Complessità di problemi e algorithmi

Sergio Flesca[1]

[1] DIMES - Università della Calabria, Via P. Bucci, 87036 Rende (CS) - Italy
flesca@dimes.unical.it

Outline

Definizione di problema

Cos'è un problema in Informatica?

Definizione di problema

- Siano dati due insiemi I, cioè l'insieme dei possibili input, e O, cioè l'insieme dei possibili output.
- Un problema P è una funzione da I a O ($P : I \rightarrow O$)

Definizione di problema

Cos'è un problema in Informatica?

Definizione di problema

- Siano dati due insiemi I, cioè l'insieme dei possibili input, e O, cioè l'insieme dei possibili output.
- Un problema P è una funzione da I a O ($P : I \rightarrow O$)

Algoritmo risolutore di un problema

Quando possiamo dire che un algoritmo risolve un problema?

- Sia dato un problema P ($P : I \to O$) e un algoritmo A.
- Innanzitutto osserviamo che i possibili output di A devono includere anche la possibilità che A non termini.
- Indichiamo quindi con il simbolo $\perp \notin O$ la non terminazione di A, ed inoltre per ogni $x \in I$ indichiamo con $A(x)$ il risultato dell'esecuzione di A avviato sull'input x ($A(x) \in O \cup \{\perp\}$).

Un algoritmo A si dice risolutore del problema P se e solo se

$$\forall x \in I, P(x) = A(x)$$

Algoritmo risolutore di un problema

Quando possiamo dire che un algoritmo risolve un problema?

- Sia dato un problema P ($P : I \to O$) e un algoritmo A.
- Innanzitutto osserviamo che i possibili output di A devono includere anche la possibilità che A non termini.
- Indichiamo quindi con il simbolo $\perp \notin O$ la non terminazione di A, ed inoltre per ogni $x \in I$ indichiamo con $A(x)$ il risultato dell'esecuzione di A avviato sull'input x ($A(x) \in O \cup \{\perp\}$).

Un algoritmo A si dice risolutore del problema P se e solo se

$$\forall x \in I, P(x) = A(x)$$

Algoritmo risolutore di un problema

Quando possiamo dire che un algoritmo risolve un problema?

- Sia dato un problema P ($P : I \to O$) e un algoritmo A.
- Innanzitutto osserviamo che i possibili output di A devono includere anche la possibilità che A non termini.
- Indichiamo quindi con il simbolo $\perp \notin O$ la non terminazione di A, ed inoltre per ogni $x \in I$ indichiamo con $A(x)$ il risultato dell'esecuzione di A avviato sull'input x ($A(x) \in O \cup \{\perp\}$).

Un algoritmo A si dice risolutore del problema P se e solo se

$$\forall x \in I, P(x) = A(x)$$

Algoritmo risolutore di un problema

Quando possiamo dire che un algoritmo risolve un problema?

- Sia dato un problema P ($P : I \to O$) e un algoritmo A.
- Innanzitutto osserviamo che i possibili output di A devono includere anche la possibilità che A non termini.
- Indichiamo quindi con il simbolo $\perp \notin O$ la non terminazione di A, ed inoltre per ogni $x \in I$ indichiamo con $A(x)$ il risultato dell'esecuzione di A avviato sull'input x ($A(x) \in O \cup \{\perp\}$).

Un algoritmo A si dice risolutore del problema P se e solo se

$$\forall x \in I, P(x) = A(x)$$

Algoritmo risolutore di un problema

Quando possiamo dire che un algoritmo risolve un problema?

- Sia dato un problema P ($P : I \to O$) e un algoritmo A.
- Innanzitutto osserviamo che i possibili output di A devono includere anche la possibilità che A non termini.
- Indichiamo quindi con il simbolo $\perp \notin O$ la non terminazione di A, ed inoltre per ogni $x \in I$ indichiamo con $A(x)$ il risultato dell'esecuzione di A avviato sull'input x ($A(x) \in O \cup \{\perp\}$).

Un algoritmo A si dice risolutore del problema P se e solo se

$$\forall x \in I, P(x) = A(x)$$

I problemi risolvibili (o calcolabili)

I problemi risolvibili (o calcolabili) sono quelli per i quali esiste un algoritmo di risolutore.

- Non tutti i problemi sono risolvibili.
- Esempio: problema delle corrispondenze
  - dati due insiemi di stringhe S_1 e S_2, esiste una stringa x tale che x è contemporaneamente la concatenazione di stringhe in S_1 e la concatenazione di stringhe in S_2?

I problemi risolvibili (o calcolabili)

I problemi risolvibili (o calcolabili) sono quelli per i quali esiste un algoritmo di risolutore.

- Non tutti i problemi sono risolvibili.
- Esempio: problema delle corrispondenze
  - dati due insiemi di stringhe S_1 e S_2, esiste una stringa x tale che x è contemporaneamente la concatenazione di stringhe in S_1 e la concatenazione di stringhe in S_2?

Analisi di complessità per casi

Esiste sempre la funzione che associa la dimensione dell'input al costo di escuzione di un algoritmo?

Purtropo no!

Nella maggioranza di casi se lanciamo un algoritmo A su due input x_1 e x_2 il tempo che ci metterà A ad eseguire x_1 sarà diverso dal tempo che ci metterà ad eseguire su x_2.

Come si può ovviare a questo problema?

Analisi di complessità per casi

Esiste sempre la funzione che associa la dimensione dell'input al costo di escuzione di un algoritmo?

Purtropo no!

Nella maggioranza di casi se lanciamo un algoritmo A su due input x_1 e x_2 il tempo che ci metterà A ad eseguire x_1 sarà diverso dal tempo che ci metterà ad eseguire su x_2.

Come si può ovviare a questo problema?

Analisi di complessità per casi

Esiste sempre la funzione che associa la dimensione dell'input al costo di escuzione di un algoritmo?

Purtropo no!

Nella maggioranza di casi se lanciamo un algoritmo A su due input x_1 e x_2 il tempo che ci metterà A ad eseguire x_1 sarà diverso dal tempo che ci metterà ad eseguire su x_2.

Come si può ovviare a questo problema?

Analisi di complessità per casi

Sia $T_A(x)$ il costo di esecuzione di A su x e con $|x|$ la dimensione di X.
Si definiscono le seguenti funzioni:

- $T_A^{worst}(n) = \max(\{T_A(x) | x \in I \wedge |x| = n\})$,
- $T_A^{best}(n) = \min(\{T_A(x) | x \in I \wedge |x| = n\})$

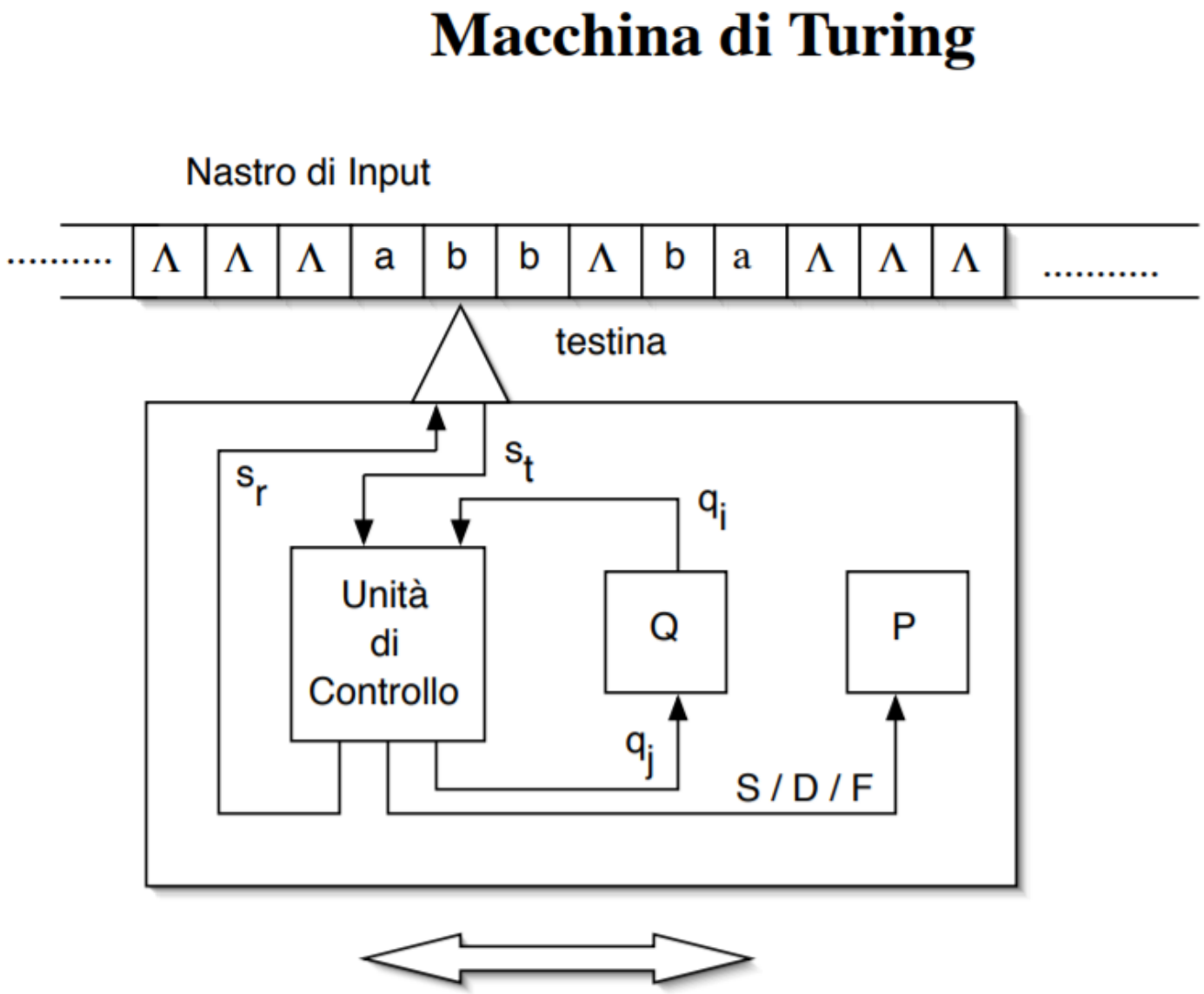
Se si considera la distribuzione di probabilità che le istanze siano effettivamente date in input all'algoritmo ($Pr(x)$) allora si può definire anche la funzione

- $T_A^{avg}(n) = \sum_{x \in I \wedge |x| = n} T_A(x) \times Pr(x)$

Lo studio della complessità di un algoritmo si esplica quindi lavorando per casi, analizzare una algoritmo A nel caso

- peggiore, vuol dire caratterizzare la funzione T_A^{worst},
- migliore, vuol dire caratterizzare la funzione T_A^{best},
- medio, vuol dire caratterizzare la funzione T_A^{avg}.

Modelli teorici: macchina di Turing



Studio di complessità in linguaggi ad alto livello: modello a costi uniformi

- Si assume che le operazioni elementari abbiano tutte un costo unitario.
- Il costo di esecuzione dell'algoritmo è quindi dato dalla somma del numero di volte che le operazioni vengono ripetute durante l'esecuzione dell'algoritmo.

Il costo dell'algoritmo è quindi dato da:

- $T_A(n) = \sum_{ist_i \in Ist} num(ist_i)$, dove *Ist* è l'insieme di tutte le istruzioni elementari dell'algoritmo e $num(ist_i)$ rappresenta il numero di volte che l'istruzione ist_i è eseguita durante l'esecuzione di *A*.

Studio di complessità in linguaggi ad alto livello: metodo dell'istruzione dominate

Definition (Istruzione dominante)

Dato un algoritmo A il cui costo di esecuzione è $T_A(n)$ definiamo operazione dominante di A un'istruzione elementare *ist* di A tale che se indichiamo con $num_{ist}(n)$ il numero di volte che *ist* è eseguita durante l'esecuzione di A su un input di dimensione n $num_{ist}(n) \in \theta(T_A(n))$[a].

[a] Chiaramente tale definizione è generica e va specializzata per l'analisi per casi

```
public static boolean presente(int[] v, int x){
    for(int i =0; i<v.length; i++)
        if(v[i]==x)
            return true;
    return false;
}
```

Figure: Esempio di istruzioni elementari dominanti nel caso peggiore (sottolineate)

Un modello più preciso: il modello a costi logaritmici

- Il metodo di analisi basato sul modello a costi uniformi è certamente il più semplice e più usato per determinare il costo di esecuzione di un programma.
- Un problema che può mettere in dubbio la validità dei risultati ottenuti applicando il metodo dell'analisi asintotica utilizzando il modello a costi uniformi è però il seguente:
  - Nel modello a costi uniformi ogni istruzione ha un costo unitario indipendentemente dal valore degli operandi
    - ad esempio, la moltiplicazione di due numeri costa sempre 1 sia che si moltiplichino numeri di 8 bit sia che si moltiplichino numeri di 1000 bit.
    - non sarebbe però un problema se la dimensione dei numeri si mantenesse la stessa durante tutto l'esecuzione dell'algoritmo (ad esempio se fosse $\theta(\log n)$ basterebbe moltiplicare la complessità ottenuta con il modello a costi uniformi per $\log n$ e si otterrebbe una stima più precisa della complessità)
    - **ci sono casi dove non è cosi semplice?**

Un modello più preciso: il modello a costi logaritmici

- Il nel *modello a costi logaritmici* si assume che il costo temporale di ogni istruzione sia proporzionale alla dimensione (numero di bit) degli operandi (ovvero al logaritmo del loro valore).
- Ad esempio se eseguiamo la moltiplicazione tra due interi n ed m il costo non è più 1 ma $\log n + \log m$.

```
public static int fact1 (int n)
{     int fattoriale=1;
      for(int i=1; i<=n; i++)
            fattoriale*=i;
      return fattoriale;
}
```

Qual'è la complessita?

Un modello più preciso: il modello a costi logaritmici

- Il nel *modello a costi logaritmici* si assume che il costo temporale di ogni istruzione sia proporzionale alla dimensione (numero di bit) degli operandi (ovvero al logaritmo del loro valore).
- Ad esempio se eseguiamo la moltiplicazione tra due interi n ed m il costo non è più 1 ma $\log n + \log m$.

```
public static int fact1 (int n)
{     int fattoriale=1;
      for(int i=1; i<=n; i++)
            fattoriale*=i;
      return fattoriale;
}
```

Qual'è la complessita?
$\theta(n \log n)$?

Un modello più preciso: il modello a costi logaritmici

- Il nel *modello a costi logaritmici* si assume che il costo temporale di ogni istruzione sia proporzionale alla dimensione (numero di bit) degli operandi (ovvero al logaritmo del loro valore).
- Ad esempio se eseguiamo la moltiplicazione tra due interi n ed m il costo non è più 1 ma $\log n + \log m$.

```
public static int fact1 (int n)
{     int fattoriale=1;
      for(int i=1; i<=n; i++)
            fattoriale*=i;
      return fattoriale;
}
```

Qual'è la complessita?
No è $\theta(n^2 \log n)$! poichè $\theta(n \log n!)$ è $\theta(n \log n^n)$ è $\theta(n \times n \times \log n)$

Complessità intriseca di un problema

- Per caratterizzare la complessità di un problema abbiamo bisogno, fondamentalmente, di due riferimenti:
- *Limite superiore:* quale è, nel caso peggiore, la quantità di tempo (o di memoria) sufficiente per la risoluzione del problema dato.
  - Per trovare un limite superiore è sufficiente conoscere qualche algoritmo (non necessariamente il migliore) per la risoluzione del problema dato e valutarne la complessità con uno dei metodi visti nelle sezioni precedenti
  - La conoscenza di un upper bound, tuttavia, non ci dà un'informazione completa sulla complessità del problema, infatti potrebbero esistere metodi di risoluzione del problema dato diversi da quelli noti che ne consentono la soluzione in modo molto più rapido.
- *Limite inferiore:* è un'indicazione della quantità di tempo (o di memoria) che, sempre nel caso peggiore, è sicuramente necessaria (anche se non sufficiente) per la risoluzione del problema.
  - Per determinare un lower bound di complessità di un problema è necessario, infatti, dimostrare che nessun algoritmo (**anche non noto**) per la risoluzione del problema stesso può fare a meno di utilizzare una certa quantità di risorsa (tempo, memoria) o di eseguire un certo numero di operazioni (ad esempio quadratico) almeno in una serie di casi particolarmente difficili.

Complessità intriseca di un problema

Definition (Upper bound alla complessità di un problema)

Un problema P ha un *upper bound* di complessità $O(g(n))$ se e solo se esiste un algoritmo A risolutore di P che ha complessità $O(g(n))$ nel caso peggiore.

Definition (Lower bound alla complessità di un problema)

Un problema P ha un *lower bound* di complessità $\Omega(g(n))$ se e solo se tutti gli algoritmi A risolutori di P hanno complessità $\Omega(g(n))$ nel caso peggiore.

Il lower bound alla complessita di un problema è anche detto **complessità intrinseca del problema**.

Algoritmi ottimali

Definition (Algoritmi ottimali)

Dato un problema P avente complessità intrinseca $\Omega(g(n))$, un algoritmo A risolutore di P si dice ottimale per P se A ha complessità $O(g(n))$ nel caso peggiore.

Metodi elementari per determinare la complessità intrinseca

- Un primo metodo che possiamo adottare consiste semplicemente nell'osservare che ogni algoritmo risolutore di un problema deve produrre l'output previsto dal problema.
- Un'altra possibilita potrebbe essere quella di dimostrare che ogni algoritmo deve leggere tutto l'input per rispondere correttamente al problema

Metodi elementari per determinare la complessità intrinseca

- Un primo metodo che possiamo adottare consiste semplicemente nell'osservare che ogni algoritmo risolutore di un problema deve produrre l'output previsto dal problema.
- Un'altra possibilita potrebbe essere quella di dimostrare che ogni algoritmo deve leggere tutto l'input per rispondere correttamente al problema (**É sempre vero?**)

Metodi elementari per determinare la complessità intrinseca

- Un primo metodo che possiamo adottare consiste semplicemente nell'osservare che ogni algoritmo risolutore di un problema deve produrre l'output previsto dal problema.
- Un'altra possibilita potrebbe essere quella di dimostrare che ogni algoritmo deve leggere tutto l'input per rispondere correttamente al problema (**É sempre vero?**)

La ***tecnica dell'avversario*** ci aiuta a trattare il secondo caso e altri casi simili.

- Essa consiste nel supporre che, dato un ipotetico algoritmo che risolve il problema assegnato con un certo costo, un invisibile avversario possa alterare i dati in ingresso in modo da mettere in crisi l'algoritmo e mostrare che il costo deve necessariamente essere maggiore.

Tecnica dell'avversario

- Sia dato un problema P e una funzione $f(n)$, vogliamo dimostrare che la complessità intrinseca di P è $\Omega(f(n))$.
- Ragionando per assurdo supponiamo che vi sia un algoritmo A risolutore di P che risolve P in "meno" di $\Theta(f(n))$ passi.
- Il fatto che A risolve P in "meno" di $\Theta(f(n))$ implica che per ogni input x di dimensione n A impiega "meno" di $\Theta(f(n))$ passi per risolvere x.
- sfruttiamo il fatto che l'esecuzione di A su x dipende esclusivamente dalla porzione di x cui A accede.
  - Ovverosia, se quando A viene lanciato su x legge una porzione x' di x e restituisce y, quando A viene lanciato su un input $\hat{x}$, che è identico a x sulla porzione x' di x, restituirà y.
  - Se possiamo scegliere un input $\hat{x}$ per il quale $P(x) \neq P(\hat{x})$ abbiamo dimostrato che A non è risolutore di P, arrivando ad un assurdo e quindi la complessità intrinseca di P deve essere $\Omega(f(n))$.

Tecnica dell'avversario - complessità intrinseca della ricerca di un elemento in un vettore non ordinato

Applichiamo la tecnica per dimostrare che la complessità intrinseca è $\Omega(n)$

- Ragionando per assurdo, supponiamo che esista un algoritmo A che è in grado di risolvere il problema su tutte le istanze di dimensione n (considerando come dimensione il numero di celle del vettore) non accedendo ai valori di tutte le celle (Se il vettore ha n celle A ne leggerà al più $n - 1$)

Tecnica dell'avversario - complessità intrinseca della ricerca di un elemento in un vettore non ordinato

Applichiamo la tecnica per dimostrare che la complessità intrinseca è $\Omega(n)$

- Diamo in input ad A un vettore di n interi $v = [x_1, \ldots, x_n]$ e un intero x tale che $\forall i \in [1..n] x_i \neq x$. A invocato su v e x restituirà falso (A è un algoritmo risolutore e deve restiutire la risposta corretta). Inoltre, per ipotesi di contraddizione dovra esistere almeno un valore $i' \in [1..n]$ tale che A non guarderà il valore di $x_{i'}$.

Tecnica dell'avversario - complessità intrinseca della ricerca di un elemento in un vettore non ordinato

Applichiamo la tecnica per dimostrare che la complessità intrinseca è $\Omega(n)$

- Consideriamo quindi il vettore v' identico a v su tutte le celle tranne che sulla cella i' in cui verrà inserito il valore x.

Tecnica dell'avversario - complessità intrinseca della ricerca di un elemento in un vettore non ordinato

Applichiamo la tecnica per dimostrare che la complessità intrinseca è $\Omega(n)$

- L'input v', x è identico all'input v, x sulle parti di v, x che A ha "guardato" per cui se invochiamo A su v', x il risultato sarà lo stesso di quando lo abbiamo invocato su v, x, ovverosia A restituirà falso. Ma la risposta corretta su v', x è vero, quindi A non è un algoritmo risolutore per la ricerca di un elemento in un vettore non ordinato.

Tecnica dell'avversario - complessità intrinseca della ricerca di un elemento in un vettore non ordinato

Applichiamo la tecnica per dimostrare che la complessità intrinseca è $\Omega(n)$

- L'input v', x è identico all'input v, x sulle parti di v, x che A ha "guardato" per cui se invochiamo A su v', x il risultato sarà lo stesso di quando lo abbiamo invocato su v, x, ovverosia A restituirà falso. Ma la risposta corretta su v', x è vero, quindi A non è un algoritmo risolutore per la ricerca di un elemento in un vettore non ordinato.

Abbiamo dimostrato quindi che la complessità intrinseca del problema di cercare un elemento in un vettore ordinato è $\Omega(n)$, l'algoritmo della ricerca lineare è quindi un algoritmo ottimale.

Un'altra tecnica per determinare lower bounds: gli alberi di decisione

Questa tecnica si basa sulla rappresentazione di un algoritmo mediante un albero di decisione.

Definition (Alberi di decisione)

Un albero di decisione è un albero in cui ogni nodo interno rappresenta un'operazione di confronto, i rami da esso uscenti corrispondono ai due esiti possibili ($>$ oppure $\leq$) e ogni foglia rappresenta l'esito di una serie di confronti.

Utilizzando questa tecnica si può tenere conto solamente degli algoritmi che producono il loro output esclusivamente sulla base dei confronti effettuati (anche detti *algoritmi basati su confronti*).

Un'altra tecnica per determinare lower bounds: gli alberi di decisione

Sia dato l'insieme di chiavi {ALFA, BETA, CHI, DELTA, EPSILON, ETA} ordinate in ordine alfabetico

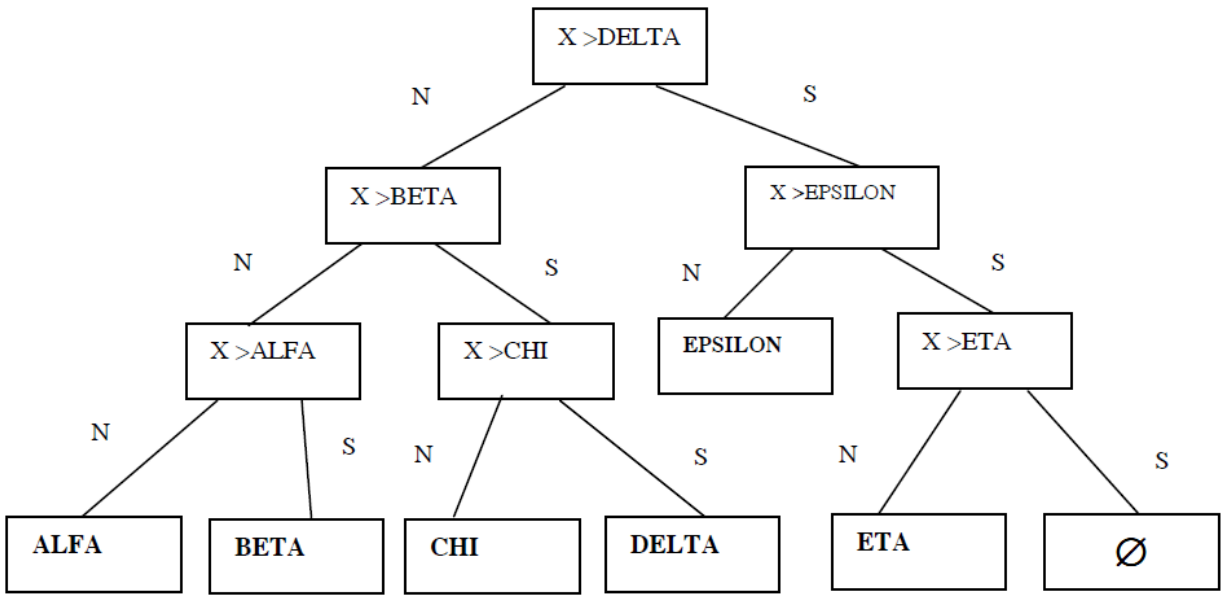


Figure: Albero di decisione corrispondente alla ricerca binaria

Un'altra tecnica per determinare lower bounds: gli alberi di decisione

Sia dato l'insieme di chiavi {ALFA, BETA, CHI, DELTA, EPSILON, ETA} ordinate in ordine alfabetico

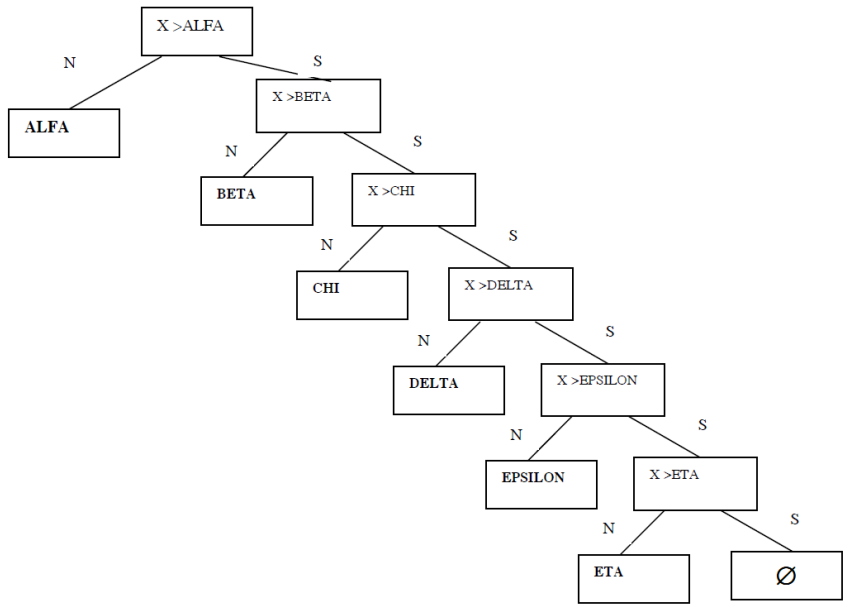


Figure: Albero di decisione corrispondente alla ricerca lineare

Un'altra tecnica per determinare lower bounds: gli alberi di decisione

- Come si determina il numero minimo di confronti che un algoritmo basato su confronti deve obbligatoriamente fare per risolvere un problema?
- Prendiamo in considerazione tutti i possibili input di dimensione *n*, e "contiamo tutte le risposte diverse che per questi input devono essere date. Ad esempio:
  - per il problema dell'ordinamento di un vettore di lunghezza n le risposte possibili corrispondono alle permutazioni degli elementi contenuti nel vettore (Quante sono?)
  - per il problema della ricerca di un elemento in un vettore ordinato le possibili risposte corrispondono alle posizioni in cui può trovarsi l'elemento.

Un'altra tecnica per determinare lower bounds: gli alberi di decisione

- Se un algoritmo (basato su confronti) è risolutore di un problema il numero di foglie presenti nell'albero di decisione corrispondente alle possibili esecuzioni su input di dimensione n deve essere maggiore o uguale al numero di risposte diverse che per questi input devono essere date
- Un qualunque algoritmo che fa al più x confronti su un input di dimensione n al massimo quante possibili risposte diverse può dare?
  - Facendo 1 solo confronto posso dare 2 risposte diverse, con 2 confronti 4, con tre 8, con k 2^k.
- quindi per il problema dell'ordinamento deve risultare $2^k \geq n! \approx n^n$, ovverosia (applicando il logaritmo) $k \geq n \log n$. Quindi il lower bound dell'ordinamento per algoritmi basati su confronti è $\Omega(n \log n)$.
- per la ricerca su domini ordinati invece $2^k \geq n + 1$ ovversia $k \geq \log(n + 1)$, quindi la ricerca in domini ordinati (per algoritmi basati su confronti) è $\Omega(\log n)$.

Un po di relax

Indovinello

Siano date 12 monete, tra cui una di peso diverso dalle altre, e una bilancia a due piatti. Stabilire con 3 pesate quale sia la moneta di peso diverso, e se è più pesante o più leggera delle altre.

1. Possiamo dimostrare che con 14 monete è impossibile stabilire qual sia la moneta di peso diverso (se c'è) e se è più leggera o più pesante?
2. che sappiamo per il caso con 13 monete?