プログラミング演習 第 5 回

問 A [05A]：整数の合計

n 以上 m 以下の整数の和は、ループを用いて簡単に求めることができる。しかし、あえて次の漸化式に沿った**再帰的関数**sum を定義し、それを用いて、標準入力 (端末等) から n と m を受けとって n 以上 m 以下の整数の和を印字するプログラムを作れ。

$$sum(n,m) = \begin{cases} 0 & n > m \\ n + sum(n+1,m) & \text{otherwise} \end{cases}$$

$n > m$ であれば、n 以上かつ m 以下の整数は存在しないので、和は当然 0 である。
補足: 再帰呼出しができる回数は限られている。そのため、$m - n$ があまりに大きいと正しく実行できず、`Segmentation fault` と表示される場合がある。また、sum の戻り値を `int` とすると、桁溢れを起こして正しくない値になる場合もある。この問題では、そのような状態にならない入力に対して正しい答を印字できればよい。また、何回まで再帰呼び出しができるか確かめてみることも有益である。

入出力例

```
入力
  20 30
出力
  275
```

```
入力
  100 20
出力
  0
```

問 B [05B]：二種類の括弧

この問題はスタックの復習です。
標準入力 (端末等) から文字列 (78 文字以下) を受けとって、() と {} の二種類の括弧について全て対応して使われているかどうかを判定するプログラムを、**再帰呼出しを用いずに**作れ。対応している場合は `yes`、そうでない場合は `no` と印字すること。
対応して使われていない文字列 (`no` と印字すべきもの) には次のような場合がある。

A. 開き括弧が現れる前に閉じ括弧が現れている: `1+2)*3` や `(1+2)*3)`

B. 期待した種類と異なる閉じ括弧が現れている: `(1+2}*3` や {`1+2)*3`}

C. 閉じ括弧が足りない: `(1+2` や `(1+2)+(3`

入出力例

入力	出力	入力	出力
`(1+2)*{3+4}`	`yes`	`(1+2)*3+4}`	`no`
`sin((1+2)*{3+4})`	`yes`	`sin((1+2)*{3+4}}`	`no`
`x/y`	`yes`	`(({(a+b)}`	`no`

ヒント

- 括弧 (`(){}`) と文字列の終わり (`'\0'`) 以外の文字は無視すればよい。

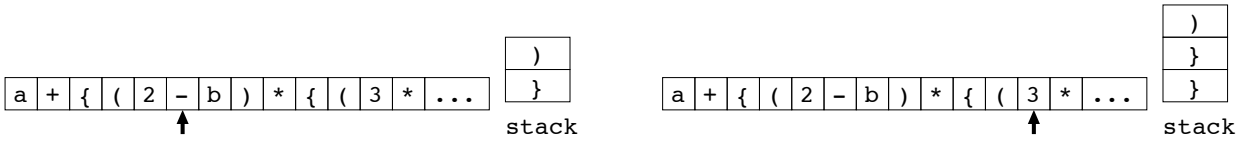

図 1: 問 B で 6 文字目と 12 文字目まで調べた時のスタックの内容

図 2: 問 C の *check* の動作 (再帰呼出し直前 (左) と再帰呼出しから戻る直前 (右))

- スタックを用いるとよい。文字列を先頭から 1 文字ずつ調べて、文字によって適切なスタック操作をする。それによって、常にスタックトップには次に現れることを期待している閉じ括弧が (もしあれば) 積まれているようにすればよい。

  この方針でプログラムを書けば、例えば、`a+{(2-b)*{(3*...` という文字列について、6 文字目と 12 文字目まで調べた時のスタックの内容は、それぞれ、図 1 のようになる。

  また、括弧が対応して使われていない場合 A〜C のうち A の場合は、`)` か `}` を見つけた時にスタックが空である状況に対応する。B の場合と C の場合も考え、それらの状況になると `no` と印字してプログラムを終了すればよい。

  なお、スタックの大きさは、入力文字列長の上限を参考に適切に決めること。

問 C [05C] : 二種類の括弧 (再帰関数版)

標準入力 (端末等) から文字列 (78 文字以下) を受けとって、() と {} の二種類の括弧について全て対応して使われているかどうかを、**再帰的関数を用いて**判定するするプログラムを作れ。対応している場合は `yes`、そうでない場合は `no` と印字すること。

入出力例

問 B と同じ。

ヒント

- 文字列 s を受けとって、それが

  1. 全ての括弧が対応して使われている
  2. 問 B で括弧が対応して使われていない場合の A
  3. 問 B で括弧が対応して使われていない場合の B または C

  のいずれに該当するかを分類し、さらに 2. であれば、対応しない閉じ括弧の位置も返す関数 *check* を定義することを考える。*check* は s を先頭から 1 文字ずつ調べて、開き括弧を見つけた時は、それに対応する閉じ括弧を探すために *check* を再帰的に用いる。

  例えば図 2 左側のように *check* に s として`"...{a+{(2-b)}}..."`が与えられ、矢印の位置で { を発見したとする。この時、矢印の次の位置から始まる文字列を引数にして *check* を再帰的に呼び出すと、図 2 右側の矢印の位置が返される。ここに } があるはずなので、本当に } があるか確認し、もしあれば続きの文字列を調べる。もしなければ、直ちに 3. に分類する。

  与えられた文字列 s が`"...{a+{(2-b)}*1` (ここで文字列の終わり)`"`や`"...{a+{(2-b))}..."`だった時は、`"a+{(2-b)}*1"`や`"a+{(2-b))}..."`を引数に *check* を再帰呼出しすることになる。再帰的に呼び出された *check* によって、これらの文字列は 1.〜3. のどれに分類されるか、またその時、文字列全体 (`"...{a+{(2-b)}*1"`や`"...{a+{(2-b))}..."`) は 1.〜3. のどれに分類すればよいかを考えて、*check* を完成させればよい。

```
char* check(int* c, char* s) {
  ...
  *c = 分類;
  return 文字列中の位置;  <--- 分類が 2. 以外の時は何を返してもよい (NULL が適切)
}
```

図 3: 分類と文字列中の位置を同時に返す関数

- 分類と文字列中の位置を同時に返すためには、例えば、分類を代入する変数へのポインタを受けとるようにすればよい。そのような *check* の定義は例えば図 3 のようになる。

問 D [05D] : 支払い

Y 円の料金を 1,000 円札、500 円硬貨、100 円硬貨、50 円硬貨、10 円硬貨を使って支払う場合の組み合わせの数を求めるプログラムを作成せよ。ここで、標準入力から支払金額を与えるものとし、支払いに利用可能な札および硬貨はいくらでもあるものとする。また Y は 10 以上 100,000 以下の 10 の倍数としてよい。

ヒント

いま、まず 1,000 円札を何枚か用いて支払いを行うことを考えると、Y 円の料金を支払う場合の組み合わせ数は、500 円以下の硬貨によって Y-1000 円の料金を 支払う場合の組み合わせ数、Y-2000 円の料金を支払う場合の組み合わせ数、...、$Y - 1000n$ 円の料金を支払う場合の組み合わせ数の合計である (ただし n は $Y - 1000n \geqq 0$ を満たす 0 以上の整数)。500 円以下の硬貨によって支払う組み合わせ数も、同様に 1,2$\cdots m$ 枚の 500 円硬貨を使う場合それぞれの組み合わせ数の合計で求めることができる。