ESERCIZI MODELLI E LINGUAGGI

- Enunciare e discutere la tesi di Church-Turing, chiarendo perché viene formulata come tesi e non come teorema.

Tutto ciò che possiamo calcolare lo possiamo calcolare con una MdT (o con un sistema di calcolo equivalente). Si tratta di una tesi perché non si è dimostrato che non esistano porblemi calcolabili con strumenti di calcolo diversi.

(P!=NP. Il che, poiché sappiamo P⊆NP, equivarrebbe ad avere P⊂NP. Si tratta di una tesi perché non sappiamo se i problemi che attualmente sappiamo risolvere come NP siano risolvibili anche come P. Tuttavia poiché nonostante gli studi vi sono molti problemi che continuano ad essere NP, congetturiamo che essi non siano più facili di NP e che dunque valga la tesi sopra.)

- Definire le classi: P, NP e indicare le relazioni di contenimento fra le classi.

La classe P è l'insieme di tutti i linguaggi per cui esiste una mdT deterministica che in tempo polinomiale li decide.
La classe NP è l'insieme di tutti i linguaggi per cui esiste una mdT non deterministica che in tempo polinomiale li decide.
P⊆NP, ma si congettura P⊂NP.

- Definire la classe dei problemi NP-completi e indicare le relazioni di contenimento rispetto alle classi P e NP.

La classe dei linguaggi NP-completi è l'insieme di tutti i linguaggi L NP tali che qualsiasi altro linguaggio in NP è polinomialmente riducibile a L.
P⊆NP⊃NPc

- Indicare un possibile risultato che permetterebbe di stabilire che P ≠ NP.

Dimostrare che un particolare problema NP non è P.

- Indicare un possibile risultato che permetterebbe di stabilire che P = NP.

NP -> P

- Per dimostrare che un problema A è NP-completo cosa è necessario dimostrare?

Che A è NP e che qualsiasi altro problema in NP è riducibile ad A. Per il secondo punto è spesso comodo ridurre un altro problema NP completo ad A.

- Definire il problema della soddisfacibilità di una formula logica e cosa è una formula in forma normale congiuntiva (CNF). A quale classe di complessità temporale appartiene il problema della soddisfacibilità di una formula CNF?

SAT: Data una formula logica mi chiedo cerco un insieme di valori di verità che la renda vera.

Una formula è detta in forma normale congiuntiva (CNF) se è in forma normale negativa e inoltre ha la forma C1ΛC2Λ...ΛCn (o {C1, C2, …, Cn}) dove Ci sono le clausole (la clausola generica è AVBVnotCV...).
Una formula è in forma normale negativa se il segno di negazione compare solo davanti agli atomi.
SAT è NP-completo.

- Definire il problema della soddisfacibilità di una formula logica e cosa è una formula in forma normale congiuntiva (DNF). A quale classe di complessità temporale appartiene il problema della soddisfacibilità di una formula DNF?

Mat: La forma normale disgiuntiva é una forma logica per la quale le clausole sono separate da OR (invece che AND) e all'interno sono legate da AND (invece che OR) tipo: (A AND B AND C) OR (-D AND F AND B) OR ...
La sua risoluzione puó avvenire in tempo polinomiale (infatti basta risolvere una singola clause per risolvere tutto). Il punto é che al momento non é conosciuto un algoritmo che converte una forma CNF in una DNF in tempo polinomiale (solo esponenziale), quindi il SAT di una CNF non puó essere polinomialmente ridotto ad una DNF e quindi DNF appartiene a P e CNF a NP.

- Nel caso di problemi NP-completi non rinunciamo a cercare soluzioni. Illustrare un possibile approccio per il problema della soddisfacibilità di formule logiche.

È possibile definire un automa non deterministico che prova tutte le possibili combinazioni e accetta se ne trova una valida. Peraltro tale automa può essere simulato con una MdT deterministica che effettui backtracking.

DPLL é un ottimo algoritmo usato per il problema SAT quando questo é in forma CNF { (A OR B OR C) AND (D OR F OR -A) AND .. }. Esso sfrutta il principio del backtracking per trovare una soluzione, piú una serie di regole euristiche per migliorare l'algortitmo:
- Unit propagation: É il principio per il quale se una variabile sta da sola in una parentesi allora quella parentesi si puó rendere vera se la variabile é settata a true (o false se la variabile é negata).
- Pure-literal elimination: se una variabile appare in tutta la formula sempre positiva (o sempre negata) allora basta settarla a true (o false se negata) per ottenere il risultato ottimale.
Infine la scelta di quale variabile esaminare di volta in volta (pickVar()) é molto importante, ma cambia di implementazion in implementazione.
L'algoritmo sará :

```
function DPLL(F) {
        if (F.hasOnlyPureLiterals()) return true;
        if (F.hasEmptyClause()) return false;
        for var in F.pureLiterals() :
                var.setRight();
                var.removeFromClause();
        for var in F.unitClauses() :
                var.setRight();
```

```
                var.removeFromClause();

        x = pickVar(F);
        return DPLL(F (x.setTrue()) || DPLL(F (x.setFalse());
```

Esso appunto prova prima settando la variabile a true e se non dovesse funzionare prova con il false. I tentativi fatti si possono rappresentare benissimo con un albero binario.

- Nel caso di problemi di ottimizzazione che sono difficili (cioè la versione come problema di riconoscimento è NP-completo) non rinunciamo a cercare soluzioni buone anche se non sempre ottime. Illustrare un possibile approccio per un problema a vostra scelta.

Come sopra, oppure

Consideriamo il problema di colorazione dei grafi: cerchiamo il minimo intero k tale che un grafo possa essere colorato con k colori (in modo che tutti i nodi adiacenti abbiano colori diversi).
Dato un grafo con n nodi $k \in [1, n]$. Possiamo effettuare una ricerca binaria partendo da k = n/2 e verificando se il grafo è k-colorabile. Qualora non lo sia proveremo con k = 3n/4, se invece lo è proviamo con k = n/4. Per ogni tentativo risolviamo il problema di decisione corrispondente. In questo modo troveremo la risposta entro O(logn) tentativi.

- Illustrare una riduzione – a vostra scelta - che mostra che un problema A è NP-completo.

Riduco polinomialmente da SAT ad A.

- Per mostrare che un problema B è NP-difficile è sufficiente fornire una riduzione da un problema A a B. Quali proprietà deve verificare A e quali deve verificare la riduzione?

Un problema è NP difficile se qualunque altro linguaggio in NP è polinomialmente riducibile al linguaggio. Affinché ciò sia vero è sufficiente che A sia NP-completo (ossia rientri tra i problemi “più difficili” tra gli NP).

- Descrivere le fasi di analisi lessicale e di analisi sintattica di un compilatore; per ciascuna delle due fasi descrivi l’input e l’output e come viene elaborato il programma.

L’analisi lessicale trasforma il testo (cioè una sequenza di caratteri) in una sequenza di token.
L’analisi sintattica trasforma questa sequenza di token in un albero di derivazione, verificando se la sequenza appartiene al linguaggio.

- Illustrare la gerarchia di Chomsky delle grammatiche; indicare per ciascuna categoria la proprietà/caratteristica a vostro giudizio maggiormente rilevante.

0: Grammatica non limitata, produzioni qualsiasi, modello di calcolo mdT
1: Grammatica contestuale, produzioni α->β con |β|>=|α|, modello di calcolo mdT lineare
2: Grammatica non contestuale, produzioni A->α, usata per specificare la sintassi
3: Grammatica regolare, produzioni A->aB o A->a, automa a stati finiti (regex)

- Descrivi l'algoritmo per eliminare le ricorsioni sinistre di una grammatica di tipo 2.

Ricorsioni dirette A->Aα
1) sostituiamo ogni produzione non ricorsiva A->δ con A'->δA'
2) sostituiamo ogni produzione ricorsiva A->Aα con A'->αA'
3) aggiungiamo A'->null

Ricorsioni non dirette A->Bα, B->Aβ
1) ordino i simboli non terminali A1, A2, … , Am
2) per ogni simbolo Aj
    a) considero i simboli precedenti Ah. Sostituisco ogni produzione Aj->Ahβ con l'insieme delle produzioni Aj->γβ per ogni produzione del tipo Ah->γ (facendo riferimento alle produzioni Ah già modificate)
    b) elimino le ricorsioni dirette

- Descrivere un analizzatore sintattico di tipo top-down.

L'analizzatore top-down genera derivazioni sinistre (al contrario di quello bottom-up, che invece genera derivazioni destre). A partire dall'assioma applica produzioni finchè sulle foglie non ottiene l'input.
Una soluzione ingenua è tentare tutte le possibilità, eventualmente effettuando backtracking (forza bruta). È però più efficiente utilizzare un parser predittivo (a tal fine la grammatica non deve avere ricorsioni sinistre), che osservando alcuni caratteri determina la mossa giusta o termina. In particolare se la grammatica è LL(1) è possibile realizzare un parser LL(1) che tramite il calcolo di first e follow crea una tabella dato il non terminale corrente e il terminale desiderato indica quale produzione applicare.

- Quali sono le proprietà di una grammatica LL(1) e per quale ragione sono la migliore scelta per descrivere i linguaggi di programmazione.

Una grammatica è LL(1) se, per ogni coppia di produzioni sullo stesso non terminale A → α e A → β, TABLE(A → α) ∩ TABLE (A → β) = Ø.
Table è la funzione definita come:
1) se λ appartiene a FIRST(α): FIRST(α) U FOLLOW(A)
2) else, FIRST(α)

Richiede di guardare un solo simbolo.

ESERCIZI
Grammatiche regolari, espressioni regolari e automi a stati finiti

- Si considerino le seguenti espressioni regolari sull'alfabeto {a,b,c}:

1) R1= a (b | aa)* c
2) R2= abb (a | cc) (ab)*

Per ciascuna costruire un automa a stati finiti deterministico o nondeterministico che riconosce il linguaggio definito dall'espressione regolare.

-> Quaderno

- Data una Grammatica regolare G, è possibile stabilire se G genera il linguaggio vuoto? Come?

Basta verificare se il corrispettivo automa va dallo stato iniziale a uno finale con una landa transazione.

- Data una Grammatica regolare G, è possibile stabilire se G genera un linguaggio finito o infinito? Come?

Basta verificare se ci sono cicli sul corrispettivo automa.

(Sugg. Per i due esercizi precedenti utilizzare il fatto che per ogni grammatica regolare G esiste un automa a stati finiti che riconosce tutte e sole le stringhe che appartengono a G)

- Alice e Biagio discutono animatamente. Stanno esaminando una grammatica regolare G in cui V_T è l'insieme dei terminali, V_N è l'insieme dei non terminali, S è l'assioma e P l'insieme delle produzioni (di tipo 3). Carla sta per fornire loro un ulteriore insieme di produzioni P', sempre di tipo 3, che tuttavia Alice e Biagio non hanno ancora visto. Carla ha avvisato che P è anch'esso basato su V_T e V_N e che chiederà ad Alice e Biagio di esaminare la grammatica regolare G' in cui V_T è l'insieme dei terminali, V_N è l'insieme dei non terminali, S è l'assioma e l'insieme delle produzioni (di tipo 3) è l'unione di P e P'. Siano L(G) e L(G') i linguaggi generati da G e G'. Alice sostiene che $L(G) \subseteq L(G')$ poiché l'introduzione di nuove produzioni potrebbe consentire di generare nuove stringhe, mentre Biagio pensa che $L(G) \supseteq L(G')$, dato che le nuove produzioni introdurranno restrizioni sulle possibilità di generazione.

$L(G) \subseteq L(G')$
Mat: Una produzione non introduce mai restrizioni, ma anzi le allarga sempre. Ha ragione Alice. Per esempio prendiamo la grammatica A-> aB B -> b|bB essa produce stringhe ab*. Se aggiungessimo la produzione B -> bA allora la grammatica genererebbe stringhe (ab*)* che sono molte di piú di quelle ab*. Quindi la tesi di Biagio é certamente falsa

- Per i seguenti linguaggi definisci un'espressione regolare o una grammatica regolare che li genera:

1) l'insieme delle stringhe binarie che contengono le stringhe 101 o 010 (o ambedue) come sottostringhe.

(0|1)*(101|010)(0|1)*

S->0A|1B A->1C|0A B->0D|1B C->0F|1B D->1F|0A F->0F|1F|0|1

2) l'insieme delle stringhe binarie che contengono le stringhe 00 e 11 come sottostringhe.

((0|1)*(00)(0|1)*(11)(0|1)*) | ((0|1)*(11)(0|1)*(00)(0|1)*)
Mat: (1|0)* (00(1|0)*11 | 11(1|0)*00) (1|0)* (penso sia piú intuitiva)

S->0A|1B A->0C|1B B->1D|0A C->1E|0C D->0G|1D E->1F|0C G->0F|1D F->0F|1F|0|1

3) l'insieme delle stringhe binarie che contengono la stringa 00 ma non la stringa 11 come sottostringhe.
S->1U|0Z U->0Z Z->1U|0T T->1D|0T|0|1|λ D->0T|0

(((01)|0)* | ((10)|0)*) (00) (((01)|0)* | ((10)|0)*)
mi sa che basta (10|0)*(00)(10|0)*

Mat: (0|10)(10)*0(0|10)*(0|1|lambda)
A -> 0Z | 1U
U -> 0Z
Z -> 0F | 1U
F -> 0F | 1V | lambda
V -> 0F | lambda
Io ho trovato questa soluzione, non riesco a capire se sia meglio o no

4) l'insieme delle stringhe binarie che contengono iniziano con 00 e finiscono con 11.

(00)(0|1)*(11)

S->0A A->0B B->1C|0B C->1D|0B D->1|1D|0B

5) l'insieme delle stringhe sull'alfabeto {x,y,z} in cui ogni x è immediatamente seguita da una y.

(z|y|xy)*

S->xX|yS|zS|λ X->yS

6) l'insieme delle stringhe sull'alfabeto {x,y,z} che contengono un numero dispari di y.

(x|z)*y((x|z)*y(x|z)*y(x|z)*)(x|z)*

S->xS|zS|yA A->xA|zA|yB|λ B->xB|zB|yA

- Sono dati M1 e M2, due automi a stati finiti che riconoscono i linguaggi L1 e L2 rispettivamente. Definire un automa M che riconosce il linguaggio L1 U L2 (formato dalle stringhe cha appartengono a L1 o a L2 o a entrambi).

-> QUADERNO

Grammatiche libere dal contesto

- Considera la seguente grammatica G (assioma S), con simboli terminali {a, b, c}

S -> aAb | AE
A -> aAbE | ab
B -> AB |c
D -> AAcE
E -> BA
F -> FA | a
1) Identifica i simboli non terminali che non sono raggiungibili a partire dall'assioma.
2) Trasforma la grammatica eliminando le produzioni inutili.

Risposta (1): D,F - non esiste la possibilità di generare una stringa che contiene D o F a partire dall'assioma S; trasformandolo in una sequenza di terminali; (2) bisogna eliminare tutte le produzioni che coinvolgono D e F

S -> aAb | AE
A -> aAbE | ab
B -> AB |c
E -> BA

- Considera la seguente grammatica G (assioma S), con simboli terminali {a, b, c}:

S -> aABb | AC
A -> aAb | ab
B -> abB
C -> ABC | c
1) Identifica i simboli non terminali che non sono utilizzati per definire stringhe del linguaggio.
2) Trasforma la grammatica eliminando le produzioni inutili.

Risposta (1): B non esiste la possibilità di eliminare B trasformandolo in una sequenza di terminali; (2) bisogna eliminare tutte le produzioni che coinvolgono B

S -> AC
A -> aAb | ab
C -> c

- **Costruire** un analizzatore sintattico predittivo a discesa ricorsiva per il linguaggio delle parentesi bilanciate, generato dalla grammatica S (assioma), simboli terminali '(' e ')' e produzioni S -> (S) S | λ. È possibile svolgere l'esercizio anche su una grammatica equivalente a quella data.

FIRST(S) = { (, λ } FOLLOW(S) = { $,) }

Parser:

	(	)	$
S	(S)S	λ	λ

- Data la grammatica per il linguaggio delle parentesi bilanciate, generato dalla grammatica S (assioma), simboli terminali ‘(‘ e ‘)’ e produzioni S -> (S) S | λ dare l’albero di derivazione per la stringa (()())().

-> QUADERNO

- Si consideri la grammatica

S -> aSbS S -> bSaS S -> λ
La grammatica è ambigua? Quanti differenti alberi di derivazione esistono per la sequenza abab? Mostrare le derivazioni sinistre e destre.

S -> aSbS -> a bSaS bS -> abaSbS -> ababS -> abab sinistra
S -> aSbS -> aSb aSbS -> aSbaSb -> aSbab -> abab destra
Pertanto è ambigua

- Sia data la grammatica - assioma E, simboli nonterminali caratteri maiuscoli, simboli terminali +, -, *, /, (,), id -

E -> TQ
Q -> +TQ
Q -> -TQ
Q -> λ
T -> FR
R -> *FR
R -> /FR
R -> λ
F -> (E)
F -> id
a) Fornire l’albero di derivazione per le stringhe (1) id + ((id* id)- id) * id (2) ((id-id) /id) + (id – (id *id))

-> QUADERNO

b) Questa grammatica è LL(1) ma la tabella di parsing non può essere calcolata usando solo i simboli FIRST; spiegare perché.

A causa delle landa-produzioni alcuni insiemi first contengono la stringa nulla. I simboli i cui first contengono stringa nulla potrebbero generare altri terminali (a seconda del simbolo da cui sono seguiti). Tali terminali non compaiono nei first, pertanto è necessario calcolarli tramite follow.

Dimostrare che la seguente grammatica
S -> aB
S -> aC
S -> B
B -> bB
B -> d

C -> B
non è LL(1).
Costruire una grammatica LL(1) equivalente alla precedente.

Soluz. La grammatica non è LL(1) a causa delle produzioni con S nella parte sinistra con stesso simbolo FIRST (S -> aB, S -> aC). Infatti non sappiamo scegliere quale delle due produzioni. Osserviamo che si può introdurre un nuovo non terminale T e modificare la grammatica nel seguente modo
S -> aT
T -> B
T -> C
S -> B
B -> bB
B -> d
C -> B
A questo punto è facile verificare che le produzioni T -> C e C -> B equivalgono a T -> B che è già presente; quindi possono essere eliminate. In questo modo otteniamo la grammatica
S -> aT
T -> B
S -> B
B -> bB
B -> d
Verifichiamo che questa grammatica è LL(1); calcoliamo i simboli FIRST
FIRST(S -> aT) = {a}
FIRST(S -> B) = {b,d}
FIRST(B -> bB) = {b}
FIRST(B -> d) = {d}
FIRST(T -> B) = {b, d}
Per produzioni con stessa parte sinistra i simboli FIRST sono disgiunti e quindi la grammatica è LL(1).

VECCHIO PDF

- Nell'analisi di complessità e, in particolare nella notazione O(), si ignorano le costanti moltiplicative e additive. Discutere questa scelta indicandone i vantaggi e gli svantaggi.

Con l'analisi della complessità si vuole valutare l'efficienza di un algoritmo, ad esempio a scopo di confronto tra algoritmi alternativi. Si sceglie di farlo in termini di valori asintotici poiché spesso si ha un nesso di questo tipo tra dimensione dell'input e tempo impiegato. A tal fine si assume che i valori costanti siano abbastanza piccoli da non modificare l'unità di grandezza del costo. Anche le costanti moltiplicative possono essere ignorate perché si ritiene che una moltiplicazione per una costante non abbia una grande influenza sul tempo di esecuzione rispetto all'andamento generale della stessa (soprattutto al crescere di un input potenzialmente infinito e dunque più grande di qualsiasi costante). In ogni caso occorre operare con una certa cautela.

- Cosa si intende quando diciamo che un algoritmo A è asintoticamente più efficiente dell'algoritmo B?

1. A è sempre meglio di B per tutti gli input
2. A è sempre meglio di B per input di grandi dimensioni <- QUESTA
3. A è sempre meglio di B per input di piccole dimensioni
4. B è sempre meglio di A per input di piccole dimension

i.

- Verificare e motivare la correttezza o meno delle seguenti affermazioni.

a) $(n!) \in \Omega (n^2)$
b) $n^{3/2} \in O(n\log n)$
c) $2^{(\log(\text{base}2)n)} \in \Theta(n)$

a) vera, perché la notazione omega comprende tutte le funzioni che crescono più velocemente e il fattoriale cresce più velocemente dell'esponenziale
b) falsa, $n^{3/2} = n^{2/2+½} = n*n^{½}$ falsa, perché la notazione ograndе comprende tutte le funzioni che crescono più lentamente
c) vera, perché la notazione teta comprende tutte le funzioni che crescono ugualmente, e per le proprietà dei logaritmi la funzione proposta è proprio n.

- Spiega la validità della seguente espressione e illustrare la differenza fra costo di un programma e andamento asintotico rappresentato con la notazione O().
  if f(n) <= g(n) then O(f(n) + g(n)) = O(g(n)).

L'espressione afferma correttamente che se f(n) ha una crescita asintotica minore rispetto a g(n), allora ai fini della notazione O() essa può essere trascurata. L'andamento asintotico viene utilizzato riportare il costo dei programmi in modo semplificato. In realtà il costo di un programma contiene addendi e costanti moltiplicative, ignorate nella notazione O(), e varia a seconda dell'input, mentre la notazione O() viene usata per includere vari input ed esprimere il costo di caso peggiore tra quelli.

- Per n > 1, sia T(n) il numero esatto di esecuzioni del comando a = a + 2 nel seguente frammento codice:

```
for i = 1 to n − 1 do
        for j = n − i + 1 to n do
                a = a + 2
```

Utilizzando la notazione O() si esprima il costo di T(n) in funzione di n; si calcoli il valore esatto di T(n) per n=10.

Il ciclo esterno verrà ripetuto n-1 volte.
Per il ciclo interno, assumiamo n = 10:

Iterazione esterna	n-i+1	#iterazioni interne
1	10	0
2	9	1

3	8	2
…		
k	10-k+1	k-1

Il numero complessivo di iterazioni interne è 1+2+...+(n-1) = (n-1)(n-2)/2 = $O(n^2)$ = T(n).
Nello specifico per n = 10, T(n) = 1+2+3+4+5+6+7+8+9

- Ripetere l'esercizio per il programma seguente

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
        for (j = 2; j <= n; j = j * 2) {
                k = k + n / 2;
        }
}
```

Il ciclo esterno viene eseguito n/2 volte (con i che va da n/2 a n).
Per il ciclo interno, assumiamo n = 10:

Iterazione esterna	j	#iterazioni interne
5	2, 4, 8	3
6	2, 4, 8	3
7	2, 4, 8	3
…		
k	2, 4, 8	3

Il numero di iterazioni interne è sempre pari a parteInferiore($\log_2(n)$).
Quindi il numero complessivo di operazioni interne è nlog(n)/2 che è O(nlogn).
Nello specifico T(n) = (10/2)*$\log_2 10$ = 5*3 = 15.

- **Esprimere il costo** asintotico dell'algoritmo usuale per la moltiplicazione di due numeri di n bit. Assumi che ciascuna operazione fra due bit abbia costo unitario.

```
1.....101 x     n bit
1.....11        n bit

_________
1 .....101 +    n       operazioni
..... 1010 +    n+1 operazioni          {Somma di n valori parziali
.....00 +
…

_________________
…….....00.....0     2n bit nel caso peggiore.
```

Eseguirò prima dei prodotti e poi delle somme. I prodotti richiedono che ogni cifra del secondo fattore sia moltiplicata per ogni cifra del primo, per un totale di n^2 operazioni elementari. Le colonne da sommare saranno al massimo 2n e su ogni colonna ci saranno n elementi. Quindi la prima e la seconda colonna richiedono al più n-1 somme, mentre le altre (che possono aver riporto) richiedono al più n somme. Dunque il numero totale di somme è $O(2n^2) = O(n^2)$. Il costo asintotico complessivo è dunque $O(n^2)$.

- Quante operazioni sono necessarie con l'usuale algoritmo per eseguire la moltiplicazione di una matrice (n x m) per una matrice (m x k)?

La matrice complessiva avrà n x k elementi, ognuno di essi sarà stato calcolato tramite la somma di m prodotti, dunque il costo complessivo è n x k x m.

- Sia dato un array ordinato A che memorizza solo 0 e 1, tale che ogni zero appare in una componente prima di componenti che memorizzano 1; in altre parole l'array è del tipo {0, 0, 0, ..., 0, 0, 1, 1, ..., 1,1,1}. Progetta un algoritmo che trova la componente più piccola tale che A[i] = 1 e abbia costo O(log n) nel caso peggiore (n rappresenta il numero degli elementi dell'array).

Algoritmo:

```
pos = i/2
length = A.length
while(true){
        if A[pos] == 1 && A[pos-1] == 0
                return possdfv
        length = length/2
        else if A[pos] == 0
                pos = pos + length/2
        else if A[pos] == 1
                pos = pos - length/2
}
```

- Enunciare e discutere la tesi di Church-Turing, chiarendo perché viene formulata come tesi e non come teorema.

Fatto sopra

- Definire il concetto di numerabilità (o equivalentemente contabilità) e mostrare utilizzando la tecnica della diagonalizzazione che l'insieme dei linguaggi sull'alfabeto {0, 1} non è numerabile.

Un insieme si dice numerabile quando ha la stessa cardinalità dei naturali (cioè lo stesso grado di infinito).

Per assurdo immaginiamo che l'insieme dei linguaggi sull'alfabeto {0, 1} sia numerabile, dunque deve esistere un ordine tra essi. Costruiamo dunque una tabella in cui a ogni colonna corrisponda un linguaggio e ad ogni riga una stringa di zeri e uni (a loro volta

numerabili perché sono in corrispondenza biunivoca con i numeri interi). In ogni cella scriviamo SI se la stringa appartiene al linguaggio e NO se non appartiene. Ora definiamo un nuovo linguaggio tale che data la stringa i-esima esso la accetta se l'iesimo linguaggio non la accetta e la rifiuta se l'iesimo linguaggio la accetta. Tale linguaggio sarà diverso da tutti quelli ordinati, e sarà stato escluso dall'ordinamento contraddicendo così la tesi.

- Assumi che i linguaggi B e C siano decidibili (cioè dato x esistono due algoritmi A1 e A2 che decidono se x appartiene a B e se x appartiene a C rispettivamente). Mostra che anche

B U C (linguaggio unione) , B ∩ C (linguaggio intersezione),
B \ C (linguaggio differenza formato dalle stringhe che appartengono a B ma non a C)
B*
sono linguaggi decidibili.

Un linguaggio è decidibile **se e solo se** esiste una MdT che lo decide, pertanto avrò una MdT per B e una per C. A partire da esse posso definire una nuova macchina:
quella per l'unione prova prima MB e accetta se MB accetta, altrimenti prova MCe accetta se MC accetta, altrimenti rifiuta;
quella per l'intersezione prova prima MB e rifiuta se MB rifiuta, altrimenti prova MC e accetta se MC accetta, altrimenti rifiuta;
quella per l'intersezione prova prima MB e rifiuta se MB rifiuta, altrimenti prova MC e accetta se MB rifiuta, rifiuta altrimenti;
quella per la **chiusura** prova prima MB e quando MB rifiuterebbe resetta a zero lo stato e prova a proseguire la scansione, accetta se arriva a fine stringa in uno stato finale (è necessario che quest'ultima macchina, se deterministica, implementi un algoritmo che permetta di effettuare backtracking in modo da analizzare la stringa di input in tutti i modi possibili).

Un algoritmo che decide un linguaggio significa che é capace di riconoscere se una certa stringa L appartiene o meno al linguaggio. Se vediamo questo algoritmo come una funzione booleana A(stringa s) che ritorna true se la stringa s appartiene
Il linguaggio Unione é deciso da una concatenazione degli algoritmi A1 e A2 (A1(s) || A2(s))
Il linguaggio intersezione é deciso da una concatenazione di A1 e A2 (cioé A1(s) && A2(s))
Il linguaggio differenza é deciso da A1(s) && !A2(s)
Il linguaggio chiusura di Klein é deciso da:

```
int j=0;
for (int i=0; i<s.length; i++) {
        if (A1(s[j, i])) {
                j = i;
        }
}
return j==s.length;
```

- Dati A1 e A2 abbiamo che A1(x) = vero se x appartiene a B falso altrimenti. Analogamente per A2 e C.

N ke senso?? Non penso sia una domanda, ma una mezza risposta a quello sopra?

- Dimostra che il seguente insieme è decidibile {B: B è la codifica di un automa a stati finiti deterministico che non accetta nessuna stringa}

Un insieme è decidibile se esiste un programma che lo decide. In questo caso è sufficiente implementare un programma che verifichi l'esistenza di un cammino dallo stato finale a quello iniziale. A tal fine l'automa può essere rappresentato come grafo diretto e si può applicare la DFS.

Soluzione: La prova per decidere se X appartiene al linguaggio si basa sui seguenti passi: utilizzando X si costruisce l'automa deterministico codificato. Quindi si verifica per ogni stato finale q se esiste un cammino dallo stato iniziale a q. Se un tale cammino esiste per almeno uno stato finale allora l'automa accetta stringhe. Per verificare che per ogni stato finale q esiste un cammino dallo stato iniziale a q è sufficiente verificare l'esistenza di cammini di lunghezza al più $|Q| - 1$, dove Q è il numero di stati dell'automa.

- Perché è sufficiente considerare cammini di lunghezza $|Q| - 1$?

Perché sarebbe inutile tornare più volte in uno stato che mi offre le stesse produzioni (e perché escludo lo stato finale).

- Mostra che l'insieme delle Macchine di Turing è numerabile.

Ogni macchina di Turing può essere tradotta in un programma in linguaggio macchina, ossia in una sequenza di zeri e uni che può essere tradotta in un intero naturale. Dunque è possibile stabilire un ordine tra le varie MdT.

- Quante sono le possibili macchine di Turing con alfabeto binario e con 4 stati?

(Assumo si parli di MdT deterministiche) Ogni stato avrá solo 2 bracci uscenti, uno per 1 e uno per 0, ognuno di questi bracci puó andare in uno dei 4 stati (compreso quello di partenza) o potrebbe anche non esistere affatto, quindi per ogni braccio ci sono 5 possibilitá. questo significa che per ogni stato ci sono 25 possibili combinazioni visto che i bracci sono 2. ognuno degli stati ha 25 possibili forme, e gli stati sono distinguibili tra loro, quindi il totale delle possibili combinazioni é 25^4 => 5^8

- Descrivere le fasi di analisi lessicale e di analisi sintattica di un compilatore; per ciascuna delle due fasi descrivi l'input e l'output e come viene elaborato il programma.

Vedi sopra.

- Illustrare la gerarchia di Chomsky delle grammatiche; indicare per ciascuna categoria la proprietà/caratteristica a vostro giudizio maggiormente rilevante.

Vedi sopra.

- Descrivi l'algoritmo per eliminare le ricorsioni sinistre di una grammatica di tipo 2.

Vedi sopra.

- **Fornire una grammatica** non ambigua per generare il linguaggio delle espressioni aritmetiche avente cinque simboli terminali: id (che rappresenta un identificatore), i simboli di operazione + e – e le parentesi tonde (e). Ovviamente la grammatica deve generare tutte e sole le stringhe che sono corrette (ad esempio non deve generare la stringa id+((id-id) o la stringa id++id-id.

E -> TF
T -> F+ | F-
F -> id | (E)

E -> TQ (Assioma)
T -> id | (E)
Q -> +TQ | -TQ | lambda

- Motivare inoltre perché questo linguaggio non può essere generato da una grammatica di tipo 3.

Per le parentesi, infatti i linguaggi di tipo 3 non riescono a tenere "memoria" di quante parentesi abbiano messo prima, e quindi non sanno quante ne devono mettere dopo. tutti i linguaggi tipo { a^n b^n | n>0 } non possono essere generati da grammatiche di tipo 3 perché richiedono di "contare" quante parentesi aperte vengono prima per capire qunate ne andranno messe dopo.

- Quali sono le proprietà di una grammatica LL(1) e **per quale ragione sono molto adatte** per descrivere i linguaggi di programmazione?

Il termine LL(1) indica le grammatiche che possono essere analizzate con un parsing top-down LL(1) ossia da un parser che analizza l'input da sinistra (Left) verso destra e costruisce una derivazione sinistra (Leftmost), leggendo un solo carattere alla volta per scegliere quale produzione applicare. Esse sono utili perché tramite una tabella costruita analizzando gli insiemi first e follow della grammatica permettono di tokenizzare una stringa senza effettuare backtracking, ossia in modo predittivo, e pertanto risultano particolarmente efficienti.
Il loro tratto distintivo è che gli insiemi first di due produzioni che presentano la stessa stringa nella parte sinistra sono sempre disgiunti.

- Quali sono le proprietà di una grammatica LR(0) e per quale ragione sono molto adatte per descrivere i linguaggi di programmazione?

Il termine LR(0) indica le grammatiche che possono essere analizzate da un parser bottom-up LR(0), ossia da un parser che analizza l'input da sinistra verso destra (Left), e costruisce una derivazione destra (Leftmost), senza leggere alcun carattere di input. Affinché una grammatica sia LR(0) per ciascuno degli stati a) può esistere una sola produzione con punto finale (assenza di conflitti reduce-reduce) e b) non possono contemporaneamente esistere una produzione con il punto finale e una con il punto non finale (assenza di conflitti shift-reduce). Simili grammatiche sono utili perché tramite le tabelle action e goto permettono di

tokenizzare una stringa senza effettuare backtracking, ossia in modo predittivo, e pertanto risultano particolarmente efficienti.

- Descrivere un analizzatore sintattico di tipo top-down specificando l'input dell'analizzatore, i possibili risultati calcolati e un possibile algoritmo per l'analisi.

input: stringa di caratteri (terminali)
risultati: albero di derivazione (se l'input può essere tokenizzato), errore altrimenti
algoritmo:
parto dall'assioma S
creo una lista di oggetti produzione L
for ogni carattere in input
    leggo il carattere
    se non posso applicare nessuna produzione restituisco errore
    confrontando la tabella LL(1) scelgo quale produzione applicare (vedi a fine file come costruire la tabella)
    aggiungo la produzione a L
    se sono a fine input e ho analizzato tutto S, accetto

- Descrivere l'algoritmo di analisi sintattica di una grammatica LL(0).

Come domanda precedente(?)

- Descrivi l'algoritmo di analisi sintattica di una grammatica LR(0).

Algoritmo:
Inizializzo una pila P con s0
while(true)
    leggo il primo carattere di input
    tramite la tavola action scelgo quale operazione eseguire
    if shift
        inserisco lo stato indicato e scorro l'input
    if reduce
        rimuovo da P tanti stati quanti i caratteri a destra della produzione
        tramite la tabella goto leggendo lo stato affiorante e il carattere a sinistra della produzione individuo lo stato da inserire
    if accetta
        termina accettando
    else
        termina in errore

(vedi a fine file come costruire le tabelle action e goto)

- Descrivi il linguaggio generato dalla seguente grammatica (S assioma, S e B simboli non terminali) S → abc, S → aSBc, cB → Bc, bB → bb. Nella gerarchia di Chomsky di che tipo è la grammatica precedente? Prova a trovare una grammatica di tipo 3 che genera lo stesso linguaggio.

Vediamo alcune stringhe generabili:
S -> abc
S -> aSBc -> aabcBc ->aabBcc -> aabbcc
S ->aSBc -> aaSBcBc -> aaabcBcBc -> aaabBccBc -> aaabbccBc -> aaabbcBcc -> aaabbBccc -> aaabbbccc

La grammatica genera L = {$a^n b^n c^n$: n>0}

Si tratta di una grammatica di tipo 1, ossia contestuale, in cui si hanno produzioni del tipo α->β, con |β| >= |α|

Non è possibile trovare una grammatica di tipo 3 **perché** le grammatiche di tipo 3 non sono in grado di contare.

Di seguito ometto il testo di altri esercizi già visti in precedenza.

- Definire un ASF che, avendo in input stringhe costruite sull'alfabeto 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, riconosce il linguaggio delle stringhe che descrivono un numero multiplo di 5 in base 10. (È ammissibile che un numero inizi per 0, come ad esempio 0051 o 012345)

-> QUADERNO

Trova una grammatica libera dal contesto per i seguenti linguaggi
!

L1: S -> A|B A -> 0A1|0A|0 B -> 0B1|B1|1
L2:
S -> aSd|ad|A|D|T
A -> bAd|bd|T
D -> aDc|ac|T
T -> bTc|bc
L: S->0S1|01

GIUGNO 2021 A

1) Con riferimento al linguaggio L = {x∈(0|1)* | 1 è sempre seguito da 010}
   a) scrivere un opportuno automa che riconosce tutte e sole le stringhe che appartengono al linguaggio
   b) scrivere una grammatica del tipo più alto possibile che genera il linguaggio L
   c) determinare un'espressione regolare che lo rappresenta

a) -> QUADERNO
b) tipo 3: S->0S|ε|1A A->0B B->1C C->0S
c) (0|1010)*

2) Definire la notazione Ω. Spiegare il significato della seguente affermazione: "Il problema P ha complessità Ω(nlogn)"

La notazione Ω viene usata per definire il costo asintotico di una funzione (ad esempio un algoritmo). In particolare, data una funzione definita su N, definiamo Ω(g(n)) come l'insieme delle funzioni che per valori maggiore di un certo n0>0 assumono un valore almeno pari a c*g(n), con c>0.
Intuitivamente se f(n)∈Ω(g(n)) allora il valore di f cresce almeno come quello di g.
Un problema risolubile tramite algoritmi ha complessità Ω(nlogn) se il numero di operazioni elementari eseguite da un qualsiasi algoritmo che lo risolve è almeno pari a nlogn moltiplicato per una qualche costante c. n indica la dimensione in byte dell'input.

3)

a) Dare la definizione di MdT
b) Assumi che una MdT debba calcolare una funzione dei dati presentati all'inizio del nastro. Cosa fa la MdT descritta in figura? C'è una configurazione finale? In caso, qual è? (nella figura il quadratino rappresenta lo spazio blank, R destra, S stop)

a) Una MdT è un modello di calcolo che prevede un alfabeto Σ contenente il simbolo _ , un insieme di stati Q di cui uno iniziale q0 e almeno uno finale q1 e una funzione di transizione che dato uno stato e un carattere di Σ (quello letto) restituisca una coppia nella forma <c, a, s> con c carattere da scrivere (eventualmente), a azione da svolgere (tra spostarsi a sinistra L, spostarsi a destra R e non spostarsi S) e s nuovo stato corrente.

È anche possibile che la macchina sia non deterministica e che dunque la funzione di transizione indichi al posto di un nuovo stato un insieme di nuovi stati.

Schematicamente la MdT è composta di un nastro infinito e di una testina che permette di leggere un carattere alla volta ed eventualmente sovrascriverlo. Sul nastro (o su altri nastri, letti da altrettante testine) devono essere presenti una codifica delle transizioni e una memoria dello stato corrente.

b)
Stato q0, con 0 va avanti, con 1 va in q1 e va avanti, con blank va in q4 e va avanti
Stato q1, con 0 va avanti, con 1 torna in q0 e va avanti, con blank va in q2
Stato q2, con blank sovrascrive 0 va in finale e torna indietro
Stato q4, con blank scrive 1 e va indietro

Stato	Stringa	Dopo spostamento
0	**1**000	1**0**00
1	1**0**00	10**0**0
1	10**0**0	100**0**
1	100**0**	1000**_**
1	1000**_**	1000**__**

2	1000_**_**	1000**_**0

Stato	Stringa	Dopo spostamento
0	**1**010	1**0**10
1	1**0**10	10**1**0
1	10**1**0	101**0**
0	101**0**	1010_**_**
4	1010**__**	1010**_**1

L'automa accetta stringhe sull'alfabeto {0, 1, _} , restituendo 1 se la stringa contiene un numero pari di 1 e 0 altrimenti.

4) S -> SaS | SbS | c

a) Fornire la sequenza di derivazioni della stringa cacbc
b) **Definire** (anche ricorsivamente) il linguaggio generato dalla grammatica
c) Stabilire se la grammatica è ambigua. Se sì, **quali interventi** si possono fare per eliminare questo problema

a) S -> SaS -> caS -> caSbS -> cacbS -> cacbc
b) Linguaggio L sull'alfabeto {a, b, c} tale che x∈L se x corrisponde a "c" o se x inizia con una c seguita da una qualsiasi sequenza di ac e/o ab (almeno un'occorenza) e termina con c.
L = {x = c(ac|ab)*c}
c) Sì, ad esempio della stringa iniziale è possibile scrivere un'altra derivazione:
S -> SbS -> SaSbS -> caSbS -> cacbS -> cacbc
È possibile riscrivere la grammatica: S -> c | caS | cbS

5)

a) Determinare l'insieme dei FIRST e dei FOLLOWER della seguente grammatica:
   S -> aSTU | ST | ε
   T -> baT | abU | ε
   U -> babS | c
b) La grammatica risulta LL(1)?

a)
FIRST(S) = {a, ε} + FIRST(T) = {b, a, ε}
FIRST(T) = {b, a, ε}
FIRST(U) = {b, c}
FIRST(aSTU) = {a}
FIRST(ST) = FIRST(S)-ε + FIRST(T) = {b, a, ε}
FIRST(ε) = {ε}

FIRST(baT) = {b}
FIRST(abU) = {a}
FIRST(babS) = {b}
FIRST(c) = {c}
FOLLOW(S) = $ + FIRST(T) + FIRST(U) + FOLLOW(U) = {$, b, a, c}
FOLLOW(T) = FIRST(U) +FOLLOW(S) = {$, b, a, c}
FOLLOW(U) = FOLLOW(S) + FOLLOW(T) = {$, b, a, c}

b) No, gli insiemi first non sono disgiunti, ad esempio avendo s e leggendo a non posso scegliere tra S -> aSTU (FIRST = {a}) e S -> ST (FIRST = {b, a, ε}).

6)

a) Se qualcuno riuscisse a provare che NP ⊇ P (strettamente), cosa possiamo dire sui problemi della classe NP\P?
b) Sapendo che 3CNF è NP-completo come posso utilizzare questo fatto per dimostrare che un altro problema è NP-completo? **Fornire un esempio** a vostra scelta.

a) Che sicuramente tali problemi non sono decidibili in un tempo polinomiale
b) Effettuando una riduzione polinomiale da 3CNF all'altro problema.

Ad esempio vogliamo dimostrare che 4CNF è NP-completo. iCNF indica una formula booleana in cui ogni clausola ha al massimo i letterali. Data una formula F di tipo 4CNF, voglio definire una formula F' di tipo 3CNF che è soddisfacibile se e solo se F è soddisfacibile.

F sarà nella forma
(a v b v c v d) ∧ (e v f v g v h) ∧ … (i caratteri indicano un letterale positivo o negativo)
e sarà vera solo se almeno un tra a,b,c,d e almeno uno tra e,f,g,h saranno verificati. Allora F' = ((a v b) v (a v c) v (a v d)) ∧ ((e v f) v (e v g) v (e v h)) sarà certamente verificata.
Al contrario F sarà falsa se tutti a,b,c,d,e,f,g,h,i,l,m,n sono falsi e in tal caso anche F' sarà falsa.
Le relazioni valgono anche in senso opposto, cioè se F' è vera anche F sarà vera, mentre se F' è falsa anche F sarà falsa.
In generale F' si costruisce come 3CNF con tante clausole quante F. L'iesima clausola di i si costruisce a partire dall'iesima clausola di f come mostrato.

7)

a) Illustrare la tesi di Church-Turing
b) Spiegare **se l'equivalenza** stabilita nella tesi **si applica anche agli automi** a stati finiti

a) Tutto ciò che possiamo calcolare lo possiamo calcolare con una MdT.
b) No, gli automi a stati finiti sono un sistema di Calcolo equivalente a una grammatica di tipo 3, adatto ad esprimere un numero di linguaggi ridotto rispetto a quelli accettati da una MdT.

GENNAIO 2020

1) Con riferimento al linguaggio R descritto dall'espressione regolare bb((aa)*+b)*a
   a) definire un ASF (deterministico o non) che riconosce il linguaggio R
   b) definire una grammatica regolare che genera R

a) -> QUADERNO

b) S -> bA A -> bB B -> bB|aF F -> aB

2) Si consideri la grammatica G: S -> T$, T -> (T)T | ε, con terminali {$, (,)}, non terminali {S, T} e assioma S, che genera il linguaggio L
   a) Stabilire se G è LL(1) oppure no; nel caso non lo sia, modificare G in una grammatica G' che sia LL(1) ed equivalente a G
   b) Descrivere un algoritmo che effettua il parsing predittivo delle stringhe di L

a)
FIRST(S) = FIRST(T) = {(, ε}
FIRST((T)T) = {(}
FIRST(ε) = {ε}
FOLLOW(S) = {$}
FOLLOW(T) = FOLLOW(S) + {)} = {$,)}

È LL(1)

b)

	(	)	$
S	S->T		S->T
T	T->(T)T	T->ε	T->ε

Per la descrizione dell'algoritmo di parser vedi esercizi precedenti.

3) Descrivere la tecnica di progettazione algoritmica denominata "programmazione dinamica", chiarendo in cosa si distingue dal divide et impera, e illustrare un algoritmo (a piacere) basato sulla programmazione dinamica.

Nella programmazione dinamica la soluzioni viene costruita (bottom-up) a partire da un insieme di sotto-problemi potenzialmente ripetuti. I sottoproblemi in cui esso si può scomporre non sono indipendenti.
Al contrario nella progammazione di tipo divide et impera il problema viene suddiviso in sotto-problemi indipendeti che vengono risolti ricorsivamente (top-down).

4) Illustrare l'algoritmo DPLL, spiegando quale problema risolve. Determinarne il **costo computazionale** e mostrare come l'algoritmo, durante i suoi passi, modifica la formula in input

(a v b v ¬c) ∧ (¬b v d) ∧ (¬a v c v d v ¬e) ∧ (¬a v b v ¬d v e) ∧ (c v d v ¬e) ∧ (c v ¬d v ¬e) ∧ (a v c v d v e) ∧ (a v c v d v ¬e)

L'algoritmo DPLL è un algoritmo che verifica la soddisfacibilità di una formula booleana. È basato sul backtracking, ossia tenta tutte le assegnazioni possibili. Per migliorare l'efficienza vengono usate due ottimizzazioni:
unit propagation: se una clausola è unitaria il letterale contenuto va resa vera in tutte le sue occorrenze
pure literal assign: se un letterale appare sempre positivo o sempre negativo viene chiamato puro. I letterali puri possono essere resi tutti veri senza effettuare scelte.

```
function DPLL(F)
    if F is empty
        then return true;
    if F contains an empty clause
        then return false;
    for every unit clause l in F
        F = unit-propagate(l, F);
    for every literal l that occurs pure in F
        F = pure-literal-assign(l, F);
    l := choose-literal(F);
    return DPLL(F Union {l}) OR DPLL(F Union {NOT l});
```

Il caso peggiore si ha quando la formula non è soddisfacibile e non si possono applicare semplificazioni, in tal caso verranno provate tutte le assegnazioni possibili. Sia n il numero di letterali usati nella formula, ognuno dei quali può assumere due valori.
Se n = 1, posso avere le assegnazioni [0] o [1] -> 2 possibilità
Se n = 2, posso avere [0,0], [1,1], [0,1], [1, 0] -> 2^2
Se n = 3, posso avere [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 0], [1, 1, 1] -> 2^3
In generale 2^n. Per ogni assegnazione devo sostituire tutti i letterali con il loro valore, costo O(n), e infine verificare, che ha costo costante. Nel complesso il costo è **$O(n2^n)$**. Verificare che non siano possibili semplificazioni richiede di scorrere tutta la formula, in generale dipenderà dall'implementazione, se la formula ha pochi letterari ma molte clausole potrebbe essere più oneroso del backtracking in se.

Applico l'algortimo:
(a v b v ¬c) ∧ (¬b v d) ∧ (¬a v c v d v ¬e) ∧ (¬a v b v ¬d v e) ∧ (c v d v ¬e) ∧ (c v ¬d v ¬e) ∧ (a v c v d v e) ∧ (a v c v d v ¬e)

1. scelgo a=1
(a v b v ¬c) ∧ (¬b v d) ∧ (¬a v c v d v ¬e) ∧ (¬a v b v ¬d v e) ∧ (c v d v ¬e) ∧ (c v ¬d v ¬e) ∧ (a v c v d v e) ∧ (a v c v d v ¬e) ∧ (a)

2. unit propagation di a
(¬b v d) ∧ (c v d v ¬e) ∧ (b v ¬d v e) ∧ (c v d v ¬e) ∧ (c v ¬d v ¬e) ∧ (a v c v d v e)
pure literal assign di c = 1

(¬b v d) ∧ (b v ¬d v e)
scelgo b = 0
(¬d v e)

3. pure literal assign di d = 0
(e)
pure literal assign di e = 1
_ -> soddisfacibile [a=1, b=0, c=1, d=0, e=1]

5) Con riferimento alle classi P ed NP:

a) Definire la classe P
b) Definire la classe NP attraverso macchine di Turing non deterministiche
c) Definire la classe NP senza impiegare il concetto di non determinismo
d) Confrontare le due definizioni di NP, giustificandone l'equivalenza
e) Se per un problema della classe NP viene individuato un lower bound polinomiale, cosa possiamo dedurre in merito alla questione P vs NP?
f) Se per un problema della classe NP viene individuato un lower bound esponenziale, cosa possiamo dedurre in merito alla questione P vs NP?

a) P è la classe dei linguaggi decidibili in un tempo polinomiale $O(n^k)$, n>=0.
b) NP è la classe dei linguaggi risolubili da MdT non deterministiche in tempo polinomiale.
c) NP è la classe dei linguaggi per cui, se x appartiene a L, allora esiste un certificato c(x) tale che: c(x) ha lunghezza polinomiale in |x| ed esiste una MdT deterministica che con input x e c(x) verifica che x appartiene a L in tempo polinomiale a |x| e |c(x)|.
d) La prima definizione è diretta, la seconda evidenzia che dato un valore è facile (polinomiale) verificare se il valore è accettato dal linguaggio. Il difficile sta nel trovare un valore accettato. Tuttavia le due definizioni sono equivalenti in quanto possiamo vedere la codifica della mdt non deterministica come un certificato.
e) Che esiste almeno un problema P che appartiene anche ad NP (e così è)
f) Che NP != P, infatti il problema con lower bound esponenziale non apparterrebbe a P.

MAGGIO 2020

1) Per n > 0, si consideri una ricorrenza T(n) il cui albero delle chiamate è binario completo (cioè, ogni nodo interno ha due figli e ciascun ramo ha la stessa lunghezza) ed n è il numero di foglie. Il lavoro compiuto in ciascun nodo dell'albero è pari a 1.

a) Esprimere T(n) in forma chiusa attraverso la notazione Θ(n).
b) Si determini il valore esatto di T(8) (albero con 8 foglie).

a) $T(n) \in \Theta(2^n)$

b) I nodi sono 4+2+1, in ognuno si compie lavoro 1, quindi T(8) = 7

2) Definire le classi P e NP e indicare le relazioni di contenimento fra le classi. Illustrare la rilevanza pratica della questione $P \stackrel{?}{=} NP$.

Definizioni sopra. La questione è di rilievo perché possiamo risolvere in modo relativamente efficiente i problemi P, al contrario di quelli NP. Tuttavia poiché non si è ancora riuscito a

dimostrare che esistano problemi NP non risolubili in tempo P (ne che tutti i problemi P siano anche NP), possiamo solo congetturare NP⊃P.

3) Con riferimento al problema SAT in cui, data una formula booleana con variabili logiche che possono essere affermate o negate occorre stabilire se la formula sia soddisfacibile o meno, si mostri che esso appartiene alla classe NP.

SAT appartiene a NP: posso costruire un automa non deterministico che mettendo a vero e a falso tutte le variabili, in tempo polinomiale arriva a uno stato finale se e solo se esiste un'assegnazione di variabili che verifica la formula.
Perché esso sia NP-completo occorre dimostrare anche che ogni problema in NP è riconducibile a SAT.
Dato un linguaggio in NP esiste un algoritmo di complessità polinomiale A (ad esempio una MdT) e un polinomio p tali che per ogni stringa x, se essa appartiene a L, esiste una stringa y (il certificato) di lunghezza non superiore a p(|x|) tale che V con x e y in ingresso decide il linguaggio.
Per ogni x è possibile costruire, in tempo polinomiale, una formula booleana F(x,y) la cui soddisfacibilità dipende solo dalle p(|x|) variabili $y_0, y_1, \ldots, y_{p(|x|)-1}$. Tale F è la funzione richiesta dalla riduzione.

4) Nell'analisi di complessità e, in particolare nella notazione O(), si ignorano le costanti moltiplicative e additive. Discutere questa scelta indicandone vantaggi e svantaggi.

C'è una domanda simile sopra. Bisogna considerare che costanti molto grandi possono alterare l'efficienza dell'algoritmo entro una certa dimensione dell'input.

5) Si consideri il seguente programma incompleto per risolvere della più lunga sottosequenza comune di due con la tecnica della programmazione dinamica. Completare il programma nei punti indicati con "....."

Input: Stringhe X e Y con n e m elementi, rispettivamente
Output: Matrice L, tale che, per i = 0,…,n-1 e j = 0,...,m-1, L[i,j] memorizza la lunghezza della stringa più lunga che è sottosequenza sia della stringa X[0..i] = x0x1x2…xi che della stringa Y [0..j] = y0y1y2…yj

```
for i =1 to n-1 do
        L[i,-1] = 0
for j =0 to m-1 do
        L[-1,j] = 0
for i =0 to n-1 do
        for j =0 to m-1 do
                if xi = yj then
                        L[i, j] = …… (1)
                else
                        L[i, j] = …… (2)
return array L
```

(1) L[i-1, j-1] + 1;

(2) max{L[i-1, j], L[i, j-1]}

Confrontare il compito di luglio 2019.

6) Costruire un automa a stati finiti (deterministico o non-deterministico) che riconosce le stringhe appartenenti al linguaggio costituito da tutte e sole le stringhe binarie contenenti almeno due simboli 1 adiacenti e almeno un simbolo 0 (in qualunque posizione, cioè possiamo avere prima la stringa 11 e poi uno 0, o viceversa).

-> QUADERNO

7) Sia data la grammatica con insieme dei simboli non terminali N = {S, T}, insieme simboli terminali T = {a, b, c}, assioma S e produzioni specificate nel seguito:

S → aSc | aTc | ac | bc

T →bTc | bc

a) Fornire un albero di derivazione della stringa aabbbccccc;
b) Di quale tipo è tale grammatica nella gerarchia di Chomsky?
c) Specificare (descrivere con notazione insiemistica) il linguaggio generato dalla grammatica.

a) -> QUADERNO

b) Tipo 2, cioè non contestuale, produzioni A->α

c) Si tratta del linguaggio sull'alfabeto {a, b, c} tale che L = $\{a^n b^m c^q: n+m = q\}$

8) Definire il concetto di grammatica ambigua e discutere se la grammatica fornita nell'esercizio precedente è ambigua o no.

Una grammatica è ambigua quando almeno una stringa ammette più di un albero di derivazione. La precedente grammatica non è ambigua perché ad ogni passo si può espandere un solo terminale e la scelta è obbligata fal non terminale in lettura.

9) Si costruisca una grammatica LL(1) per il linguaggio $\{a^n, n \geq 1\}$

S -> aT
T -> aT|ε

FEBBRAIO 2020

1) Illustrare la notazione O(.) – O grande - e fornire la definizione di problema computazionalmente "facile".

La notazione O() viene usata per esprimere il costo asintotico degli algoritmi. In particolare data una funzione su N f(n), si dice che $f(n) \in O(g(n))$ f(n) <= a*g(n) + b, con a>0, b>=0. f(n) rappresenta il numero di istruzioni eseguite dall'algoritmo rispetto a n dimensione dell'input. Un problema è facile se il suo andamento asintotico cresce lentamente. Solitamente con facile si intende polinomiale, ossia risolubile da una MdT deterministica in un tempo $O(n^k)$, con k >=1.

2. Definire il problema decisionale Vertex Cover e mostrare che esso appartiene alla classe NP. (Suggerimento: fornire un algoritmo non deterministico che "indovina" l'insieme di nodi da cercare).

Dato un grafo G e un intero k, il problema vertex cover si chiede se esista un insieme di k nodi che coprono tutti gli archi.
Un algoritmo non deterministico che lo risolva sceglie un insieme di nodi e verifica se tale insieme copre tutti gli archi ed ha dimensione pari a k.

```
Sia A un array i nodi di dimensione k
for(int i = 0; i<k; i++)
        a[i] = choice (G.nodi)          //sceglie alcuni dei nodi in G
if allCovered(G,a)
        return 1;
return 0
```

3. Si consideri il seguente programma per il calcolo dei numeri di Fibonacci

```
int fib (int n) {
if (n < 2)
        return 1;
return fib(n-1) + fib(n-2);
}
```

Specificare il numero di chiamate ricorsive effettuate quando n = 4. Esprimere la complessità in funzione del parametro n.

```
4
|  \
3   2
|  \   | \
2  1  1  0
|  \
1  0
```

Come si vede dall'albero le chiamate per n = 4 sono 9. In generale vi è un limite superiore è dato dal numero di elementi dell'albero binario completo $O(2^n)$.

4. Quali sono gli ingressi e il risultato calcolato dalla macchina di Turing universale? **Perché il risultato è considerato importante?**

La MdT universale prende in input la codifica di una MdT e l'input a cui applicare la MdT. Essa è in grado di simulare la MdT e restituirne l'output. Ciò è importante perché significa che essa costituisce un modello di calcolo.

5. Cosa si intende per problema non decidibile (o non calcolabile)? Fornire un esempio di problema non decidibile.

Un problema è non decidibile se non esiste un algoritmo che lo decide, ossia se dato un input non è possibile determinare se esso soddisfa o meno la condizione posta dal problema. Ne è un esempio il problema della fermata che dato un programma e un input x vuole sapere se l'esecuzione del programma con input x termina o meno.

6. Descrivere un analizzatore sintattico di tipo top-down specificando l'input dell'analizzatore, i possibili risultati calcolati e un possibile algoritmo per l'analisi.

7. Descrivere un automa a stati finiti che riconosce il linguaggio formato dalle stringhe binarie di a e b costituito da tutte e sole le stringhe binarie contenenti un numero dispari di a e un numero dispari di b (in qualunque posizione; quindi l'automa accetta le stringhe aaabbb, ababab, aabaabab e rifiuta le stringhe aabbb, b e aabaaba).

-> QUADERNO

8. Sia data la grammatica G = (N, T, P, S) con insieme dei simboli non terminali N = {S, A}, insieme simboli terminali T = {a, b}, assioma S e produzioni specificate nel seguito:

S → Sa (1)
S → Ab (2)
S → A (3)
A → Ab (4)
A → b (5)

Fornire un albero di derivazione della stringa bbbaa; specificare il tipo di grammatica e discutere se la grammatica è ambigua o no.

albero -> QUADERNO

La grammatica è di tipo 3 e non è ambigua, infatti ad ogni passaggio posso espandere un solo simbolo terminale e vi è una sola produzione corretta.

9. Fornire una grammatica per generare il linguaggio delle espressioni aritmetiche avente cinque simboli terminali: "id" (che rappresenta un identificatore), i simboli di operazione "+" e "–" e le parentesi tonde "(" e ")".
Ovviamente la grammatica deve generare tutte e sole le stringhe che sono aritmeticamente corrette (ad esempio non deve generare la stringa "id+((id-id)" o la stringa "id++id-id").
L'eventuale costruzione di una grammatica **non ambigua** sarà particolarmente apprezzata.

E -> TF
T -> F+ | F-
F -> id | (E)

LUGLIO 2019

1) Illustrare una riduzione – a vostra scelta - che dimostra che un problema è NP-difficile.

Un problema L è NP-difficile se ogni altro problema in NP è polinomialmente riducibile a P.

Riduzione da 3SAT a programmazione intera. (So che 3SAT è NP-difficile)
Ogni istanza di 3SAT contiene variabili booleane e ogni istanza di PI contiene variabili intere. Data una formula con n variabili, per ogni variabile xi del problema 3SAT abbiamo due variabili vi e wi. Per limitare ogni variabile di PI a 0 e 1 (in questo modo ci limitiamo ad un problema più semplice di PI con qualsiasi valore, ma se dimostriamo che il problema più semplice è difficile possiamo dedurre che lo è anche IP) aggiungiamo i vincoli
0 <= vi <= 1 e
0 <= wi <= 1
Se xi è vero vi = 1 e wi = 0, viceversa se xi = falso vi = 0 e wi = 1. Aggiungiamo poi i vincoli
1 <= vi + wi <= 1
Ora per ogni clausola di 3SAT definiamo un vincolo secondo la logica dei seguenti esempi:
C = (x1 v x2 v x3) -> v1 + v2 + v3 >= 1

C = (x1 v not x2 v not x5) -> vi + w2 + w5 >= 1

Occorre ora verificare due cose:
1) qualsiasi soluzione SAT fornisce una soluzione per IP
Se la soluzione SAT richiede che un letterale sia vero, in automatico saranno poste a 1 (o a zero) alcune variabili e, per come abbiamo definito le disuguaglianze, esse saranno sicuramente soddisfatte.
2) qualsiasi soluzione IP fornisce una soluzione per SAT
Se la soluzione IP richiede che vi sia vero allora xi è vero, viceversa vi falso richiede xi falso. In questo modo, per come abbiamo definito le disuguaglianze, ci sarà almeno un letterale vero in ogni clausola e la formula sarà soddisfatta.

Nota che da questo ragionamento non possiamo concludere che IP è NP-completo perchè non abbiamo dimostrato che IP stesso appartiene alla classe NP.

2) Definire le classi P e NP e indicare le relazioni di contenimento fra le classi.

Vedi sopra

3) Dimostrare che se un problema A è riducibile in tempo polinomiale ad un problema B e che B è riducibile in tempo polinomiale a C, allora A è riducibile a C.

Ricordiamo la definizione di riduzione polinomiale:
Un problema L1 è polinomialmente riducibile a un problema L2 se esiste una funzione f calcolabile in tempo polinomiale tale che $x \in L1$ se e solo se f(x) appartiene a L2.

Dunque abbiamo una tale funzione tra A e B, chiamiamola f(x), e un'altra tra B e C, chiamiamola g(x). Di conseguenza g(f(x)) è una funzione di riduzione tra A e C (e non f(g(x))).

4) Nell'analisi di complessità e, in particolare nella notazione O(), si ignorano le costanti moltiplicative e additive. Discutere questa scelta indicandone i vantaggi e gli svantaggi.

Vedi sopra

5) Si consideri il seguente programma per il calcolo della più lunga sottosequenza comune a due stringhe X e Y.
Input: Stringhe X e Y con n e m elementi, rispettivamente
Output: Matrice L, i = 0,.., n-1, j = 0,..m-1, la lunghezza L[i,j] della più lunga stringa che è comune sia alla stringa X[0..1] = x0x1...xi che alla stringa Y[0..j] = y0y1...yj.

```
for i =1 to n-1 do
        L[i,-1] = 0
for j =0 to m-1 do
        L[-1,j] = 0
for i =0 to n-1 do
        for j =0 to m-1 do
                if xi = yj then
```

```
            L[i, j] = L[i-1, j-1] +1
else
            L[i, j] = max{L[i-1, j], L[i, j-1]}
```

Eseguire il programma per le stringhe X = “GMJTAUZ” e Y = “MZJAWGU” fornendo la **matrice L calcolata**. Esprimere il costo del programma in funzione della lunghezza dell’input. Motivare le risposte.

n = 7, m = 7

i/j	-1	0 m	1 mz	2 mzj	3 mzja	4 mzjaw	5 mzjawg	6 mzjawgu
-1	_	_	0	0	0	0	0	0
0 g	_	0	0	0	0	0	1	1
1 gm	0	1	1	1	1	1	2	2
2 gmj	0	1	1	2	2	2	3	3
3 gmjt	0	1	1	2	2	2	3	3
4 gmjta	0	1	1	2	3	3	4	5
5 gmjtau	0	1	1	2	3	3	4	5
6 gmjtauz	0	1	2	3	4	4	5	6

Il costo è dato dall’istruzione di costo costante compiuta nel doppio ciclo che è O(n*m) (ogni stringa è (#byte usati per un carattere)*(lunghezza della stringa))

6) Descrivere un analizzatore sintattico di tipo top-down specificando l’input dell’analizzatore, i possibili risultati calcolati e un possibile algoritmo per l’analisi.

Vedi in fondo

7) Descrivere un automa a stati finiti che riconosce il linguaggio formato dalle stringhe binarie di 0 e 1 costituito da tutte e sole le stringhe binarie contenenti un numero pari di e un numero pari di 1 (in qualunque posizione; quindi l’automa accetta le stringhe 0011, 01010101, 00100100 e rifiuta le stringhe 0111 e 00100).

-> QUADERNO

8) Sia data la grammatica G = (N, T, P, S) con insieme dei simboli non terminali N = {S, A}, insieme simboli terminali T = {a, b}, assioma S e produzioni specificate nel seguito:
S → aS (1)

S → aA (2)
A → bA (3)
A → b (4)
Fornire un albero di derivazione della stringa aabbb; specificare il linguaggio generato dalla grammatica e discutere se la grammatica è ambigua o no.

albero -> QUADERNO

L = {$a^n b^m$, con n, m>0}

Non è ambigua.
9) Fornire una grammatica non ambigua per generare il linguaggio delle espressioni aritmetiche avente cinque simboli terminali: id (che rappresenta un identificatore), i simboli di operazione + e – e le parentesi tonde (e). Ovviamente la grammatica deve generare tutte e sole le stringhe che sono corrette (ad esempio non deve generare la stringa id+((id-id) o la stringa id++id-id). Motivare inoltre perché questo linguaggio non può essere generato da una grammatica di tipo 3.

DA FARE

- Dimostro che vertex cover è NP-completo **Sbagliato, 2CNF non è completo. Si può ridurre da independent set a 3SAT e da vertex cover a independent set.**

Per la dimostrazione che è NP vedi sopra (con l'algoritmo)
Rimane da mostrare che è completo. Per far ciò riduco da 2CNF (che è NP-completo) a Vertex Cover.
Risolvere vertex cover significa che per ogni lato devo inserire nell'insieme almeno uno dei due nodi adiacenti. Ossia per ogni lato i voglio che sia vero (xi v yi), con xi e yi i due nodi adiacenti. Mettendo insieme per tutti gli archi ottengo
F = (x1 v y1) ∧ (x2 v y2) … (xm v ym) (alcuni nodi possono coincidere, ad esempio si può avere x1 = y2)
Alla fine di F, sia s la variabile definita come x1 + y1 + x2 + y2 +... + xm + ym = k, aggiungo la clausola … ∧ (s)
In questo modo ogni soluzione F è verificata se e solo se lo è vertex cover, e viceversa non lo è se e solo se non lo è vertex cover.

APPUNTI

PARSING LL(1) (TOP-DOWN)
1) Elimino le ricorsioni sinistre
2) Calcolo FIRST e FOLLOW
E → TE'
E' → +TE' | ε
T → FT'
T' → ∗FT' | ε
F → (E) | id

FIRST(E) = FIRST(T) = {(, id}

FIRST(E') = {+, ε}
FIRST(T) = FIRST(F) = {(, id}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}

FIRST(TE') = {(, id}
FIRST(+TE') = +
FIRST(FT') = {(, id}
FIRST(*FT') = *
FIRST((E)) = (

1) aggiungi il $ in FOLLOW (S)
2) per ogni produzione della forma A → αBβ, aggiungi in FOLLOW(B) ogni terminale in FIRST(β)
3) per ogni produzione della forma A → αB o della forma A → αBβ con β ⇒* ε, aggiungi in FOLLOW(B) ogni simbolo in FOLLOW(A)

FOLLOW(E) = {$,)}
FOLLOW(E') = FOLLOW(E) = {$,)}
FOLLOW(T) = FIRST(E') + FOLLOW(E) + FOLLOW(E') = {+, $,)}
FOLLOW(T') = FOLLOW(T) = {+, $,)}
FOLLOW(F) = FIRST(T') + FOLLOW(T) + FOLLOW(T') = {*, +, $,)}

3) Costruisco la tabella
1. Per ogni produzione A → α applica i passi 2 e 3
2. Per ogni terminale a ∈ FIRST(α), aggiungi A → α in M(A, a)
3. Se ε ∈ FIRST(α) aggiungi A → α in M(A, b) per ogni simbolo (anche il $) b ∈ FOLLOW (A)
4. Poni ogni entrata indefinita ad error

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT			FT		
T'		ε	*FT'		ε	ε
F	id			(E)		

PARSING LR (BOTTOM-UP)
S -> aSc | ac | T
T -> bTc | bc

1) aggiungo S' -> S
2) set of item

s0:
S’ -> .S
S -> .aSc | .ac | .T
T -> .bTc | .bc

s0 -> s1 (a)
s1:
S -> a.Sc | a.c
S -> .aSc | .ac | .T
T -> .bTc | .bc

s1 -> s1 (a)

s1 -> s2 (b)

s1 -> s3 (T)

s0 -> s2 (b)
s2:
T -> b.Tc | b.c
T -> .bTc | .bc

s0 -> s3 (T)
s3:
S’ -> T.

s0 -> s4 (S)
s4:
S’ -> S.

s1 -> s5 (c)
s5:
S -> ac.

s1 -> s6 (S)
s6:
S -> aS.c

s2 -> s2 (b)

s2 -> s7 (c)
s7:
T -> bc.

s2 -> s8 (T)
s8:
T -> bT.c

s6 -> s9 (c)
s9:
S -> aSc.

s8 -> s10 (c)
s10:
T -> bTc.

Altri stati (**da dove cicciano?**):
s12: (T)
T -> bT.c

s13:
T -> b.Tc | b.c
T -> .bTc | .bc

s12 -> s14 (c)
s14:
T -> bTc. (non è uguale a s10?)

s13 -> s12 (T)

s13 -> s15 (c)
s15:
T -> bc.

s13 -> s13 (b)

3) Costruzione tavole
• Un arco fra due stati sa e sb etichettato con simbolo terminale x: equivale a passare da sa a sb quando input è x (shift): si inserisce sb in pila
• Quando si giunge in uno stato con . alla fine bisogna eseguire reduce: si tolgono tanti stati dalla pila quanti sono i simboli a destra della produzione e si inserisce un nuovo stato in pila (tavola Goto)
• Goto[s,U]: usa le transizioni tra stati etichettate con simbolo non terminale

Stato	Action				Goto	
	a	b	c	$	S	T
0	S1	S2			goto 4	goto 3
1	S1	S2	S5		goto 6	goto 3
2		S2	S7			goto 8
3	R: S->T	R: S->T	R: S->T	R: S->T		

4				**A**		
5	R: S->ac	R: S->ac	R: S->ac	R: S->ac		
6			S9			
7	R: T->bc	R: T->bc	R: T->bc	R: T->bc		
8			S10			
9	R: S->aSc	R: S->aSc	R: S->aSc	R: S->aSc		
10	R: T->bc	R: T->bc	R: T->bc	R: T->bc		