Programmazione in userRPL

Indice

1. Scopo del documento

Si trovano diversi manuali di programmazione in userRPL, sia in inglese che in spagnolo, senza contare i vari suggerimenti disponibili in luoghi di discussione come i gruppi su comp.sys.hp48. Ci sono molti siti [1] e risorse utili, bisogna dedicarci una ricerca, ma un buon elenco da cui partire è quello offerto da hpcalc.org qui. Tuttavia si trova poco in italiano [2] mentre in inglese, forse perchè ho letto poco, l'esposizione a volte non mi ha soddisfatto[3] o è correlata ad un vasto argomento[4]. Dunque ho deciso di creare questo documento per iniziare un qualsiasi utilizzatore dell'hp50g, o altre macchine che permettono di programmare in userRPL, all'uso del linguaggio di programmazione userRPL. Con "uso del linguaggio di programmazione userRPL" intendo l'esposizione delle strutture più comunemente usate per creare la maggior parte dei programmi utili.

Avverto che l'userRPL è adatto a risolvere problemi *non* intensivi dal lato della computazione, salvo non si voglia aspettare molto, almeno ciò accade sull'hardware di riferimento. Per problemi computazionalmente intensivi sono disponibili altri tools di programmazione che sfruttano meglio le capacità di elaborazione dell'hardware di riferimento, ad esempio hpgcc3.

1.1. Note

Spesso riguardo a certe strutture del codice sarebbe possibile dire molto ma, per motivi di tempo, si cerca di dire il necessario rispetto al problema che si vuole risolvere in quel momento, sperando di coprire il resto con modifiche future.

---

[1] Uno su tutti: hpcalc.org .

[2] Anche se ci sono diverse discussioni su vari forum (uno consigliato è helpcalculator) che qualche indicazione possono darla, ma sono sparse, purtroppo.

[3] In effetti, per motivi di brevità, sembra che anche l'esposizione in questo documento non sia completa come previsto. Vabbè male che vada è l'ennesimo documento free sull'argomento, la ridondanza non fa mai male.

[4] Si vedano ad esempio le "maratone" relative a vari argomenti di matematica qui.

Il documento verrà espanso e modificato di volta in volta, quindi non aspettatevi di trovarlo con tutti i capitoli e contenuti sin da subito! E' come se fosse un cantiere continuo!

1.2. Materiale didattico minimo aggiuntivo

- Hp50g user's guide (da ora UG) capitolo 21;
- Hp50g advanced user's guide (se potete la versione bookmarked che qui hanno tolto :-( ). Consigliatissimo, la sua abbreviazione sarà AUR in questo documento;

1.3. Hardware di riferimento

La (mitica) hp50g.

1.4. Versioni del documento

Elenco delle modifiche al documento, in ordine di data.

- 21 Nov 2012
- 30 Dic 2012
- 14 Feb 2013
- 15 Feb 2013
- 17 Feb 2013
- 18 Feb 2013
- 23 Feb 2013
- 24 Feb 2013

1.5. Utilizzo del documento

Cercherò sempre di pubblicare il sorgente editabile (in .docx) del documento. Questo è liberamente modificabile ed utilizzabile, a patto di rendere tributo a tutti coloro che l'hanno modificato in precedenza con il loro impegno (ovvero non dovete rivendicarne la paternità originale).

1.6. Contributi

Elenco di coloro che hanno contribuito a modificare il documento (nomi reali o nickname è uguale).

- Pierfrancesco A.
- Fabio F. (contributi su hpcalc.org)

1.7. Contatti

Per suggerimenti di qualsiasi genere [5] : temporarymailname-userrpl@yahoo.it (no non è temporaneo, è che ho scelto un nome ingannevole).

[5] Specie richiesta di spiegazioni ulteriori che aiutano a capire dove non è chiaro il documento.

2. Introduzione e definizione di semplici funzioni

Una delle esigenze più comuni durante l'esecuzione del calcoli è la seguente: si ha una certa funzione e si vuole il suo valore, tuttavia i parametri della funzione variano e si vorrebbe calcolare il relativo valore della funzione senza dover ogni volta riscriverla da capo.

Esempio: data $f(x,y,z) = 2x + 3y + 4z$ si vuole il valore della funzione relativo alle triple $\{(x = 1, y = 1, z = 2), (1,3,2), (2,5,4)\}$.

Vedremo in questo capitolo come fare.

2.1. Prerequisiti

- Avere un interprete del linguaggio userRPL, può essere una calcolatrice fisica come un emulatore. Deve essere impostato in modalità RPN;
- Capacità di navigare e familiarità con i menu base della calcolatrice (per gli esempi);
- Capacità di utilizzare la calcolatrice come una qualsiasi calcolatrice scientifica (non grafica nè programmabile), ad esempio saper calcolare $5^5 + \sqrt{5} + \log(\cos(236°))$;
- Capacità di installare programmi e prendere confidenza con le loro funzionalità di base;
- Capacità di riuscire a fare ricerche non troppo complesse su internet;
- Pazienza e volontà di applicarsi;
- Un minimo di conoscenza dell'inglese, giusto per le traduzioni di qualche vocabolo;
- Opzionali:
  - Installare e prendere confidenza con debug4x;
  - saper salvare un documento in formato userRPL con debug4x (facile);
  - saper trasferire file dal computer alla calcolatrice (facile);

2.2. Flags

Sull'hp50g le flags impostate col tick (la spunta) che possono cambiare l'esecuzione sono sono la 27, la 105, la 119 e la 127.

2.3. Funzioni di un parametro e blocchi di codice

In questa sezione vedremo come scrivere funzioni di un parametro, del tipo $f(x)$, di cui vogliamo sapere il risultato dati in input diversi valori di x.

Per prima cosa dobbiamo creare un ***blocco di codice*** entro il quale viene definita la funzione, ed in generale ***un programma userRPL***. Un blocco di codice viene definito dall'apertura e dalla chiusura dei *caporali* («...»).

In debug4x possiamo scrivere (creando un nuovo file userRPL), su due linee separate o sulla stessa linea:

```
\<<
\>>
```

Ottenendo:

```
«
»
```

2.3.1. Definire il parametro in input ; variabile locale ; stack

In questo blocco di codice possiamo fare tante cose, ma a noi interessa definire una funzione di un parametro. La funzione di un parametro, per il nome stesso, richiede il valore del parametro in input per produrre un risultato numerico. Ovvero $f(x) = 3x$ produce il risultato $3x$ (un risultato simbolico) se si lascia il parametro simbolico ($x = x$), produce invece 15 (un risultato numerico) se in input riceve un risultato numerico ($x = 5$). Vediamo dunque come fare per ***memorizzare valori numerici in input***.

All'inizio del blocco di codice, subito dopo il primo caporale, aggiungiamo la scrittura (in debug4x):

```
\-> x
```

Che fa ottenere:

```
« → x
»
```

Come codice complessivo. Sull'hp50g avremo:

Cosa fa questa scrittura? In realtà ha una semantica vasta ma, come scritto nel capitolo 1, ci limitiamo alle esigenze rispetto al problema da risolvere. Dunque, dicevamo, la semantica di questa aggiunta è la seguente: "*Prendi il valore nel livello dello stack precedente al livello dello stack del blocco di codice corrente e salvalo nella* ***variabile locale*** *x*".

Che significano i livelli dello stack? Come potete vedere nell'immagine, la calcolatrice mostra i valori nei vari livelli di memoria, lo ***stack*** infatti è suddiviso in livelli (che possono contenere dati più o meno complessi). In particolare il programma o "funzione" sarà presente, prima dell'esecuzione, al livello 1, mentre il valore in input deve essere memorizzato al livello 2. Al momento dell'esecuzione il programma leggerà il livello 2 (che è precedente al livello 1) per salvare il contenuto di questo nella variabile 'x'. Per maggiori informazioni sullo stack vedere articoli online, come quello su wikipedia.

Cosa è una variabile locale? E' una variabile che esiste solo entro un blocco di codice, con il suo contenuto, dopodichè viene cancellata.

Quindi in questo modo riusciamo a memorizzare l'input necessario per eseguire la funzione. Non ci resta che definire la funzione che utilizzi questo input.

2.3.2. Definire la funzione e migliorare la leggibilità dei programmi; indentazione; commenti

Per definire la funzione che utilizzi il parametro in input, subito dopo aver memorizzato il parametro, bisogna aprire un nuovo blocco di codice. In questo definiremo la funzione. Dunque scriviamo, possibilmente in maniera ordinata su righe diverse da quelle di memorizzazione del parametro:

```
« → x
   «
   »
»
```

Bisogna per forza creare un blocco di codice interno per definire la funzione? Sì[6] se si definisce questa come sequenza di istruzioni in formato RPN (si vedrà meglio in seguito cosa vuol dire).

Osservate la differenza di spazio tra il blocco di codice più interno e quello esterno[7] , ovvero una differenza di ***indentazione*** che serve per facilitare la lettura del programma. Un altro strumento per facilitare la lettura dei programmi, sempre su debug4x o comunque su editor al computer[8], sono i ***commenti***[9]. Per definire un commento si usa il simbolo "@" e tutto ciò che viene dopo questo simbolo, sulla stessa riga di testo, è considerato commento e non istruzione. Ad esempio:

```
@apriamo il blocco di codice che
@definirà l'intera funzione
«
→ x @ memorizziamo il parametro
     @ in input
     @ nella variabile locale x

  @apriamo un blocco di codice
  @interno per definire
```

---

[6] O meglio, per le fonti lette finora sì.

[7] Questa si vede in debug4x, sulla hp50g lo spazio per la visualizzazione è ristretto e quindi evitiamo tutti questi dettagli per facilitare la lettura del codice.

[8] Potreste chiedervi "perchè usare la calcolatrice se devo usare il computer?". La calcolatrice è uno strumento di calcolo (dove tuttavia si possono definire facilmente piccole funzioni o programmi), il computer uno strumento di calcolo e progettazione. Quindi usiamo il computer per progettare e definire programmi che useremo spesso durante i calcoli usando la sola calcolatrice.

[9] I quali vengono soppressi, per risparmiare spazio, sulla calcolatrice.

```
    @ la  struttura  della  funzione
    «
    »
»
```

Sulla calcolatrice invece avremo:

Siccome è brutto premiamo enter ed avremo:

Quindi con le freccie ci spostiamo sul primo livello dello stack e premiamo edit:

Ottenendo una visualizzazione più leggibile ed editabile:

2.3.2.1. Visibilità delle variabili

Si è scritto in precedenza che una variabile locale esiste solo nel suo blocco di codice, allora come farà ad essere usata nel blocco interno di codice che abbiamo appena usato? In realtà l'usabilità (o ***visibilità) di una variabile***, almeno per ciò che abbiamo scritto, si ha nel blocco di codice in cui viene definita per la prima volta ed in tutti i blocchi di codice interni a questo. Siccome abbiamo definito "x" nel blocco di codice più grande, allora "x" sarà utilizzabile anche nel blocco di codice interno a quello più grande. Vedere pure AUR 1-7 e seguenti.

2.3.2.2. Torniamo alla definizione della funzione ; elementi di un programma come istruzioni sullo stack

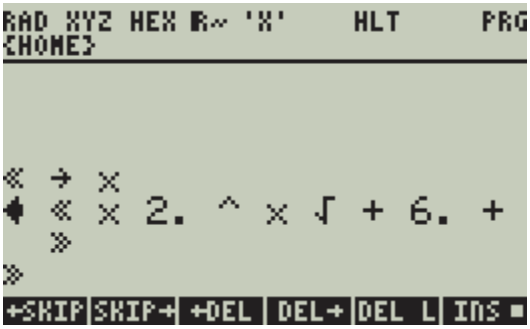
Detto ciò iniziare a definire la funzione aggiungendo istruzioni come se dovessimo digitarle manualmente nello stack (ovvero il normale uso della calcolatrice). Ad esempio vogliamo che la funzione sia $f(x) = x^2 + \sqrt{x} + 6$ ed allora scriveremo, poichè siamo in modalità RPN:

```
@apriamo il blocco di codice che
@definirà l'intera funzione
«
→  x  @ memorizziamo il parametro
      @ in input
      @ nella variabile locale x

   @apriamo un blocco di codice
   @interno per definire
   @la struttura della funzione
   «  x  2  ^  @calcolo x^2
      x  √  @calcolo √x
      +  @sommo i due calcoli
         @precedenti
      6  +  @aggiungo
            @a ciò che abbiamo
            @calcolato prima
            @ 6
   »
»
```

E' importante osservare che ***i singoli elementi di un programma sono processati come una sequenza di istruzioni sullo stack***. E' come se le digitassimo manualmente, ma in realtà sono già tutte pronte e la calcolatrice non fa altro che eseguirle in sequenza. Quindi come avremmo risolto manualmente, con operazioni dirette sullo stack, il calcolo della funzione per un valore di x, così lo risolviamo per qualsiasi valore numerico memorizzato nel parametro x.

Sulla calcolatrice (o digitando subito sulla calcolatrice o trasferendo il programma dal computer alla calcolatrice) avremo:



Digitiamo enter finchè non arriviamo alla seguente schermata[10]:

Dopodichè digitiamo 'F1' sullo stack:

Premiamo store ed abbiamo salvato la nostra funzione (o mini programma) in F1. Ovvero avremo il seguente stack:

Ma andando a vedere i file nella HOME avremo:

---

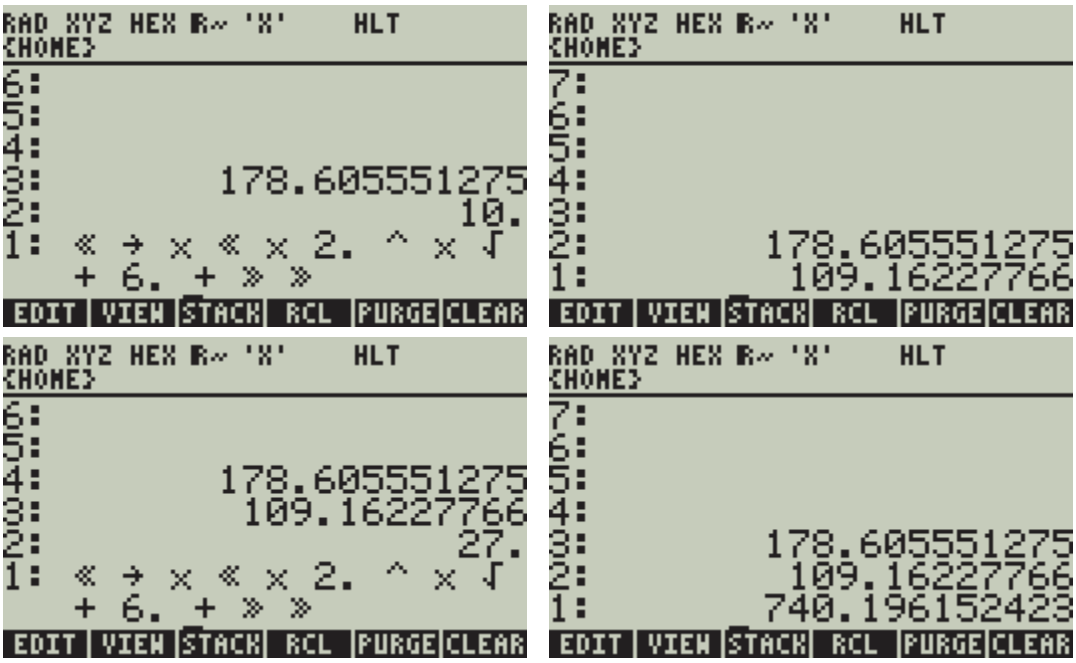
[10] manca '+' finale nella seguente immagine perchè non ho fatto le immagini nello stesso momento, ma si capisce la sequenza di operazioni da fare.

Il nostro bel programmino pronto. Quindi usiamolo! Torniamo alla visualizzazione dello stack e mettiamo come parametro di input il valore 13, ovvero $x = 13$.

Poi richiamiamo la funzione, ci sono molti modi di farlo ma per brevità ne esporremo uno: andate nella home, selezionate il programma F1 e premete RCL (recall). Tornando alla schermata dello stack avrete:

Ora premete EVAL:

Ecco il risultato della vostra funzione. Possiamo richiamarla con qualsiasi altro valore come abbiamo prima visto:



2.4. Funzioni di più parametri

Rifacendoci alla sezione relativa alle funzioni di un parametro possiamo estendere il concetto a funzioni di più parametri.

Per definire una funzione di più parametri è necessario prima memorizzare più parametri, vediamo come fare.

2.4.1. Memorizzare più parametri

Il concetto è l'estensione di quello visto anteriormente. Invece di leggere solo il livello dello stack precedente al programma leggiamo tanti livelli dello stack precedenti per quanti sono i parametri in input del programma. Per chiarire, vediamo il codice:

```
«
→  @diciamo che stiamo
   @memorizzando
x 1  @memorizza il contenuto
     @dello stack 4 livelli
     @prima del programma
x 2  @3 liv prima
x 3  @2 liv prima
x 4  @1 liv prima
   «
   »
»
```

Quindi stiamo definendo una funzione con 4 parametri, per la precisione $f(x_1, x_2, x_3, x_4) = x_1 - 2x_2 + 3x_3 - 4x_4$. Come prima non ci resta che definire facilmente la funzione nel blocco interno di codice.

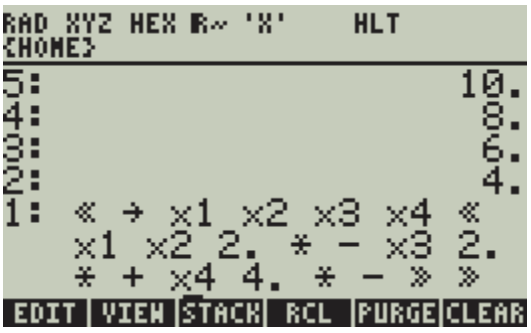
```
«
```

```
→  @diciamo che stiamo
   @memorizzando
x 1  @memorizza il contenuto
     @dello stack 4 livelli
     @prima del programma
x 2  @3 liv prima
x 3  @2 liv prima
x 4  @1 liv prima
   «  x 1  @mettiamo nello
         @stack il valore
         @memorizzato da x1
      x 2  2  *  @2*x2
      -  @x1-2*x2
      x 3  3  *  @3*x3
      +  @x1-2*x2+3*x3
      x 4  4  *
      -  @x1-2*x2+3*x3-4*x4
   »
»
```

Se siamo in debug4x selezioniamo il "send RPL to emulator" ottenendo sull'emulatore (altrimenti dobbiamo inviare il file di testo sulla calcolatrice con gli accorgimenti opportuni, vedere prerequisiti) la seguente schermata:

Salviamo il programma col nome 'F2'

Ed eseguiamo $f(10,8,6,4)$ ripetendo i passaggi visti nella sezione precedente.


RAD XYZ HEX R↔ 'X' HLT
{HOME}
5: 10.
4: 8.
3: 6.
2: 4.
1: « → x1 x2 x3 x4 «
x1 x2 2. * - x3 2.
* + x4 4. * - » »
EDIT VIEW STACK RCL PURGE CLEAR


RAD XYZ HEX R↔ 'X' HLT
{HOME}
7:
6:
5:
4:
3:
2:
1: -4.
EDIT VIEW STACK RCL PURGE CLEAR

3. Progetto Eulero

Come ho scritto in precedenza questo documento vuole tentare di dare una spiegazione dei concetti basilari per la programmazione in UserRpl, in particolare si vorrebbero esporre quelli via via più complessi facendo uso, nel caso di rimandi, di concetti precedentemente esposti senza usare quelli ancora da esporre [11] . Inoltre, dei concetti esposti, si vorrebbe dare la versione più esaustiva possibile, senza omettere alcuna possibile potenzialità (o limitazione) [12] . Un compito che, per il sottoscritto, è abbastanza arduo (se non impossibile) e, seppur io abbia notificato i miei limiti nella sezione iniziale, questo pensiero limita la mia l'ispirazione a procedere col documento. Dunque ho deciso di aggiungere una sezione che basa la discussione solo su esempi, quindi senza pretese di esaustività o grandi spiegazioni. Io non apprezzo particolarmente i testi che espongono la teoria tramite esempi, poichè lasciano il compito di indurre buone definizioni al lettore, ma se è il massimo del contributo che posso produrre è sempre meglio di niente.

Questi esempi non saranno i primi che mi vengono in mente, ma saranno la risoluzione dei problemi del progetto eulero. I problemi non verranno risolti nella maniera ottima, l'obiettivo è soltanto "risolverli" esponendo man mano le tecniche usate.

Altra nota sui problemi del progetto. Questi possono essere risolti in maniera molto elegante a livello matematico, individuando algoritmi molto efficienti, tuttavia qui si vedrà solo un approccio informatico abbastanza grezzo.

3.1. Note

Il codice menzionato in questo capitolo si può trovare su assembla.

3.2. Prerequisiti

[11] Come molti testi fanno, personalmente lo trovo odioso.

[12] Le omissioni, almeno per i testi che ho letto, ci sono sempre state, purtroppo.

- Aver compreso il capitolo 2 (e dunque: aver soddisfatto tutti i prerequisiti del capitolo menzionato).

3.3. Flags

Sull'hp50g le flags impostate col tick (la spunta) che possono cambiare l'esecuzione sono sono la 27, la 105, la 119 e la 127.

3.4. Problema 1 ; come affrontare la scrittura di un algoritmo ; pseudocodice ; modulo

***Problema 1**: Sommare tutti i numeri naturali (dunque* $\mathbb{N}$ *, ricordando che* $\mathbb{N}^* = \mathbb{N} \cup \{0\}$ *) inferiori a mille tali che siano multipli di tre o di cinque.*

Esempio del problema: dati i numeri $\{1,\ldots,14\}$ la soluzione è $3+5+6+9+10+12=45$.

Per prima cosa, per trovare **un** algoritmo risolutivo[13], risolviamo il problema in un linguaggio simile a quello naturale che tuttavia scompone il pensiero in passi, avvicinandosi alla descrizione di un algoritmo, detto ***pseudocodice***.

Pseudocodice 1:

- $\forall\, i \in \{1,\ldots,999\}$
  - se $i \bmod 3 = 0$ o $i \bmod 5 = 0$
    - aggiungi i alla somma

Dove $x \bmod y$ indica la funzione che ritorna il resto della divisione (con quoziente intero) tra x ed y. Se il resto è zero vuol dire che x è multiplo di y.

Dato lo pseudocodice, possiamo facilmente tradurre questo in codice userRPL.

3.4.1. Strutture iterative ; test su condizioni ; FOR indice NEXT ; IF THEN ELSE ; operatori

[13] Niente vieta che un problema possa essere risolto in molti modi.

booleani ; notazione per l'assegnamento nello pseudocodice

Il codice userRPL è il seguente (*attenti a non spuntare "real tabs" in debug4x*):

```
«
  0
  →
  sum @variabile che ritornera' la somma totale, parte da 0
  «
   1 999 @estremi di variazione dell'indice
              @del for
   FOR i @definiamo l'indice usato nel for
    IF
     i 3 MOD @i mod 3
     0 == @i mod 3 e' pari a zero
     i 5 MOD
     0 ==
     OR @ se i mod 3 e' pari a zero o
        @ se i mod 5 e' pari a zero
    THEN
     sum i + @aggiungi i a sum
        'sum' STO @memorizza il nuovo valore
      END
   NEXT

   @mostra il risultato
   sum
  »
»
```

Analizziamo alcune delle strutture sintattiche usate nel programma (si consiglia la lettura dell'AUR da pag 1-13 a pag 1-22). Per prima cosa spieghiamo il funzionamento della struttura associata alla parola chiave (o *keyword*) IF. L'IF corrisponde ad un test su una condizione.

Pseudocodice IF THEN ELSE:

- IF
  - Condizione da soddisfare.
- THEN
  - istruzioni da eseguire se la condizione è soddisfatta.
- ELSE (opzionale)
  - istruzioni da eseguire se la condizione non è soddisfatta.
- END

La condizione da soddisfare può essere una *condizione atomica*, ad esempio, una condizione atomica sarebbe "uno è uguale ad uno?". Le condizioni atomiche sono, solitamente, quelle dove il risultato è un'unica risposta "Sì/No". Le *condizioni atomiche possono essere combinate in condizioni complesse*, che combinano più risposte "Sì/No" per ottenere una risposta "Sì/No" unica. Nel nostro caso si è usata una condizione complessa del tipo se " $i \, mod \, 3 = 0$ o $i \, mod \, 5 = 0$ ". Dal capitolo due (combinato con una certa pratica su funzioni semplici) si dovrebbe aver acquisito l'esperienza per capire perchè l'operazione $i \, mod \, 3$ viene scritta come `i 3 MOD` ; ripetendo brevemente: le funzioni sull'hp50g prendono gli argomenti dallo stack in un certo ordine, `i` e `3` sono gli argomenti e `MOD` è la funzione.

Le funzioni che connettono le risposte delle condizioni atomiche sono dette ***operatori booleani***. Questi sono AND, OR, NOT, XOR (vedere AUR 1-11 , 1-12 ; oppure qui alla sezione operatori). Invece all'interno delle condizioni atomiche possono essere usati gli *operatori di confronto* tipo "maggiore","uguale", ecc... vedere AUR 1-11.

Oltre all'IF c'è un'altra struttura sintattica che è data dalla parola chiave FOR . Il FOR ha diverse varianti, in questo caso abbiamo usato quella più semplice.

Pseudocodice FOR variabile NEXT:

- Si stabiliscono il valore iniziale ed il valore finale che l'indice del for deve assumere. Il primo numero è il valore iniziale, il secondo è il valore finale.
- FOR variabile (dove variabile indica l'indice del FOR)
  o istruzioni da eseguire nel blocco del FOR.
- NEXT

Come funziona?

1. Si prende la variabile del FOR e la si pone pari al valore iniziale, nel nostro caso: `i=1` ;
2. poi si eseguono le istruzioni del blocco del FOR;

3. dopodichè si incrementa il valore della variabile del for, quindi i:=i+1 [14] ;
4. si testa se il nuovo valore di i è maggiore del valore finale: se sì, si prosegue con le istruzioni dopo la keyword NEXT (il FOR è finito); se no, si ritorna al passo 2.

3.4.2. Efficienza

Come si era scritto all'inizio, i problemi del project euler sono pensati per essere risolti efficientemente se li si analizza dal punto di vista matematico (cosa che, ripeto, non faremo qui). L'hp50g ha una capacità di calcolo elevata ma, purtroppo, per mantenere la retrocompatibilità con programmi già sviluppati sulle calcolatrici precedenti deve emulare tutta una serie di istruzioni che la rallentano decisamente. Lo UserRPL viene emulato, includendo una notevole quantità di automatismi di controllo che contribuiscono a rallentare il codice. Per scrivere codice ottimizzato per il processore della calcolatrice si dovrebbe usare l'hpgcc3 almeno duecento volte più veloce dello userRPL.

Tuttavia ciò ha anche lati positivi, il problema appena affrontato è molto semplice dal punto di vista della programmazione "grezza" (senza ottimizzazioni) ma mostra tutti i suoi limiti su un hardware lento o malamente sfruttato (come quello dell'hp50g). Bisogna aspettare più di un minuto per ottenere il risultato sull'hp50g, questo potrebbe spingere il programmatore, prima di usare l'hpgcc, ad ottimizzare, se possibile, il suo algoritmo. Ricordate sempre che ***un algoritmo ottimizzato produce benefici su qualsiasi piattaforma di calcolo***, quindi fa ottenere un beneficio generale.

3.5. Problema 2 ; utilizzare altri programmi dall'interno di un programma ; simulare risposte Sì/No con numeri interi ;

[14] Notare che := è la notazione per l'assegnamento in pseudocodice. Cioè i:=i+1 significa "prendi il valore di i, incrementalo di uno e rimemorizzalo in i".

identificatore di un programma e posizione di questo

***Problema 2**: Data la sequenza di Fibonacci calcolata in modo da avere termini che non superino il valore di quattro milioni, dire qaunto vale la somma dei termini pari nella sequenza.*

Affrontiamo il problema a livello di pseudocodice, sempre in maniera grezza così da utilizzare più strutture sintattiche possibili. Possiamo pensarla così: prima calcoliamo l'ennesimo numero di fibonacci [15], poi verifichiamo se questo sia entro i limiti stabiliti, poi verifichiamo ancora che sia pari e nel caso lo aggiungiamo alla somma. Quindi:

Fibonacci1 (notare anche AUR da 2-1 a 2-4):

- Dato l'indice $n \geq 0$ del numero di fibonacci da calcolare
- Se $n \leq 1$
  - ritorna il valore n e termina l'esecuzione
- Altrimenti
  - Poni $fi \coloneqq 0$ e $fiplus1 \coloneqq 1$
  - $\forall k \in \{2, \dots, n\}$ (in questo caso si vuole dire, ripeti $n-2$ volte)
    - Poni $temp = fi + fiplus1$ (calcoliamo il k-esimo numero di Fibonacci)
    - assegna $fi \coloneqq fiplus1$, $fiplus1 \coloneqq temp$ (manteniamo in memoria il k-esimo ed il (k-1)-esimo numero di Fib.)
  - ritorna il valore $fiplus1$

Risolutore problema 2:

- Poni $n \coloneqq 0$, $somma \coloneqq 0$ e $menoDi4M \coloneqq 1$ (menoDi4M ***funge da booleano o da riposta "Sì/No"*** se è 0 è "No" se è 1 è "Sì")
- Finchè $menoDi4M = 1$
  - Poni $fDiN \coloneqq fibonacci(n)$ (chiamata ad un altro programma!)
  - Se $fDiN > 4milioni$
    - Poni $menoDi4M \coloneqq 0$
  - Altrimenti se $fDiN \bmod 2 = 0$

[15] Ricordando che la sequenza parte da zero, quindi è l'(n+1)-esimo.

        - Poni $somma \coloneqq somma + fDiN$
    - Ritorna $somma$

Si vede che nel risolutore del problema 2 c'è ***una chiamata ad un altro programma da noi creato***, ovvero: fibonacci(n) . Significa che all'algoritmo che implementa il programma fibonacci (può essere *Fibonacci1* o un altro) passiamo il valore n e ci aspettiamo di ottenere un valore di ritorno (che assegniamo a fDiN). In generale il valore di ritorno potrebbe anche non esserci.

Non è la prima volta che chiamiamo un programma all'interno del nostro programma, poichè in realtà ogni uso di funzioni già presenti nella calcolatrice, come MOD o + (la somma), equivale ad una chiamata ad un programma. MOD richiede due argomenti, somma pure. Ad esempio, usando la notazione corretta per mettere gli argomenti e la chiamata alla funzione nello stack: 5 3 MOD oppure 5 3 + sono chiamate a funzioni/programmi. La nostra chiamata sarà simile, cioè n fibonacci (o come vogliamo chiamarlo). L'unica differenza sarà nella posizione delle funzioni offerte dall'hp50g rispetto a quelle da noi create o installate. Le funzioni dell'hp50g sono memorizzate in librerie compatte (file unici) nella porta 2 (la memoria flash della calcolatrice). Le nostre funzioni (o programmi) sono solitamente memorizzate nella home (memoria ram della calcolatrice, senza batterie si cancella) o in cartelle all'interno della home. Per ora semplifichiamo le operazioni e manteniamo ogni nostra funzione nella home della calcolatrice. ***Ogni funzione o programma è identificato col nome con cui viene salvato*** e si può richiamare facendo uso del nome appena menzionato.

3.5.1. Sfruttare le proprietà dello stack per eseguire operazioni ; START NEXT ;

Iniziamo a vedere ora gli algoritmi. Occupiamoci prima del programma sui numeri di Fibonacci.

```
«
  0
  1
  →
  n @memorizza il numero di fibonacci da cercare
    @ottenuto da input
```

```
  fi @memorizza il valore 0
  fiplus1 @memorizza il valore 1
  «
    IF
      n 1 ≤ @n≤1
    THEN
      n @lasciamo n sullo stack
        @come valore di ritorno
    ELSE
      2 n
      FOR i @per i che va da 2 ad n
        fi fiplus1 +
          @calcoliamo fi+fiplus1
          @lasciandolo sullo stack
        fiplus1
          @lasciamo questo valore sullo stack
        'fi' STO
          @memorizziamo il valore
          @di fiplus1 (sullo stack) in 'fi'
          @che assorbe il livello 1 dello stack
        'fiplus1' STO
          @memorizziamo fi+fiplus1 in fiplus1
      NEXT

      @finito il for lasciamo sullo stack
      @il valore da tornare
      fiplus1
    END
  »
»
```

Si osserva che si è lavorato non poco con lo stack. Lo stack è stato introdotto nel capitolo 2 ma qui si spiegherà brevemente qualche concetto.

La struttura dello stack è come un'enorme pila di cassetti. Ogni cassetto contiene delle informazioni. Se si aggiunge un'informazione si aggiunge un cassetto sopra la pila. Se si consuma un'informazione si toglie un cassetto sempre da sopra la pila. Sulle calcolatrici HP lo stack è ordinato al contrario, il livello più basso (il cassetto sopra la pila) è numerato con l'etichetta 1 e poi via via che si scende nella pila di casetti il numero del livello aumenta.

Data questa struttura si possono effettuare delle ***operazioni sullo stack***, per scoprirle accendete la calcolatrice, cliccate TOOL, e poi STACK; dopodichè cercate le operazioni esposte nel menù nell'AUR. Alcune operazioni di uso comune sono lo SWAP, cioè si scambiano i primi due cassetti della pila. Sull'hp50g significa scambiare le informazioni presenti nel livello 1 con quelle presenti nel livello 2. Oppure l'operazione DUP, che implica la

duplicazione del cassetto in cima alla pila ponendo il duplicato sempre in cima. Quindi si duplica il livello 1 ottenendo livello 1 e livello 2 uguali.

Si può anche usare lo stack con accortezza sapendo che delle operazioni consumano un certo numero di livelli nello stack stesso. Ad esempio, nel programma mostrato, prima si aggiorna `fi`, consumando il valore `fiplus1` appena inserito nello stack. Poi si aggiorna `fiplus1` consumando il valore `fi` `fiplus1` + precedentemente inserito nello stack.

Per esempio analizziamo il codice che fa un pesante uso dello stack sui numeri di fibonacci presente nell'AUR (pag 2-3):

```
«
  →
  n
  «
    @ se n≤1 allora ritorna n
    IF n 1 ≤
    THEN n
    ELSE
      0 1 @poni questi valori nello stack
      2 n
      START
        DUP @copia il valore al livello 1
            @cioè copia fiplus1 rendendolo
            @il nuovo fi alla prossima iterazione
        ROT @ruota i primi 3 livelli nello stack
            @da
            @ 3: liv3
            @ 2: liv2
            @ 1: liv1
            @ si passa a
            @ 3 : liv1
            @ 2 : liv2
            @ 1 : liv3
            @ quindi si avrà
            @ 3 : copia liv 1 fatta da DUP
            @ 2 : liv2 (cioè fi)
            @ 1 : liv1 (fiplus1)
        + @somma fi e fiplus1
          @ottenendo il nuovo fiplus1
      NEXT

      SWAP @quando il ciclo termina si avrà
           @ 2: fib di n-1
           @ 1: fib di n
           @con lo swap si inverte l'ordine
      DROP @si cancella il livello 1
           @quindi sullo stack rimane solo
           @fib di n
    END
  »
```

»

In questo algoritmo si è usata la struttura iterativa ***START NEXT*** (AUR 1-17 e seguenti), che è identica al FOR indice NEXT, solo che non offre al programmatore un indice da utilizzare (e dovrebbe essere usata solo quando il programmatore non ha bisogno di indici durante l'iterazione, come nel nostro programma precedente).

Quindi, dopo aver sostituito nel programma da noi precedentemente scritto FOR i con START, possiamo passare alla scrittura del programma che fa la somma dei numeri di fibonacci effettivamente utili. Il programma per trovare i numeri di fibonacci lo chiamiamo fibsn (fib start next).

```
«
  0
  0
  1
  4000000
  →
  sum
  n
  menoDi4M
  fourM
  «
    WHILE
      menoDi4M 1 ==
    REPEAT
      n fibsn EVAL @si esegue fibonacci(n)
      IF
        DUP @si duplica il risultato di fib(n)
        fourM > @se è maggiore di 4Milioni
      THEN
        0 'menoDi4M' STO
        DROP @si pulisce il risultato
      ELSE
        IF
          DUP @si duplica il risultato di fib(n)
          2 MOD 0 ==
            @se il risultato è pari
        THEN
          'sum' STO+
            @si aggiunge il risultato sullo stack
            @ alla variabile sum
        ELSE
          DROP @si pulisce il risultato che non
               @è d'interesse
        END
        1 'n' STO+ @aumentiamo n
      END
    END

    sum
```

»
»

Si nota subito che i programmi definiti dall'utente hanno un funzionamento diverso dai programmi presenti nelle librerie della calcolatrice. Infatti bisogna prima richiamare il programma tramite il suo identificatore, usando: nomeProgramma o 'nomeProgramma' ; per poi valutare il programma tramite EVAL. Invece i programmi nelle librerie non hanno bisogno di essere valutati con EVAL.

3.6. Problema 3 ; scomporre un obiettivo in una composizione di sottoobiettivi ; efficacia ed efficienza di un algoritmo ;

> ***Problema 3**: Dato un numero, in particolare 600851475143, trovare i suoi fattori primi e selezionare il più grande tra questi.*

Vediamo come afrontare il problema, scomponendolo in sottoproblemi. Per prima cosa scriviamo un algoritmo (o programma) per individuare i fattori primi di un numero, dopodichè scriviamo un algoritmo per analizzare questi e scegliere il massimo. Dunque l'obiettivo si raggiunge ***componendo due sottoobiettivi***:

- trovare i fattori primi di un generico numero;
- trovare il massimo in una lista di numeri.

Affrontiamo il primo obiettivo, ci sono molti modi per individuare i fattori primi di un numero, qui useremo il più intuitivo e nel caso in cui l'hp50g impieghi troppo tempo per la nostra pazienza cercheremo di aumentare l'efficienza degi algoritmi avvicinandoci alla soluzione "elegante" (in senso matematico) con cui ogni problema del project euler può essere risolto. In sostanza ***quando scriviamo un algoritmo prima cerchiamo l'efficacia***, ovvero la risoluzione corretta al problema, ***e poi***, nel caso, ***l'efficienza***.

FattoriPrimi 1:

- Dato un numero n di cui si vogliono i fattori primi
- $divisore \coloneqq 1$, $listaDivisori = \{1\}$
- Fintantochè $n > 1$
  - $\forall\, divisore \in \{2, \dots, n\}$
    - Se $n \bmod divisore = 0$
      - $listaDivisori \coloneqq listaDivisori \cup \{divisore\}$
      - $n \coloneqq n/divisore$
      - esci dal ciclo $\forall$ in cui è racchiusa questa istruzione
- ritorna $listaDivisori$

Ora vediamo il secondo sottoobiettivo

Massimo elemento in una lista:

- Data una lista di elementi L
- Poni $max = L(1)$
- $\forall\, e \in L$
  - se $e > max$
    - Poni $max \coloneqq e$
- ritorna max

3.6.1. Codice ; uscita anticipata da strutture iterative ; DEBUG ; variabili locali e globali ; profiling per individuare algoritmi lenti ;

Per prima cosa affrontiamo il codice per trovare i fattori primi. In userRPL, purtroppo, non esiste l'istruzione ***break che permette di uscire anticipatamente da strutture iterative***. Si può simulare con un errore ma è brutta. Allora, poichè il FOR non ammette condizioni oltre a quella dell'indice e modificare l'indice durante il FOR non è il massimo, convertiamo il FOR in while nel caso ci serva uscire anticipatamente da una struttura iterativa.

Un altro tool molto importante per gestire programmi complessi è il ***DEBUG***, rinvio alla sezione nell'AUR (1-31 e seguenti).

```
«
  {1}
  2
  0
  →
  number @numero da fattorizzare
  factors
  d
  break
```

```
«
  WHILE
    number 1 >
  REPEAT
    WHILE
      d number ≤
      break 1 ≠
      AND @finchè d è minore del numero
          @e non bisogna uscire anticipatamente
    REPEAT
      IF
        number d MOD
        0 == @ number MOD d == 0
      THEN
        d 'factors' STO+ @aggiunto in lista
        'number' d STO/ @number := number/d
        1 'break' STO @uscita anticipata
      ELSE
        1 'd' STO+ @d+1
      END
    END

    2 'd' STO @reset d
    0 'break' STO @reset break
  END

  factors
»
»
```

Questo programma restituisce i fattori primi del numero, dall'ultimo trovato (che sarà il primo della lista) al primo (cioè 1). Potremmo notare che, per come è costruita la lista di fattori primi ed il programma, i fattori primi più piccoli saranno sempre trovati prima dei fattori primi più grandi, quindi il massimo sarà sempre in prima posizione della lista. Tuttavia questa conoscenza non ci interessa (perchè dobbiamo sfruttare i problemi per vedere come si programma in userRPL[16]) e dunque vediamo il programma che trova il massimo in un lista di numeri

```
«
  -1
  0
  →
  numberList @lista di numeri
  max @max := -1 per convenzione
  n @numero di elementi nella lista
  «
```

[16] Da notare che si cerca di usare poche funzioni già fornite dall'HP, proprio per vedere come districarsi con il solo userRPL in certi problemi. Avremmo potuto usare la funzione NEXTPRIME ad esempio.

```
    numberList LIST→
      @scompatta la lista in elementi sfusi
      @e lascia sul primo livello il
      @numero di elementi della lista
    'n' STO @salviamo il numero di elementi
            @della lista

    @sfruttiamo il fatto che gli elementi
    @sono sullo stack per valutarli man mano

    @salviamo il primo sullo stack come max
    'max' STO
    'n' 1 STO- @n:=n-1 , ne abbiamo consumato uno

    WHILE
      n 0 > @fnchè ci sono elementi
    REPEAT
      IF
        DUP @dupliciamo il livello 1
        max >
          @se l'elemento corrente sullo stack
          @ è maggiore di max
      THEN
        'max' STO @lo salviamo
      ELSE
        DROP @lo cancelliamo
      END
      'n' 1 STO- @n:=n-1
    END

    max
  »
»
```

Un'osservazione sulle variabili locali e globali (AUR 1-7 e seguenti). Le variabili dopo la freccia sono definite come variabili locali al programma, finito il programma lo spazio di memoria che occupano viene liberato. Le variabili locali vanno definite prima delle istruzioni vere e proprie del programma, come se fossero argomenti di una funzione. Tuttavia niente ci vieta di definire variabili nel bel mezzo del programma, questo si può fare usando, ad esempio, l'istruzione valore 'nomeVar' STO. Questa operazione, però, definisce una variabile globale che esisterà anche dopo la terminazione del programma e potrebbe sovrascrivere variabili precedentemente memorizzate nella memoria della calcolatrice. Inoltre l'uso di variabili globali rallenta (di poco) l'esecuzione del programma stesso. Poi esiste un altro tipo di variabili, quelle "compilate", che vedremo in seguito.

Infine combiniamo i due programmi.

```
«
  600851475143
  'findFactors1' EVAL
```

```
  'maxNumberInList' EVAL
»
```

Eseguendo l'algoritmo notiamo che questo è lentissimo, almeno sull'hp50g. Bisogna, allora, velocizzarlo. Tuttavia non tutti i pezzi dell'algoritmo complessivo sono lenti in modo inaccettabile, allora bisogna individuare quelli lenti per concentrarsi nel migliorare solo quelli e non tutto l'algoritmo. Questa tecnica si chiama ***profiling***, e mostra il tempo impiegato per eseguire dei pezzi selezionati di codice. Il profiling con lo userRPL è manuale, e viene fatto usando il comando TICKS ed il semplice programmino esposto a pagina 3-253 dell'AUR. Il nostro obiettivo sarà prendere i tempi dell'esecuzione del sottoproblema "trova fattori" ed i tempi del sottoproblema "trova il massimo", per fare questo diminuiamo la grandezza del numero da scomporre (ma non troppo) e poi prendiamo i tempi. In pratica dobbiamo modificare opportunamente l'algoritmo che fa uso degli altri due algoritmi come segue:

```
«
  TICKS @start time findFactors
  @7876356017 991*2399*3313
  149 257 331 * * @12674983
  'findFactors1' EVAL
  TICKS @end time di findfactors

  @sullo stack si ha
  @ 3: start time
  @ 2: risultato findFactors
  @ 1: end time
  UNROT
  @ 3: end time
  @ 2: start time
  @ 1: risultato findFactors
  UNROT
  @ 3: risultato findFactors
  @ 2: end time
  @ 1: start time
  - B→R 8192 /
    @ritorna il tempo in secondi

  SWAP @mettiamo il risultato di findFictors
       @al livello uno
  TICKS @start time maxNumberInList
  SWAP @come prima
  @ 3: elapsed time findF
  @ 2: start time maxN
  @ 1: res findF
  'maxNumberInList' EVAL
  TICKS @end time maxNumberInList

  @ 4: elapsed time findF
```

```
  @ 3: start time maxN
  @ 2: res maxN
  @ 1: end time maxN
  UNROT UNROT @ come prima
  - B→R 8192 /
»
```

Si ottiene che findFactors1 richiede 82 secondi, mentre maxNumberInList richiede 0.3 secondi. E' ovvio che bisogna rendere più efficiente findFactors. Tra le diverse possibili ottimizzazioni si sceglie la seguente: facendo uso della funzione offerta dall'hp NEXTPRIME, che restituisce il numero primo successivo ad uno dato (anche se quello corrente è un numero primo, si ottiene il successivo), invece di incrementare il valore d di un'unità la volta lo incrementiamo fino al prossimo numero primo.

```
«
  {1}
  2
  0
  →
  number @numero da fattorizzare
  factors
  d
  break
  «
    WHILE
      number 1 >
    REPEAT
      WHILE
        d number ≤
        break 1 ≠
        AND @finchè d è minore del numero
            @e non bisogna uscire anticipatamente
      REPEAT
        IF
          number d MOD
          0 == @ number MOD d == 0
        THEN
          d 'factors' STO+ @aggiunto in lista
          'number' d STO/ @number := number/d
          1 'break' STO @uscita anticipata
        ELSE
          d NEXTPRIME @prossimo primo maggiore di d
          'd' STO @sarà il nuovo d
        END
      END

      2 'd' STO @reset d
      0 'break' STO @reset break
    END

    factors
  »
»
```

In termini di codice chiamiamo questo algoritmo findFactors2, in termini di programma sulla calcolatrice lasciamo findFactors1 per non dover modificare diversi file (l'importante è sapere che il codice è salvato in un file diverso dal codice di findfactors1).

Otteniamo che il nuovo findFactors richiede solo 19 secondi, è ancora ampiamente migliorabile ma per ora supponiamo che basti ed eseguiamo il vero problema che infatti viene risolto in un tempo ragionevole.

3.7. Problema 4

> ***Problema 4**: dati due numeri di tre cifre moltiplicati tra loro, trovare il massimo risultato del prodotto che sia palindromo.*

A differenza dei problemi precedenti, dove si sono sviluppati algoritmi abbastanza generici (validi per valori di input diversi da quelli posti dal problema) qui, per pigrizia, limiteremo l'algoritmo a trattare solo numeri generati dal prodotto di numeri a tre cifre. Questo porta dei vantaggi (per la risoluzione del problema). Ad esempio il prodotto di numeri a 3 cifre genera un numero di sei cifre e quindi possiamo individuare due sottoproblemi:

- Dire se un numero a 6 cifre è palindromo;
- Trovare il massimo numero a 6 cifre palindromo.

Vediamo lo pseudocodice:

E' palindromo?:

- Dato n di 6 cifre
- Rendi questo numero una stringa s
- $\forall i \in \{1, \dots, 3\}$
  - Prendi il carattere iniziale della stringa s_i e rimuovilo dal resto dela stringa, ottenendo $s := s - s_i$
- I caratteri iniziali sono le varie cifre del numero, cioè s_1, s_2, s_3 si concatenano in maniera inversa, ottenendo $s_3 s_2 s_1$ e si vede se questa stringa combacia con la stringa rimanente s
  - Se sì ritorna 1 (vero)
  - Altrimenti 0 (falso)

Risolutore problema 4:

- Poni $max \coloneqq 0$
- $\forall\, i \in \{999, \ldots, 100\}$
  - $\forall\, j \in \{i, \ldots, 100\}$
    - Se $i \cdot j$ è palindromo e $i \cdot j > max$
      - $max \coloneqq i \cdot j$
      - Esci dal ciclo
- ritorna max

L'unica piccola ottimizzazione introdotta sin da subito è la seguente: j parte da i perchè se partisse da 999 ripeterebbe prodotti già fatti; infatti così si ha: 999*999, 999*998, ..., 999*100, 998*998, 998*997, ..., 998*100, 997*997, ..., 100*100 . Ciò ci permette di ripetere prodotti tipo: 998*999, 997*999, 997*998 e così via.

3.7.1. Codice ; controllo flag ; manipolazione stringhe ; flags per salvare valori booleani ; FOR STEP ; non è sicuro usare "i" come identificativo per le variabili

Per la ***manipolazione delle stringhe*** vedere l'UG capitolo 23. Per le flag bisogna vedere l'UG capitolo 24 dove si menzionano le ***user flags utilissime per implementare valori "Sì/No"*** (o booleani) senza dover usare variabili locali aggiuntive.

Vediamo il codice per vedere se un numero è palindromo

```
«
  →
  sixDigitN
  «
    @salviamo le flag correnti
    PUSH
    @puliamo la flag 105
    @poichè ci interessano numeri esatti senza
    @punto seppur interi
    -105 SF @-105 poichè è una flag di sistema
            @e queste vanno da -1 a -128

    sixDigitN XQ →STR
      @si trasforma il numero in una stringa
      @ma prima lo si porta in forma esatta
      @nel caso avesse il punto aggiunto
      @dalla modalita' "approx"
```

```
    @tra l'altro XQ setta la flag -105 nuovamente
  DUP @si duplica la stringa
  HEAD @si prende c_1
  SWAP @si riprende la stringa
  TAIL @si toglie alla stringa il
       @primo carattere
  DUP
  HEAD @c_2
  SWAP
  TAIL
  DUP
  HEAD @c_3
  SWAP
  TAIL
  @ situazione stack
  @ 4: c_1
  @ 3: c_2
  @ 2: c_3
  @ 1: stringa rimanente, se palindr.
  @       c_3c_2c_1
  4 ROLLD @sposto la stringa rimanente
          @al livello 4 e faccio salire il
          @resto delle cifre
  SWAP + @c_3c_2 il + sulle stringhe, concatena
  SWAP + @c_3c_2c_1

  IF
    == @se le stringhe sono uguali
  THEN
    1
  ELSE
    0
  END

  @reimpostiamo le flag prima salvate
  POP
»
»
```

Da notare che le flag di sistema, visibili dal menù "mode" con i valori da 1 a 128, in realtà ***vanno richiamate con valori uguali a quelli del menù mode ma con segno negativo***. Inoltre si può osservare l'uso di operazioni che manipolano stringhe, come HEAD e TAIL.

Dopodichè rimane da vedere il codice che testa i vari numeri, ovvero il risolutore del problema 4:

```
«
  0
  →
  maxV
  «
    999 100
    FOR k
      k 100
      FOR j
        IF
```

```
          k j * DUP
          isPalindrome
            @se k*j è palindromo
          SWAP
          maxV > @k*j maggiore di max
          AND
        THEN
          k j * 'maxV' STO
        END
      -1 STEP
    -1 STEP

    maxV
  »
»
```

Nel codice sovrastante si nota l'uso della struttura iterativa ***FOR STEP***. Il funzionamento è simile al FOR NEXT solo che invece di modificare la variabile, menzionata dopo la keyword FOR, aumentandola di uno, la si modifica aumentandola (o diminuendola) del valore che precede la keyword STEP. Se l'aumento è positivo, allora si eseguirà il ciclo fintantochè il valore attuale della variabile è minore o uguale del valore finale dato come input del FOR; se l'aumento è negativo, viceversa, si eseguirà il ciclo fintantochè il valore attuale della variabile è maggiore o uguale del valore finale dato come input del FOR.

Inoltre si vede che non si è usato i come indice del FOR. perchè? perchè i è (anche) usata dal sistema della calcolatrice per individuare la parte immaginaria di un numero complesso e quindi è meglio non usarlo come identificatore di una variabile.

Come potrete notare questo codice impiega tantissimo tempo per eseguire, almeno sull'hp50g usando lo userRPL, quindi dobbiamo introdurre delle ottimizzazioni. Senza andare troppo a fondo nel lato matematico del problema cerchiamo di mantenere la stessa struttura tentando però di sfruttare la seguente idea: se si testassero i prodotti in modo tale da testare prima i prodotti più grandi (che era l'idea iniziale, per questo motivo si inizia da 999 con gli indici) saremmo sicuri che, una volta trovato un prodotto palindromo, questo sia il più grande di tutti quelli rimanenti. L'idea allora è la seguente, invece di mantenere fisso un fattore e diminuire l'altro, vediamo il problema al contrario fissando un fattore e diminuendo l'altro fino al valore del fattore fissato. In sostanza avremo il cambiamento del ciclo da:

- $\forall\, k \in \{999, \ldots, 100\}$
  o $\forall\, j \in \{k, \ldots, 100\}$

a:

- $\forall\, k \in \{999, \ldots, 100\}$
  - $\forall\, j \in \{999, \ldots, k\}$

In questo modo si avranno i prodotti: 999*999, 999*998, 998*999, 999*997, 998*997, 997*997, ..., 101*100, 100*100 .

Ora, non è garantito che il prodotto appena eseguito sarà più grande di tutti quelli ancora da eseguire, ma è garantito che se il prodotto corrente, il più grande palindromo trovato, dato da $k_1 \cdot j_1$ è più grande di $999 \cdot (j_1 - 1)$ allora sarà più grande di tutti i possibili prodotti palindromi rimanenti. Questo perchè $k_1 \cdot j_1$ è il massimo palindromo ottenuto valutando i prodotti $k \cdot j$ con $k \in \{999, \ldots, j_1\}$ e $j \in \{999, \ldots, j_1\}$ quindi non c'è modo di ottenere un valore più grande avendo da una parte il termine massimo, 999 , e dall'altra un termine per forza più piccolo di j_1 (perchè il secondo termine 999 a j_1 è stato già valutato).

Dunque in pseudocodice diventa:

- Poni $max \coloneqq 0$
- $\forall\, k \in \{999, \ldots, 100\}$
  - Se $max > 999 \cdot k$
    - Esci dal ciclo $\forall\, k$
  - $\forall\, j \in \{999, \ldots, k\}$
    - Se $i \cdot j$ è palindromo e $i \cdot j > max$
      - $max \coloneqq i \cdot j$
- ritorna max

In codice userRPL si ha:

```
«
  0
  999
  999
  →
  maxV
  k
  j
  «
    10 CF @la flag 10 indica se abbiamo
          @trovato il massimo palindromo o meno
    WHILE
      k 100 ≥
      10 FC? @se k ≥ 100 AND F10==0
      AND
    REPEAT
      IF
        maxV 999 k * > @maxV > 999*k
      THEN
```

```
      10 SF  @ F10 := 1
    ELSE
      11 CF @impostiamo una flag per uscire
            @dal ciclo al primo palindromo
            @trovato tenendo fisso k e
            @variando su j
      999 'j' STO @ reset j
    END
      WHILE
        j k ≥
        10 FC?
        AND
        11 FC?
        AND @se j ≥ k AND F10==0 AND F11==0
      REPEAT
        IF
          k j * DUP
          isPalindrome
            @se k*j è palindromo
          SWAP
          maxV > @k*j > maxV
          AND
        THEN
          k j * 'maxV' STO
          11 SF @ F11 := 1
        END

        'j' 1 STO- @j:=j-1
      END

      'k' 1 STO-
  END

  maxV
»
»
```

Notare l'uso delle flag riservate all'utente, molto comode.

3.8. Problema 5 ; != "non uguale" ;

***Problema 5**: qual è il più piccolo numero che sia divisibile da ogni numero compreso tra 1 e 20.*

Anche se nei problemi precedenti non si è mai analizzato il problema a livello matematico, se non giusto per scrivere l'algoritmo, questo è proprio facile. Basta trovare il minimo comume multiplo tra tutti i numeri compresi tra 1 e 20 e si ottiene il più piccolo numero divisibile tra uno e venti.

Tuttavia a noi interessa armeggiare con la sintassi dello userRPL quindi rendiamoci la vita difficile ed affrontiamo il problema in maniera grezza.

Risolutore problema 5:

- Dato un numero n
- Poni èMultiplo:=0 e c_n:=n+1
- Fintantochè èMultiplo = 0
  - Poni èMultiplo:=1
  - Per k da 1 a n sempre se èMultiplo = 1
    - Se c_n mod k != 0 (!= non uguale)
      - Poni èMultiplo:=0
  - Se èMultiplo=0
    - c_n:=c_n+1
- ritorna c_n

3.8.1. Codice ; commenti sui valori iniziali delle variabili locali ; differenze tra variabile senza apici e 'variabile' ; identificatori ammissibili per variabili

Il codice è il seguente:

```
«
  1 @c_n
  1 @k
  →
  n
  cn
  k
  «
    n 'cn' STO+ @cn:=n+1
    10 CF @f10:=0 , f10 : eMultiplo

    WHILE
      10 FC? @f10 == 0
    REPEAT
      10 SF @eMul := 1
      1 'k' STO @reset k:=1

      WHILE
        k n ≤
        10 FS?
        AND @f10 == 1 AND k<=n
      REPEAT
        IF
          cn k MOD
          0 ≠ @cn mod k != 0
        THEN
          10 CF @no Mul
        ELSE
          'k' 1 STO+
        END
```

```
    END

    IF
      10 FC? @ eMul == 0
    THEN
      'cn' 1 STO+ @next number
      @else f10==1 and the while stops
    END
  END

  cn
»
»
```

Qui osserviamo alcune cose, che prima abbiamo tralasciato. La prima è: se si usano diverse variabili locali assegnando a queste dei valori arbitrari iniziali (vedere "cn" e "k"), è opportuno, per leggere più velocemente il codice, ***commentare il valore da assegnare ad una data variabile*** indicando a quale variabile sarà assegnato. In precedenza si era commentato al contrario (azione poco utile), ovvero indicando quale valore avrebbe avuto una variabile, inserendo il commento subito dopo la variabile stessa.

La seconda cosa è ***la differenza del richiamo di una variabile*** tramite il suo semplice identificatore o tramite il suo identificatore contenuto tra apici (tipo 'k'). Se si richiama la variabile senza apici, si mette sullo stack, subito, il suo contenuto. Insomma si ha la seguente equivalenza: k è uguale a 'k' EVAL .

La terza nota riguarda ***i nomi ammissibili per gli identificatori di variabili*** (o di programmi). Devono iniziare con una lettera, minuscola o maiuscola (dell'alfabeto latino o greco), dopo la prima lettera ammettono altre lettere e numeri ma non ammettono altri caratteri. Ad esempio c_n non è ammissibile come identificativo di una variabile.

Infine si ha che l'esecuzione del codice, sull'hp50g, è davvero lenta. Questi problemi hanno il pregio di far scoprire dei dettagli sul linguaggio (qualora lo sviluppatore non conosca molto il linguaggio, come me) ed inoltre mettono una voglia matta di sfruttare meglio l'hw della calcolatrice tramite l'uso di programmi scritti in codice nativo con hpgcc. Su questo ci sono pro e contro, tuttavia, che prima o poi esporrò[17].

---

[17] A fronte di un guadagno di velocità, con hpgcc, si perdono le librerie offerte dall'hp e quindi bisogna trovare le librerie apposite in C da

Per ora andiamo avanti sfruttando al massimo l'userRPL, l'ottimizzazione degli algoritmi e le librerie fornite dall'HP, sfruttando proprio la funzione per il calcolo del minimo comune multiplo.

- Dato n come numero finale dell'intervallo 1, ..., n ; posto lcm:=1
- Per k da 1 ad n
  - Poni lcm:=LCM(lcm,k)
- ritorna lcm

```
«
  1 @lmc
  →
  n
  lcm
  «
    1 n
    FOR k
      k lcm LCM 'lcm' STO
    NEXT

    lcm
  »
»
```

Velocità non dico istantanea ma quasi, sicuramente molto più accettabile dei minuti rispetto all'algoritmo di prima.

3.9. Problema 6 ; cambio di font per lo pseudocodice ;

Problema 6: *trovare il valore della seguente differenza* $\sum_{i=1}^{100} i^2 - (\sum_{i=1}^{100} i)^2$.

Il problema si risolve facilmente sapendo le formule chiuse delle somme del tipo $\sum_{i=1}^{n} i^k$, dove $\sum_{i=1}^{n} i$ è la nota sommatoria di

---

integrare nel proprio progetto. Non che sia una cosa difficile ma è meno immediata. Inoltre un programma C è molto più pesante in termini di spazio occupato (proprio perchè non è interpretato ed ha tutte le istruzioni "ready to run"). Non è un problema memorizzarlo nella scheda SD che può contenere migliaia di programmi scritti in C, anche complessi; ma è un problema quando viene caricato in ram per eseguire. Una programma per hp50g occupa in ram almeno quanto occupa su disco, e la ram nell'hp50g è poca (circa 500kb complessivamente).

Gauss. Da notare che $(\sum_{i=1}^{n} i)^2 = \sum_{i=1}^{n} i^3$. Tuttavia noi procediamo per il metodo grezzo, ben sapendo, ormai, che l'hp50g è abbastanza lenta con lo userRPL se si devono fare calcoli intensivi.

Nota: da ora lo pseudocodice userà perlopiù il font usato per il testo normale (LM mono), usarò il saxMono (codice) o il cambria math (simboli matematici) solo quando sarà richiesta maggiore chiarezza.

Risolutore problema 6:

- Dato n come numero finale della somma da 1 ad n. Poni sum:=0 e sumSQ:=0
- Per k da 1 ad n
  - Poni sum:=sum+k e sumSQ:=sumSQ+k^2
- Ritorna sum^2 - sumSQ

3.9.1. Codice

```
«
  0 @sum
  0 @sumSQ
  →
  n
  sum
  sumSQ
  «
    1 n
    FOR k
      k 'sum' STO+
      k k * 'sumSQ' STO+
    NEXT

    sum sum * sumSQ -
  »
»
```

Stavolta il problema è stato abbastanza generoso, in termini di grandezza dell'input, da permettere anche all'hp50g con userRPL di risolverlo quasi immediatamente.

3.10. Problema 7

Problema 7: *escludendo uno, se due è il primo numero primo, trovare il decimilleunesimo (10 001st) numero primo.*

Questo è un compito assai arduo per l'hp50g, con un algoritmo grezzo non c'è speranza di ottenerlo in qualche minuto. Dobbiamo usare NEXTPRIME ed eventualmente pensare ancor di più alla struttura matematica del problema.

Risolutore problema 7:

- Dato l'n-esimo numero da trovare
- Per k da 1 ad n
  - calcola il k-esimo primo
- ritorna l'n-esimo primo trovato con k:=n .

3.10.1. Codice

```
«
  1 @cp
  →
  n
  cp
  «
    1 n
    FOR k
      cp NEXTPRIME 'cp' STO
    NEXT

    cp
  »
»
```

La speranza è che questo codice sia veloce, usando NEXTPRIME il centesimo primo lo si trova in meno di un secondo (contro i minuti richiesti dall'algoritmo menzionato prima), quindi facciamo affidamento alla libreria hp. Infatti si termina in un tempo abbastanza ragionevole. (si potrebbe ottimizzare sviluppando noi una funzione PRIME cumulativa, mettiamo un segnalibro per ricordarci in futuro, evidenziazione verdognola!)

3.11. Problema 8 ; gestire numeri come stringhe per maneggiarne facilmente le cifre

Problema 8: *Dato un numero di 1000 cifre (scritto in seguito) trovare il più grande prodotto formato da 5 sue cifre consecutive.*

Il numero è:

7316717653133062491922511967442657474235534919493496983520312774506326239578318016984801869478851843858615607891129494954595017379583319528532088055111254069874715852386305071569329096329522744304355766896648950445244523161731856403098711121722383113622298934233803081353362766142828064444866452387493035890729629049156044077239071381051585930796086670172427121883998797908792274921901699720888093776657273330010533678812202354218097512545405947522435258490771167055601360483958644670632441572215539753697817977846174064955149290862569321978468622482839722413756570560574902614079729686524145351004748216637048440319989000889524345065854122758866688116427171479924442928230863465674813919123162824586178664583591245665294765456828489128831426076900422421902267105562632111110937054421750694165896040807198403850962455444362981230987879927244284909188845801561660979191338754992005240636899125607176060588611646710940507754100225698315520005593572972571636269561882670428252483600823257530420752963450

L'idea per trovarlo si basa sulla manipolazione delle stringhe. Gestire ***le cifre di un numero è più facile se si tratta questo come una*** stringa. Dunque l'algoritmo in pseudocodice è il seguente:

- Dato il numero 'n' come stringa
- Prendi le prime 5 cifre di 'n' (a partire da sinistra) e poni prod:=*prodottoDelleCinqueCifre* , maxProd:=prod, ed inoltre c_1:=*primaCifraDiN*
- Per k da 6 a 1000
  - Estrai la cifra k-esima, detta c_k da 'n'
  - Poni prod:= $prod \cdot c_k / c_1$ (cioè il massimo prodotto è dato dal prodotto corrente delle cinque cifre, ottenuto togliendo la prima cifra del prodotto precedente ed aggiungendo la nuova)
  - Se $prod > maxProd$
    - maxProd:= $prod$
- Ritorna maxProd

E' interessante, in questo caso, gestire l'input del problema. Protremmo fissare una variabile locale nel programma pari all'intero numero ma non sappiamo se l'hp50g gestisce numeri così lunghi senza modificare qualcosa (per quanto ne so sì), quindi lo memorizziamo come una stringa compresi i caratteri di ritorno. Insomma la stessa stringa che abbiamo esposto sopra. Quindi andiamo in debug4x e copiamo il numero mettendolo tra doppi apici, poi lo portiamo sull'emulatore e verrà convertito in una stringa. Il carattere di ritorno a capo, se si passa una stringa dopo averla elaborata con l'emulatore di debug4x è "linefeed" che è associato al carattere 10 (AUR j-1 e seguenti, UG capitolo 23)

3.11.1. Codice ; preferire la leggibilità alla velocità tramite azioni sullo stack ; eguaglianza tra pseudocodice e codice ;

Per prima cosa ci serve un piccolo programmino che dato un numero in forma numerica ne calcola il prodotto delle sue cifre:

```
«
  1
  →
  numStr
  prod
  «
  WHILE
    numStr
    SIZE 0 >
  REPEAT
    numStr HEAD @prende la prima cifra
    numStr TAIL 'numStr' STO
      @la toglie dal resto del numero
    NUM @la trasforma in numero
    48 - @si ottiene il vero numero
    'prod' STO*
  END

  prod
  »
»
```

Il programma sopra esposto è digitMul e viene usato dal risolutore del problema 8 (***leggermente diverso dallo pseudocodice che comunque serve a dare un'idea abbastanza precisa dell'algoritmo***):

```
«
  @l'input è salvato come "inputProblem8"
  "0" @prodStr (perche' è una stringa vedi codice)
```

```
0 @maxProd
" " @remainingString
48 @l'associazione numerica ai caratteri indica
   @che 48 è associato con 0 e così via fino a
   @57 che è 9
→
prodStr
maxProd
remStr
charConst
«
  inputProblem8 'remStr' STO
    @memorizziamo la stringa in input
    @nella stringa rimanente

  @"121212023040599999684 8" 'remStr' STO
    @per test

  @prendiamo le prime 5 cifre del numero
  @e le togliamo dal numero stesso
  1 5
  FOR k
    remStr HEAD
    remStr TAIL 'remStr' STO
  NEXT
  + + + +
    @concateniamo le prime 5 cifre
    @che sono viste ancora come stringhe

  'prodStr' STO
    @salviamo il prodotto
    @come stringa poiche' nel caso dia
    @zero perderemmo le varie cifre

  @salviamo il prodotto delle cifre in maxProd
  prodStr digitMul EVAL
  'maxProd' STO

  WHILE
    remStr
    SIZE 0 >
      @fintantochè ci sono caratteri da elaborare
  REPEAT
    remStr HEAD
    remStr TAIL 'remStr' STO
    @abbiamo il primo carattere della stringa
    @nello stack
    NUM @il codice numerico associato al carattere
    IF
      DUP @duplichiamo il codice num
      10 ≠ @se non è linefeed
    THEN
      @allora e' un numero e modifica
      @prodStr
      CHR @inverso di NUM
      prodStr TAIL
      SWAP + 'prodStr' STO
        @concateniamo la cifra al prodotto
```

```
        @compariamo il prodotto corrente
        @al maxProdotto
        prodStr digitMul EVAL
        IF
          DUP
          maxProd >
        THEN
          DUP 'maxProd' STO @nuovo max
        END
        DROP @puliamo il calcolo
      ELSE
        @se è linefeed
        DROP @lo cancelliamo dallo stack
      END
    END

    maxProd
  »
»
```

Il codice è molto meno efficiente dello pseudocodice perchè non si usa l'osservazione di dividere per la prima cifra del prodotto precedente e moltiplicare per la nuova cifra, però dovrebbe terminare in un tempo ragionevole. Si può notare come ***si è evitato di fare un uso pesante delle operazioni sullo stack, cercando di richiamare il più possibile*** (ma non troppo) ***le variabili locali***. Questo, anche se rende leggermente meno veloce l'esecuzione, rende più leggibile il programma, che così sarà più facile da modificare serve.

Invece nell'esecuzione ci ha messo un pochetto, questo perchè elaboriamo spesso (con comandi come TAIL, SIZE, HEAD) stringhe molto lunghe.

3.12. Problema 9 ; sulle triple pitagoriche

> ***Problema 9:*** *Esiste un'unica tripla* (a,b,c) *con* $a,b,c \in \mathbb{N}$ *tale che* $a<b<c$, $a+b+c=1000$ *e* $a^2+b^2=c^2$ *(detta pitagorica). Trovarla e calcolare* $a \cdot b \cdot c$.

Stavolta il problema, nella versione grezza, è computazionalmente oneroso. Potremmo provare, ad esempio, così: testiamo se le seguenti triple che danno come somma mille rispettano la condizione di tripla pitagorica: Fissiamo prima $a \coloneqq 1$, $b \coloneqq 2$ e $c \coloneqq 997$. Poi diminuendo 'c' (finchè questo si mantiene maggiore di 'b') aumentiamo 'b' e facciamo i test. Se questi test non hanno successo allora aumentiamo 'a' e ripetiamo. Quindi partiamo da $a \coloneqq 2, b \coloneqq 3, c \coloneqq 995$. Il numero dei tentativi

dovrebbe essere nell'ordine (approssimo per eccesso) di 500^3 che sono molti per l'efficienza dello userRPL sull'hp50g. Per diminuirne il numero osserviamo[18] che 'c' non può essere molto più grande di 'b' ed 'a' altrimenti non si ottiene il quadrato. Anzi deve essere vero che $a+b>c$. Infatti se così non fosse, supponiamo $c>a+b$, allora potremmo dire che $c=a+b+k,\ k\in\mathbb{N}$. Ma allora $a^2+b^2=(a+b+k)^2$ e ciò è falso. Neanche $c=a+b$ è possibile, poichè se così fosse $a^2+b^2=(a+b)^2$ ed è assurdo anche questo.

Con questa conoscenza possiamo togliere di mezzo diverse triple, in particolare possiamo partire dal primo risultato utile per avere $a+b=501$ rispettando gli altri vincoli "di costruzione": $a\coloneqq 3, b\coloneqq 498, c\coloneqq 499$. In questo modo l'hp50g procede abbastanza velocemente (sempre se si può aspettare qualche minuto).

3.12.1. Codice

```
«
 3 @a
 498 @b
 499 @c
 →
 a
 b
 c
 «
  1 CF @f1==0 non abbiamo trovato la tripla

  WHILE
    1 FC? @finchè la tripla non è trovata
  REPEAT
    IF
      a a * b b * +
      c c *
      == @a^2 + b^2 = c^2
    THEN
      1 SF @abbiamo trovato la tripla
    END

    IF
      1 FC? @se non si è trovata la tripla
    THEN
      IF
        @se non possiamo più incrementare
        @b (è ≥ di c) e non abbiamo trovato la tripla
        b 1 + c 1 - ≥ @b+1 > c-1
      THEN
        @abbiamo finito i tentativi per
```

[18] Stiamo, quindi, analizzando meglio il problema, cosa che spesso evitiamo di fare.

```
        @il dato a, aumentiamo a e ripartiamo
        @settando gli altri valori di conseguenza
        'a' 1 STO+
        501 a - 'b' STO
        499 'c' STO
      ELSE
        @continuiamo il tentativo con l'a fissato
        @b+1 ; c-1
        'b' 1 STO+
        'c' 1 STO-
      END
    END
  END

  @risultato
  a b c
  a b * c *
 »
»
```

Tuttavia è possibile (basta accedere al solo thread di discussione del problema stesso) risolvere il problema anche a mano in non troppi passaggi, ma gli algoritmi qui proposti non sono affatto ottimizzati!