Esercitazione 5: Complessità e Poly-reduction

Informatica Teorica 23/24; Docente: Fabio Zanasi; Tutor: Gabriele Lobbia*

17 Aprile 2024

Oggi faremo degli esercizi sulla complessità degli algoritmi, provandola direttamente con python e poi studiando la poly-reduction.

1 Esercizi: Misurare la Complessità (∼1h)

Prerequisiti. Complessità di tempo, notazione asintotica Big-O.

Esercizio 1.1 (5/10m)**.** Data $f(n) = 2n^3 + 4n^4$ qual è x tale che $f(n) = O(x)$? Dimostrare dando esplicitamente $c, m \in \mathbb{N}$ tali che, per ogni $n \geq m$, $f(n) \leq c \cdot g(n)$.

Somma di due interi

Esercizio 1.2 (10m)**.** Per sommare due numeri interi di n cifre, l'algoritmo è:

1. Scrivi i due numeri uno sopra all'altro, in colonna.
2. Posizionati sulla colonna più a destra.
3. Poi somma le due cifre della colonna su cui sei posizionato.
4. Se il risultato è minore di dieci, scrivilo nella riga sotto e nella stessa colonna.
5. Altrimenti scrivi lì l'ultima cifra del risultato e spostati a sinistra "riportando uno" cioè aggiungendo 1 alla successiva addizione.
6. Se ci sono ancora cifre a sinistra, spostati di una colonna a sinistra e vai al passo tre.
7. Altrimenti: finito.

Supponendo che i due numeri da sommare siano lunghi al più n cifre, quanti passi richiede l'operazione? (per "passo" intendiamo la somma di una cifra). Determinare quindi la complessità computazionale in notazione Big-O.

Correzione esercizi insieme

Prodotto di due numeri interi

Ora vedremo come cambia la complessità al variare dell'algoritmo usato.

Esercizio 1.3 (15m, Metodo delle somme ripetute)**.** Per *definizione*, moltiplicare due numeri X e Y significa sommare Y volte X:

$$X \times Y = \underbrace{X + \cdots + X}_{Yvolte}$$

In Python possiamo scrivere (vedi file *MultAddRipetute*)

```
def add_ripetute(X,Y):
    risultato = 0
    for i in range(Y):
        risultato += X
    return risultato
```

*Email: gabriele.lobbia@unibo.it

(i) Qual è la complessità computazionale (sempre, d'ora in poi, in notazione Big-O) di questo algoritmo? Giustificare la risposta.

(ii) Sperimentare quanto ci mette l'algoritmo eseguendo il programma con python. Supponendo che il codice precedente sia contenuto in un modulo *MultAddRipetute.py* , provare da shell i comandi

python3 -m timeit -n 1 -r 1 "from MultAddRipetute import add_ripetute;
r = add_ripetute(pow(10,3),pow(10,3)); print(r);"

python3 -m timeit -n 1 -r 1 "from MultAddRipetute import add_ripetute;
r = add_ripetute(pow(10,6),pow(10,6)); print(r);"

Cosa si ottiene?

(iii) Assumendo che in un anno ci sono $3.15e + 7$ secondi e che in un secondo siano eseguiti 100 passo elementare di somma di una cifra, in quanto tempo viene eseguito il seguente comando?

python3 -m timeit -n 1 -r 1 "from MultAddRipetute import add_ripetute;
r = add_ripetute(pow(10,9),pow(10,9)); print(r);"

Esercizio 1.4 (15m, Metodo della moltiplicazione scolastica)**.** La moltiplicazione che si insegna a scuola e' descritta dalla seguente funzione python3 (vedi file $Mult_sums$)

```python
from MultAddRipetute import add_ripetute;
import math
def mult_sums(X,Y):
        risultato = 0
        cifre = int(math.log10(Y))+1 # numero di cifre di Y

        def ennesima_cifra(numero, n):
                return numero // 10**n % 10

        for i in range(cifre):
                y = ennesima_cifra(Y,i)
                r = add_ripetute(X,y)
                risultato += (r * pow(10,i))

        return risultato
```

(i) Descrivere in linguaggio naturale l'algoritmo implementato nel codice python precedente. Qual'e' la complessità computazionale di questo algoritmo?

(ii) Supponendo che il codice precedente sia contenuto in un modulo $Mult_sums.py, provaredashellilcomandoseg$

python3 -m timeit -n 1 -r 1 "from $Mult_sumsimportmult\_sums$;
r = mult_sums(pow(10,9),pow(10,9)); print(r);"

Ripetere anche per la moltiplicazione di 10^3 e 10^6 come nell'esercizio precedente.

Esercizio 1.5 (15m, Metodo della moltiplicazione di Karatsuba)**.** Fino a non molto tempo fa (1960) si riteneva che $O(n^2)$ fosse la complessità computazionale della moltiplicazione. Ma nel 1962 Anatoly Karatsuba scoprì un algoritmo che abbassa la complessità computazionale a $O(n^{log_2 3})$.

(i) Perché è meglio?

Questo è l'algoritmo di Karatsuba implementato in Python (vedi file *Karatsuba*):

```
def KaratsubaMultiply(x, y):
        if x < 10 and y < 10:
                return x*y
        sx=str(x)
        sy=str(y)
        m = max(len(sx), len(sy))
        sx='0'*(m-len(sx))+sx
        sy='0'*(m-len(sy))+sy
        m1=int((m+1)/2)
        m2=int(m/2)
        lox=int(sx[m1:])
        hix=int(sx[:m1])
        loy=int(sy[m1:])
        hiy=int(sy[:m1])
        z0 = KaratsubaMultiply(lox,loy)
        z1 = KaratsubaMultiply((lox+hix),(loy+hiy))
        z2 = KaratsubaMultiply(hix,hiy)
        return (z2*10**(2*m2))+((z1-z2-z0)*10**m2)+(z0)
```

Il test:

```
python3 -m timeit -n 1 -r 1 "from Karatsuba import karatsuba_multiply;
r = karatsuba_multiply(pow(10,9),pow(10,9)); print(r);"

python3 -m timeit -n 1 -r 1 "from Karatsuba import karatsuba_multiply;
r = karatsuba_multiply(pow(10,12),pow(10,12)); print(r);"

1000000000000000000
1 loop, best of 1: 3.13 msec per loop
1000000000000000000000000
1 loop, best of 1: 686 usec per loop
```

(ii) Confrontare i tempi dell'algoritmo di Karatsuba con quello della moltiplicazione scolastica.

Curiosità: L'interprete Python addotta la moltiplicazione di Karatsuba per numeri molto grandi, vedi:

1. `https://www.wikiwand.com/it/Algoritmo_di_Karatsuba`;
2. `https://www.wikiwand.com/en/Multiplication_algorithm`.

Correzione esercizi insieme

2 Esercizi: Poly-reduction (∼45m)

Problema "Hitting Set"

L'Hitting Set Problem consiste nel, data una famiglia di insiemi finiti $\{S_1, S_2, \ldots, S_n\}$ trovare, se esiste, il minimo (in cardinalità) insieme H che interseca ogni S_i. In altre parole, vogliamo $H \cap S_i \neq \emptyset$ per ogni i con $|H|$ più piccola possibile. Per esempio, si pensi all'insieme delle bandiere delle nazioni, ove in ogni bandiera è presente un insieme di colori (bianco, rosso, verde per l'Italia). L'"hitting Set" è il più piccolo iniseme di colori che sono presenti in tutte le bandiere.

L'obiettivo di questa sezione è dimostrare che questo problema è in **NP** attraverso la poly-riduzione. Per fare ciò, introduciamo un altro problema, ovvero MAXSAT, che è una generalizzazione di KSAT (SAT per formule dove ogni clausola contiene al più K letterali) adatta a problemi di ottimizzazione.

Per definire MAXSAT iniziamo descrivendo le formule in WCNF: queste sono coppie (c, w) con $c \in$ CNF (i.e. congiunzione di seguenze di disgiunzioni) e $w \in \mathbb{Z}$ un peso naturale. Il problema MAXSAT (massima soddisfacibilità) consiste nel, data una formula $F \in$ WCNF, trovare l'assegnazione di valori di vertià che la renda vera massimizzando i valori delle clausole con valore 1.

Esercizio 2.1 (30m)**.** Trovare una poly-reduction dal problema "Hitting Set" al problema MAXSAT con formule WCNF.
Suggerimento: Supponi che per ogni famiglia di insiemi $\{S_1, \ldots, S_n\}$ esista un *insieme universo* E dove vivono tutti gli S_i, i.e. per ogni $i = 1, \ldots, n$, $S_i \subseteq E$.
Suggerimento 2: Gli elementi dell'insieme H corrisponderanno alle variabili a cui assegnamo valore di verità 1, come facciamo a costruire una formula così?

Correzione esercizi insieme

Test in pySat

Si può affrontare questo problema anche con python, usando pySat, che sta per *SAT technology in Python* (dove SAT indica proprio la Boolean satisfability). Per farlo installare pySat seguendo le istruzioni a questo link.

Per farlo, definiano prima una funzione che genera famiglie di insiemi, adatte a provare la classe Hitman del modulo pysat.examples.hitman. In Python, vedi file *generaTest* in cui è riportato il codice sotto.

```
import random
import math

NUMERO_INSIEMI_DEFAULT = 100
NUMERO_ELEMENTI_DEFAULT = 1000
def get_numero_max_elementi_per_insieme(numero_elementi):
        return math.floor(math.log(numero_elementi))

def genera_insiemi(numero_insiemi = NUMERO_INSIEMI_DEFAULT,
        numero_elementi = NUMERO_ELEMENTI_DEFAULT,
get_numero_max_elementi_per_insieme=get_numero_max_elementi_per_insieme):
        insiemi = []
        numero_max_elementi_per_insieme
get_numero_max_elementi_per_insieme(numero_elementi)
        print(f"numero max elementi per insieme :
    {numero_max_elementi_per_insieme}")
        for i in range(numero_insiemi):
```

```
            CARDINALITA_INSIEME =
random.randint(1,numero_max_elementi_per_insieme)
            insieme = set()
            for e in range(CARDINALITA_INSIEME):
            insieme.add(random.choice(range(numero_elementi)))
                insiemi.append(insieme)
    return insiemi

if __name__ == "__main__":
    print(genera_insiemi())
```

Poi usiamo pySat (vedi file *HittingSets*).

```
from pysat.examples.hitman import Hitman
from generaTest import genera_insiemi
import math
def funzioneLog10(numero_elementi):
    return int(math.log10(numero_elementi))

sets = genera_insiemi(
    numero_insiemi=pow(10,3),
    numero_elementi=pow(10,5),
    get_numero_max_elementi_per_insieme=funzioneLog10)

print("Inizia test...")
h = Hitman(bootstrap_with=sets, solver='m22', htype='lbx')
hittingSet = h.get()
print(f"hittingSet: {hittingSet}")
print(f"numero elementi: {len(hittingSet)}")
```

Si possono variare i parametri

numero_insiemi, *numero_elementi* o anche *get_numero_max_elementi_per_insieme*

per provare l'algoritmo, che non ha complessità polinomiale e quindi per valori dei parametri $\geqslant 10^7$ di *numero_elementi* diventa impraticabile.

3 Esercizi: NP-completezza (∼30m)

Esercizio 3.1 (10m)**.** Consideriamo due linguaggi L e L'. Dimostrare che se L è **NP**-completo, $L' \in \mathbf{NP}$ e $L \leq_p L'$, allora anche L' è **NP**-completo.

Esercizio 3.2 (10m)**.** Assumi $3SAT \in \mathbf{NP}$ e che ogni formula Booleana F ammette una riscrittura in $3CNF$ equivalente F^3 (F e F^3 hanno lo stesso valore di verità per ogni assegnazione di valori di verità alle variabili) con tempo di computazione polinomiale. Dimostra che è **NP**-completo.

Correzione esercizi insieme