots v_{k_m}\}$ ($k_i < k_j$ for all $i < j$) then min_{k_i} can be calculated by negating all positive numbers between k_i and k_{i+1} i.e. placing the parenthesis before v_{k_i} and after v_{k_i-1}.

   On the other hand, for calculating max_k once "$-$" is placed before v_k, we can do the following:

$$max_k = \max(v_k + max_{k+1}, v_k - min_{k+1})$$

This means that in order to design the algorithm, we can place all numbers in an array, such that we negate the ones that are right after a "$-$" operator. According to the observation above, we would need a variable to keep track of the sum of the positive numbers, as well as two other variable to keep the min and max. The solution is given below:

```
1  long max = 0;
2  long min = 0;
3  long pos = 0;
4
5  for (int i = N - 1; i >= 0; i -= 1) {
6      if (v[i] > 0) {
7          max += v[i];
8          min += v[i];
9          pos += v[i];  // keeps track of positive numbers
10     } else {
11         max = Math.max(v[i] - min, v[i] + max);
12         //
13         // the positive numbers must be negated. However we have already added them
14         // into min (line 8), thus in order to negate them, we must remove them twice
15         //
16         min += v[i] - 2 * pos;
17         pos = 0;
18     }
19 }
20
21 out.println(max);
```

The algorithm above, traverses the values in inverse order, such that at each position i is calculating min_i and max_i. As the first number in the array will always be positive, the `max` will hold the solution to the problem, after the loop terminates.

Exercise P11.2 *Line Breaks.*

One of the basic problems of typesetting is breaking the words into lines and then breaking the lines into pages, with the resulting layout as beautiful and readable as possible. Your task is a (greatly simplified) version of the first part, that is to decide where to place the line breaks in a given text.

The input text T is a sequence of paragraphs $P_1, \dots, P_k$ that are processed independently, and each paragraph P_i is a list of n_i words and has given page width $w_i > 0$. Every word in paragraph P_i has length at most w_i.

The output is the same text with one or more consecutive words on one line, every two words on a line are separated by exactly one space. All the characters (including spaces) have the same width and so the length $len(L)$ of line L is the number of word-characters and spaces in between them, for example "`This is an example.`" has length $4 + 1 + 2 + 1 + 2 + 1 + 8 = 19$.

The goal is to format the text such that every line of paragraph P_i has length at most w_i, but also as nicely as possible: Informally, we want all the lines to have length as close to w_i as possible. And formally, we assign every line L penalty $(w_i - len(L))^2$, that is the square of the number of spaces you would need to add to make the line exactly w_i characters long. For example a line with length exactly w has penalty 0 and a line with just one single-character word would have penalty $(w_i - 1)^2$. The last line of the resulting paragraph is an exception – the penalty of this line is always 0, as the length of the last line does not matter for typesetting.

For every paragraph, the goal is to find the optimal line breaks that minimize the sum of the penalties of the resulting lines. Note that a *"greedy"* solution that would make every line as long as possible before starting a new line is generally not optimal – see the example below.

Input The input consists of several paragraphs. The first line of the file contains the integer $k > 0$, the number of paragraphs to follow.

Each paragraph P_i is independent of the others and consists of several lines: The first line contains the integers $n_i > 0$, the number of words of the paragraph, and $w_i > 0$, the width of the page, separated by a space. The following n_i lines contain the words of the paragraph, one word each.

The words may consist of English letters, numbers and the following characters[1]: `-.,?!:;'"()[]{}`

Output For every paragraph, the output should contain a single line with the smallest possible penalty.

Grading This exercise awards no points. The program should be reasonably efficient and work in $O(w_1 n_1 + \cdots + w_k n_k)$ time complexity. Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=25012&problem=AD8H11P2`. The enrollment password is "`asymptotic`".

Examples

Input

```
2
9 9
A
B
C
D
E
F
GGGGGGGGG
HHH
I.
8 18
Lorem
ipsum
dolor
sit
amet,
consectetur
adipiscing
elit.
```

Output

[1]But the exact set should not matter to your program anyway.

```
32
107
```

Example optimal formatting with marks for page width and line penalties.

```
A B C    | 16
D E F    | 16
GGGGGGGGG| 0
HHH I.   | 0 (last line)

Lorem ipsum       | 49
dolor sit amet,   | 9
consectetur       | 49
adipiscing elit.  | 0 (last line)
```

Notes For this exercise we provide an archive containing a program template available at `https://www.cadmo.ethz.ch/education/lectures/HS18/DA/uebungen/AD8H11P2.LineBreaks.zip` The archive also contains additional test cases (which differ from the ones used for grading). Importing any additional Java class is **not allowed** (with the exception of the already imported ones `java.io.`{`InputStream`, `OutputStream`} and `java.util.Scanner` class.

Solution Again, we will use dynamic programming for this task: for every $i \in \{0, \ldots, n\}$, let m_i be the total penalty if the last line break was just before word w_i (not considering the special case for last word of a paragraph here). We may set $m_0 = 0$, as a break before the word w_0 is the start of the text. Note that the actual characters of the text did not matter, only the word lengths.

Now to compute m_i if we know all previous values of $m_{j<i}$, consider how can the solution with the last break before w_i look: For some $x > 0$, it will consist of the best solution with the last line break before w_{i-x}, a line break and then x words $w_{i-x}, w_{i-x+1}, \ldots, w_{i-1}$ on the last line. The penalty of the best solution with the last line break before w_{i-x} is already known in m_{i-x}, and it is straightforward to compute the penalty for line with words $w_{i-x}, w_{i-x+1}, \ldots, w_{i-1}$. If we try all the possible values of $0 < x \leq i$ such that the last line has length at most w, we take m_i as the minimal total penalty over all such x. We might need to check $O(w)$ values of x since $x \leq (w+1)/2$ (there can be at most $(w+1)/2$ space-separated words on a line).

Now for the last line of paragraph, we only need to change the way m_n is computed: We still consider all feasible values of x (with the last line fitting into w characters) and take minimum of m_{n-x}, but do not add the last line penalty. This can be accomplished with a simple condition.

This gives us solution with running time $O(nw^2)$, since we try $O(w)$ values of x for every of n words, and in every case it can take us $O(w)$ time to calculate the penalty of a given range of words $w_{i-x}, w_{i-x+1}, \ldots, w_{i-1}$ on a single line.

However, this can be made faster by for example pre-computing the sum of word lengths as $s_i = \sum_{j=0}^{i} |w_j|$, and then the length of line with words $w_a, \ldots, w_b$ is $s_b - s_a + (b - a - 1)$ (last part for the spaces). Another solution is to start with $x = 1$ and increment it, and in every step remember the length of the considered last line so far, incrementing it with every added word – the penalty is then computed in $O(1)$ time.

With either solution, we get $O(nw)$ time solution.