بسمه تعالی

گزارش‌کار زبان اسمبلی و ماشین

Full Pyramid Pattern

استاد

دکتر قدیری

نویسندگان

جواد باجلان

حسین ابراهیمی

دانشگاه شهید مدنی آذربایجان

پاییز 1403

فهرست

عنوان **صفحه**

مقدمه..... ۳
الگوریتم..... ۴
پیاده‌سازی..... ۷
کد کامل..... ۱۳
تصاویر..... ۱۶

مقدمه

الگوی هرم کامل (***Full Pyramid Pattern***) یکی از الگوهای محبوب در برنامه‌نویسی است که به صورت یک ساختار مثلثی چاپ می‌شود. این الگو معمولاً از کاراکترهای خاص مانند * یا دیگر کاراکترها تشکیل شده و در هر خط، تعداد مشخصی از کاراکترها نمایش داده می‌شود تا شکل یک هرم به ارتفاع n ایجاد شود.

ویژگی اصلی این الگو در تقارن آن است؛ به این صورت که هر خط از هرم نسبت به مرکزیت خود دارای تعداد یکسانی فاصله (space) و کاراکتر (*star*) است. این تقارن باعث می‌شود که الگو به‌صورت زیبا و متوازن نمایش داده شود.

الگوی هرم کامل یک مثال عالی برای یادگیری مفاهیم پایه‌ای در برنامه‌نویسی است، مانند:

- **حلقه‌های تو در تو:** برای چاپ خطوط و کنترل تعداد کاراکترها در هر خط
- **شرط‌ها:** برای تعیین موقعیت کاراکترها (ستاره یا فاصله)
- **مدیریت ورودی:** برای ساخت الگوهای انعطاف‌پذیر بر اساس ورودی کاربر.

کدهای نوشته شده در ادامه، مربوط به اسمبلر ***EMU۸۰۸۶*** برای سیستم‌عامل ***DOS*** است.

الگوریتم

شرح مسئله

هدف از این الگوریتم، تولید و نمایش یک الگوی هرم کامل در خروجی است که با استفاده از کاراکترهایی نظیر $star$ و $space$ ساخته می‌شود. ورودی الگوریتم، عددی است که مشخص می‌کند هرم چه ارتفاعی داشته باشد. هر خط از هرم دارای تعداد مشخصی از کاراکترها و فاصله‌ها است تا شکل متقارن هرم ایجاد شود.

الگوریتم تولید الگوی هرم کامل را می‌توان به مراحل زیر تقسیم کرد:

۱. **دریافت ورودی:**

از کاربر یک عدد صحیح n به عنوان تعداد خطوط هرم دریافت می‌شود.

۲. **محاسبه تعداد ستون‌ها:**

برای ارتفاع n، هر خط دارای تعداد ستون‌های معادل $2n - 1$ است. این محاسبه به منظور تنظیم محدوده کاراکترهای چاپی انجام می‌شود.

۳. **ساخت حلقه‌های تو در تو:**

- **حلقه بیرونی:** تعداد خطوط n را مدیریت می‌کند.
- **حلقه داخلی:** مسئول چاپ فاصله‌ها ($space$) و ستاره‌ها (*) برای هر خط است:

فاصله ها:تعداد فاصله‌ها در هر خط با فرمول $n - i$ به طوری که $1 \le i \le n$ شماره خط است یا عبارتی دیگر $n - i$ خانه اول از $2n - 1$ خانه هر خط.

ستاره‌ها: تعداد ستاره‌ها در هر خط با فرمول $2i - 1$ محاسبه می‌شود یا به عبارتی دیگر تا قبل از $(n - i) + (2i - 1) + 1 = n + i$ امین خانه و آخرین فاصله باید ستاره پر شود.

۴. **چاپ خطوط:**

برای هر خط، ابتدا تعداد مناسب فاصله‌ها پرینت می‌شود پس از آن تعداد ستاره‌ها پرینت می‌شود و بعد از تکمیل هر خط، یک خط جدید پرینت می‌شود یعنی مجموعا $n^2 + \frac{n(n+1)}{2} = \frac{3n^2 + n}{2}$ کارکتر به ازای ارتفاع n چاپ میشود.

۵. **اتمام الگوریتم**

مثال:

تعداد سطرها به اندازه $n = 5$

تعداد ستون‌ها به اندازه $2n - 1 = 2(5) - 1 = 9$

$n - i$ خانه اول هر سطر برابر با فاصله و بین $n - i$ و $n + i$ امین خانه ستاره‌ها

شبه‌کد:

```
n <- input
      m := (n * 2) - 1
      for i := 0 -> n
            for j := 1 -> m + 1
                  if j < n - i
                        output <- SPACE
                        continue
                  else if j <= n + i
                        output <- *
                        continue
                  break
            output <- line_break
```

```
SI <- input
DX := (SI * 2) - 1
for BX := 0 -> SI
      for CX := 1 -> DX + 1
            if CX < SI - BX
                  output <- SPACE
                  continue
            else if CX <= SI + BX
                  output <- *
                  continue
            break
      output <- line_break
```

پیاده سازی

در ابتدا کارکترهای *star* و *space* را در *data segment* تعریف و ذخیره می‌کنیم.

```
data SEGMENT
    star:                DB '*', 24h
    space:               DB ' ', 24h
    return:              DB 0Dh, 0Ah, 24h
       invalid_input:  DB 0Dh, 0Ah, 'Invalid input', 0Dh, 0Ah, 24h
ENDS
```

چون هر کارکتر *ASCII* هفت بیت فضا اشغال می‌کند، تعریف هر کدام به عنوان یک بایت کافی ست.

0Dh معادل کد *ASCII carriage return* است که *cursor* را به اول خط باز می‌گرداند.

0Ah معادل کد *ASCII new line* است که *cursor* را به خط بعدی می‌برد.

به معنای *24h* که معادل کارکتر $ در *ASCII* است، در ادامه پرداخته می‌شود.

سپس سگمنتی برای پشته (*stack segment*) تعریف می‌کنیم:

```
stack SEGMENT
    DW   32  dup(?)

ENDS
```

64 بایت برای پشته برای این برنامه کافی ست.

حال به پیاده‌سازی سگمنت کد (*code segment*) و پیاده‌سازی الگوریتم می‌پردازیم.
در ابتدای الگوریتم یعنی

```
n <- input
```

ارتفاع هرم را از کاربر به عنوان عددی صحیح دریافت می‌کنیم.
بدین منظور از *API* مربوط به سیستم‌عامل *DOS*[1] استفاده خواهیم کرد. سیستم عامل *DOS*، *API* هایی برای ارتباط با *I/O* در نظرگرفته است که از طریق وقفه `21h`[2] در دسترس هستند.
API مورد نظر ما `INT 21/AH=01h`[3] خواهد بود، کارکتری را از ورودی استاندارد می‌خواند و در رجیستر `AL` ذخیره می‌کند. بایستی هر رقم از عدد ورودی را به عنوان کارکتر خوانده و سپس آن را به معادل عدد صحیح تبدیل نمود.

بدین منظور فرض کنید کاربر رشته `123` را وارد کرده است. ابتدا `BX=0000h` تنظیم میشود.
`AL=31h` سپس آن را از معادل اسکی `0` کم نموده و سپس `BX=BX*10+AL` یعنی `BX=1` . مکررا `AL=32h` و سپس کم کردن `30h` از آن و `BX=BX*10 + AL` یعنی `BX=12` و `AL=33h` و کم کردن `30h` از آن و `BX=BX*10+AL` یعنی `BX=123`.
همچنین بایستی شروطی را چک کرد، برای مثال اگر کلید *Enter* وارد شد، به دریافت ورودی پایان دهد و یا اگر ورودی غیر از اعداد بودند، خطایی نمایش دهد.

با همه این تفاسیر به رویه زیر می‌رسیم:

```
catch_n:
    XOR BX, BX          ; set BX to zero
    _read_num:
        MOV AH, 01h     ; AH = 01h
        INT 21h         ; call DOS

        CMP AL, 0Dh     ; check if return
        JE _on_return


        XOR AH, AH      ; set AH to zero
        SUB AL, 30h     ; AL = AL - (the 0 in ASCII)

        JS _on_error    ; if the result is negative (SF=1) then it's not a digit
        CMP AL, 9       ; check if it's greater than 9 then it's not a digit
        JG _on_error

        MOV CX, BX      ; BX = CX

        SHL CX, 03h     ; CX = CX << 3 = CX * 2^3
        SHL BX, 01h     ; BX = BX << 1 = BX * 2^1
        ADD BX, CX      ; BX = BX + CX = BX * (2^3 + 2^1) = BX * 10

        ADD BX, AX      ; BX = BX + AX


        JMP _read_num ; do next loop iteration

    _on_return:       ; if the input is return then return the number in AX
        MOV AX, BX    ; AX = BX
```

[1] https://www.ctyme.com/intr/cat-010.htm
[2] https://www.ctyme.com/intr/int-21.htm
[3] https://www.ctyme.com/intr/rb-2552.htm

```
    RET         ; return to the previous IP


_on_error:      ; if the input character is not a digit then return 0 as error
    XOR AX, AX  ; set AX to zero
    RET         ; return to the previous IP
```

دلیل استفاده از دو عملیات Shift Left (SHL) و جمع آن‌ها این است که این روش سریع‌تر از ضرب (MUL) عمل می‌کند. در مقایسه، اجرای یک دستور MUL به طور متوسط حدود 114 سیکل پردازنده بیشتر از دو شیفت و یک جمع زمان می‌برد.

همچنین استفاده از XOR BX,BX به جای MOV BX,0 نیز به این دلیل است.

حال به سراغ پیاده‌سازی رویه‌ای برای پرینت کارکترها می‌رویم و از *DOS* می‌خواهیم رشته‌ای را نمایش دهد. بدین منظور از[4] INT 21/AH=09h استفاده خواهیم کرد.

این *API* آدرس یک رشته در DS:DX که به بایت 24h یا معادل اسکی $ ختم شده باشد را صفحه نمایش می‌دهد.

بنابراین:

```
echo_string:
    PUSH AX      ; save previous value of AX to stack
    PUSH DX      ; save previous value of DX to stack

    MOV DX, AX   ; DX = AX
    MOV AH, 09h  ; AH = 09h
    INT 21h      ; call dos

    POP DX       ; restore previous value of DX from stack
    POP AX       ; restore previous value of AX from stack

    RET          ; just return to the previous IP
```

حال که رویه های کمکی را نوشت ایم به سراغ پیاده سازی اصل الگوریتم میرویم :

```
main:
    MOV BX, stack    ; use the stack segment as default for SS
    MOV SS, BX

    MOV BX, data     ; use the data segment as default for DS
    MOV DS, BX

    CALL catch_n     ; call catch_n to get height of the pyramid
    TEST AX, AX      ; AND between AX and itself. if the result is zero then ZF=1
    JZ _error        ; jump to _error if AX=0 (invalid inputs)

    MOV SI, AX       ; AX = SI

    MOV AX, return    ; print newline
    CALL echo_string

    MOV DX, SI        ; DX = SI
    SAL DX, 1         ; DX = DX << 1 = DX * 2
    INC DX            ; DX = DX + 1

    XOR BX, BX        ; set BX to zeros
```

[4] https://www.ctyme.com/intr/rb-2562.htm

```
_loop1:
    CMP BX, SI              ; continue if BX < SI is meet
    JGE _end                ; break the first loop if BX >= SI

    MOV CX, 1               ; CX = 1
    _loop2:
        CMP CX, DX          ; break the second loop if CX > DX
        JG _continue_loop1

        MOV DI, SI          ; DI = SI
        SUB DI, BX          ; DI = DI - BX = SI - BX

        CMP CX, DI          ; print space if CX < DI or CX < SI - BX
        JL _space

        MOV DI, SI          ; DI = SI
        ADD DI, BX          ; DI = DI s BX = SI + BX

        CMP CX, DI          ; break the second loop if CX > DI or CX > SI + BX
        JG _end_loop1       ; or print star if CX <= SI + BX

        MOV AX, star            ; print star
        CALL echo_string
        JMP _continue_loop2     ; do next second loop iteration

        _space:
            MOV AX, space       ; print space
            CALl echo_string

        _continue_loop2:
            INC CX              ; CX = CX + 1 and do the second loop again
            JMP _loop2
    _continue_loop1:
        MOV AX, return          ; print newline
        CALL echo_string

        INC BX                  ; BX = BX + 1 and do the first loop again
        JMP _loop1
_error:
    MOV AX, invalid_input       ; print error
    CALL echo_string
_end:
    MOV AX, 4C00h               ; terminate the process with return code 00h
    INT 21h
```

دریافت ورودی از کاربر:

```
CALL catch_n
TEST AX, AX
JZ _error
```

از آن‌جایی که رویه کمکی `catch_n` را به صورتی تعریف کردیم که در ورودی غیر از عدد صحیح، صفر برگرداند، به این ترتیب بعد از فراخوانی `catch_n` مقدار رجیستر `AX` را بررسی کرده که از صحیح بودن مقدار آن مطمئن شویم و در غیر این صورت با پرینت کردن ارور به برنامه پایان می‌دهد.

همچنین بعد از ورودی کاربر نیاز است به خط بعدی برویم تا مشکلی در نمایش هرم پیش نیاید:

```
MOV AX, return
CALL echo_string
```

```
MOV DX, SI
SAL DX, 1
INC DX
```

برای محاسبه تعداد ستون‌ها که همان $m := (n * 2) - 1$ در شبه‌کد است، از یک شیفت به چپ و `INC` استفاده می‌کنیم.

به منظور نگه‌داری ردیف، رجیستر `BX` را برابر صفر قرار می‌دهیم (معادل i در شبه‌کد):

```
XOR BX, BX
```

حلقه اول یعنی `for i:=0->n` در شبه‌کد که ابتدای حلقه را با لیبل `loop1_` مشخص می‌کنیم:

```
_loop1:
        CMP BX, SI
        JGE _end
```

همچنین شرط حلقه یعنی `i<n` را در ابتدا بررسی می‌کنیم و در صورت عدم برقراری، به حلقه پایان می‌دهیم و عملیات تکرار `i+=1` را در پایان هر دور حلقه اجرا می‌کنیم.

حلقه تودرتو دوم یعنی `for j:=1->m+1` در شبه‌کد که ابتدای حلقه را با لیبل `loop2_` مشخص می کنیم:

```
MOV CX, 1
_loop2:
      CMP CX, DX
      JG _continue_loop1
```

همچنین شرط حلقه یعنی `j<m+1` را در ابتدا بررسی می‌کنیم و در صورت عدم برقراری، به حلقه پایان می‌دهیم و دور بعدی حلقه قبلی یعنی `loop1_` را اجرا می‌کنیم و عملیات تکرار `j+=1` را در پایان هر دور حلقه اجرا می‌کنیم.

```
MOV DI, SI
SUB DI, BX

CMP CX, DI
JL _space
```

در این قسمت از کد اسمبلی به قسمت

```
if j < n - i
      output <- SPACE
```

در شبه‌کد اشاره دارد که حاصل $n - i$ محاسبه شده و با j مقایسه می‌گردد و در صورت کوچک‌تر بودن j به `space_` پرش انجام می‌دهد و کارکتر فاصله را نمایش می‌دهد.
همچنین مانند قبل برای کارکتر ستاره خواهیم داشت:

```
MOV DI, SI
ADD DI, BX

CMP CX, DI
```

```
JG _end_loop1

MOV AX, star
CALL echo_string
```

همانند شبه‌کد

```
else if j <= n + i
      output <- *
```

با این تفاوت که شرط عکس را چک می‌کند و در صورت عدم برقراری شرط `j<=n+i` به حلقه تودرتوی دومی پایان می‌دهد همانند `break` که در شبه‌کد موجود است.

و سپس کد ادامه حلقه دوم را خواهیم داشت:

```
JMP _continue_loop2


_continue_loop2:
      INC CX
      JMP _loop2
```

که همان عملیات تکرار `j += i` و انجام دوباره حلقه دوم است.

هم‌چنین برای حلقه دوم خواهیم داشت:

```
_continue_loop1:
      MOV AX, return
      CALL echo_string

      INC BX
      JMP _loop1
```

که مربوط به پرینت کردن خط جدید در انتهای پردازش هر خط و عملیات تکرار `i+=1` و انجام دوباره حلقه اول است و در انتها نیز خاتمه دادن به پروسه برنامه از طریق اطلاع دادن به سیستم‌عامل DOS از طریق *API*،[5] `INT 21/AH=4Ch` که کد وضعیت را در `AL` دریافت می‌کند و به محیط با این کد از وضعیت خاتمه برنامه اطلاع می‌دهد:

```
MOV AX, 4C00h
INT 21h
```

که ما از کد وضعیت `0` به معنای اتمام موفقیت‌آمیز برنامه استفاده نمودیم.

[5] https://www.ctyme.com/intr/rb-2974.htm

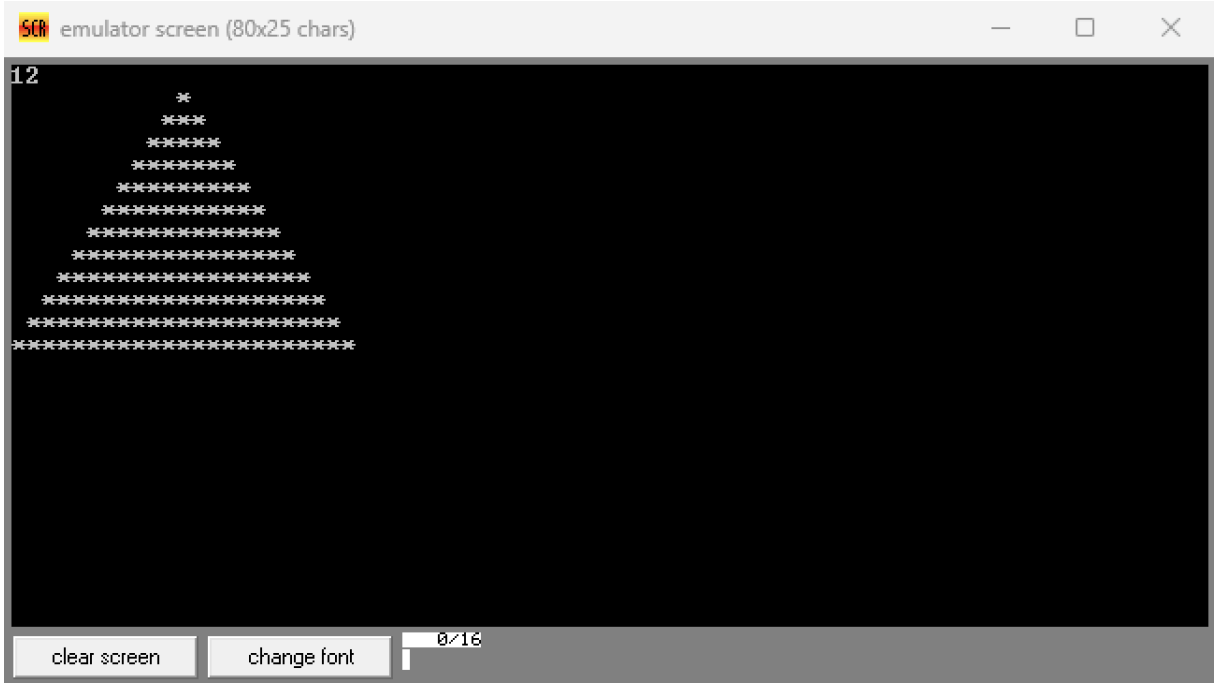
کد کامل(نهایی)

```
data SEGMENT
    star:           DB '*', 24h
    space:          DB ' ', 24h
    return:         DB 0Dh, 0Ah, 24h
    invalid_input:  DB 0Dh, 0Ah, 'Invalid input', 0Dh, 0Ah, 24h

ENDS



stack SEGMENT
    DB   32  dup(?)

ENDS



code SEGMENT
    ASSUME CS:code,SS:stack,DS:data

main:

    MOV BX, stack
    MOV SS, BX

    MOV BX, data
    MOV DS, BX

    CALL catch_n
    TEST AX, AX
    JZ _error

    MOV SI, AX

    MOV AX, return
    CALL echo_string

    MOV DX, SI
    SAL DX, 1
    INC DX

    XOR BX, BX

    _loop1:
        CMP BX, SI
        JGE _end


        MOV CX, 1
        _loop2:
            CMP CX, DX
            JG _end_loop1
```

```
            MOV DI, SI
            SUB DI, BX

            CMP CX, DI
            JL _space

            MOV DI, SI
            ADD DI, BX

            CMP CX, DI
            JG _end_loop1


            MOV AX, star
            CALL echo_string
            JMP _end_loop2

            _space:
                MOV AX, space
                CALl echo_string

            _end_loop2:
                INC CX
                JMP _loop2

        _end_loop1:
            MOV AX, return
            CALL echo_string

            INC BX
            JMP _loop1
    _error:
        MOV AX, invalid_input
        CALL echo_string
    _end:
        MOV AX, 4C00h
        INT 21h

catch_n:
    XOR BX, BX
    _read_num:
        MOV AH, 01h
        INT 21h

        CMP AL, 0Dh
        JE _on_return



        XOR AH, AH
        SUB AL, 30h

        JS _on_error
        CMP AL, 9
        JG _on_error

        MOV CX, BX
```

```
        SHL CX, 03h
        SHL BX, 01h

        ADD BX, CX
        ADD BX, AX


        JMP _read_num

    _on_return:
        MOV AX, BX
        RET

    _on_error:
        XOR AX, AX
        RET



echo_string:
    PUSH AX
    PUSH DX

    MOV DX, AX
    MOV AH, 09h
    INT 21h

    POP DX
    POP AX

    RET


ENDS


END main
```

تصاویر

ورودی(ارتفاع هرم) = ۱۲

خروجی :