ALMA MATER STUDIORUM
Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Matematica

Il problema della primalità in complessità computazionale

Tesi di Laurea in Informatica

Presentato da:
Saverio Tassinari

Relatore:
Chiar.mo Prof.
Simone Martini

Sessione II
Anno Accademico 2012 - 2013

A mia mamma e mio fratello
per avermi supportato
e sopportato fino ad oggi.

Ai miei nonni,
mi dispiace
di non aver fatto in tempo.

Al prof. Martini
per la professionalità
e la cortesia.

Agli sbregiz
per la gag.

Indice

Introduzione

Nel corso degli ultimi decenni la teoria della complessità computazionale ha visto uno sviluppo incredibile, risolvendo molti dei problemi introdotti nel corso del XX secolo in seguito alla nascita della teoria dell'informazione. Tra le altre, la volontà di approfondire argomenti spesso solo accennati nei programmi del corso di laurea e la voglia di conoscere da vicino una delle più grandi svolte degli ultimi anni, sono alcune delle motivazioni che mi hanno spinto a scrivere la qui presente tesi, suddivisa in tre capitoli strutturati come segue.

Il primo introduce la teoria della complessità computazionale ed il concetto basilare di *Macchina di Turing*, un modello di calcolo introdotto nel 1936 da Alan Turing, considerato da molti il padre dell'informatica. Pur essendo una macchina ideale, ricopre un ruolo fondamentale in questa materia, poiché ogni altro modello di calcolo può essere ricondotto ad essa.

Sulla base della quantità di risorse che la Macchina d Turing ha a disposizione e della loro tipologia (in particolare *tempo* e *spazio*), definiremo le classi più importanti, **P** ed **NP**. La prima contiene tutti i problemi decisionali che una Macchina di Turing riesce a risolvere usando una quantità di tempo polinomiale rispetto all'input fornito. La classe **NP**, per la cui definizione dovremo introdurre una variante della Macchina di Turing classica, contiene tutti quei problemi che, pur non essendo calcolabili in tempo polinomiale, permettono di essere *verificati* in tempo polinomiale. Nella parte finale ci si domanda in che relazione stiano queste classi e si considererà il *Problema del Millennio* $\mathbf{P} \neq \mathbf{NP}$.

Nel secondo capitolo si studieranno più in profondità le due classi di cui sopra, in particolare scopriremo che esiste un sottoinsieme di **NP** di problemi cosiddetti *completi*. Questi sono informalmente i problemi più difficili di **NP**, tanto che se si riuscisse a trovare un algoritmo in tempo polinomiale per *un* problema **NP**-completo, si potrebbe sfruttare questo per dimostrare che *ogni* problema di **NP** sta in realtà in **P**. Inutile dirlo, l'esistenza di questa classe di problemi, secondo il parere di molti esperti, depone a favore dell'ipotesi che effettivamente sia $\mathbf{P} \neq \mathbf{NP}$. Seguiranno alcuni esempi di questi problemi, assieme all'ipotesi che esistano problemi detti *intermedi*, ovvero che stanno in $\mathbf{NP} \setminus \mathbf{P}$.

Nell'ultimo capitolo prenderemo in considerazione il problema dei numeri primi da un punto di vista computazionale. Uno degli strumenti più importanti è il *Piccolo Teorema di Fermat* che dà una condizione necessaria per la primalità di un numero. Si tratterà brevemente del percorso che negli ultimi 40 anni gli studiosi hanno intrapreso per tentare di classificare correttamente questo problema. Maggiore spazio sarà riservato al recente *algoritmo AKS* (dai nomi dei tre matematici che l'hanno ideato nel 2002). Questo, servendosi di una generalizzazione del teorema citato sopra e sfruttando importanti proprietà dei polinomi su campi finiti, stabilisce che il problema dei numeri primi appartiene alla classe **P**. In particolare sarà esposto l'algoritmo e sarà dimostrata la sua correttezza.

Per motivi di sintesi sono state omesse alcune dimostrazioni, quindi per approfondimenti si rimanda alla bibliografia che si trova nella parte finale di questo documento.

Capitolo 1

Introduzione alla complessità computazionale

In questo capitolo verrà introdotta la complessità computazionale, partendo dalla definizione di *Macchina di Turing*, spiegando nel dettaglio i motivi per cui è il modello di riferimento per questa materia. Definiremo poi le principali classi di complessità in base alle risorse prese in considerazione.

1.1 La Macchina di Turing

Per fissare la nozione di algoritmo, nel 1936 Alan Turing propose un modello, chiamato ora proprio *Macchina di Turing* (in breve MdT), estremamente elementare, ma, come vedremo, tuttora utilizzato. Definire con precisione questo strumento richiede, però, un formalismo apparentemente complicato, quindi risulta utile descriverlo passo per passo.

Una MdT è costituita da un'unità di controllo e da k nastri su cui operano altrettanti dispositivi mobili di lettura e scrittura detti *testine*. Questi nastri sono finiti a sinistra e infiniti a destra e possono essere visti come una sequenza di *celle*, ognuna delle quali può contenere uno e un solo simbolo appartenente all'*alfabeto* della macchina. Le testine possono leggere un simbolo alla volta e si muovono ad

istanti fissati $t_0, t_1, t_2, \ldots$. Per comodità di comprensione, supporremo sempre che i nastri siano sempre almeno $k \geq 3$, in modo da avere come primo il *nastro di ingresso*, su cui è consentita la sola lettura, e come ultimo quello di *uscita*, dove la macchina può soltanto scrivere. I restanti $k-2$ nastri sono detti *nastri di lavoro* e su di essi la macchina può sia leggere che scrivere.
Ad ogni passo della computazione di una MdT viene eseguita una sequenza di operazioni in questo modo:

- l'unità di controllo modifica il proprio stato in base allo stato attuale e ai simboli letti dalle testine (gli stati in questione devono essere *finiti*);
- ogni testina, a parte quella di lettura, ha la possibilità di scrivere un nuovo simbolo dell'alfabeto sulla cella corrente;
- ogni testina può spostarsi a destra o sinistra oppure restare ferma.

In maniera più rigorosa:

Definizione 1.1.1 (Macchina di Turing)**.** *Una* Macchina di Turing *(o MdT) a k nastri è una 7-upla $M = (Q, \Gamma, \Sigma, q_0, F, b, \delta)$ dove:*

1. *Q è l'insieme finito non vuoto degli stati in cui si può trovare l'unità di controllo;*
2. *Γ è l'alfabeto della macchina, ovvero un insieme finito non vuoto di simboli che possono comparire nelle celle dei $k-1$ nastri diversi da quello di ingresso;*
3. *$\Sigma \subseteq \Gamma \setminus \{b\}$ è l'alfabeto di ingresso, ovvero l'insieme dei simboli che si possono trovare nel nastro di ingresso (questo insieme non include il simbolo speciale di* spazio *definito dopo, ed è quindi un sottoinsieme proprio di Γ);*
4. *$q_0 \in Q$ è lo stato iniziale, cioè lo stato in cui l'unità di controllo si trova all'istante t_0;*
5. *$F \subseteq Q$ è l'insieme degli stati finali;*
6. *$b \in \Gamma \setminus \Sigma$ è il simbolo che denota il carattere speciale spazio;*

7. *δ è detta funzione di transizione, ovvero la funzione che determina i cambiamenti di stato della macchina, ed è così definita:*

$$\delta : Q \times \Gamma^{k-1} \longrightarrow Q \times \big(\Gamma \times \{S, D, F\}\big)^{k-1}$$

dove i simboli S e D indicano rispettivamente gli spostamenti a sinistra e a destra della testina, mentre F indica che questa resta ferma.

A prima vista questa macchina potrebbe sembrare obsoleta e lontana dai moderni calcolatori, ma è proprio qui che risiede l'importanza della MdT: tutti gli strumenti introdotti nell'ultimo secolo per formalizzare l'idea di algoritmo, e quindi per stabilire quali problemi siano calcolabili meccanicamente e quali no, sebbene sempre più veloci e potenti, sono tutti riconducibili alla Macchina di Turing.
Questa affermazione, centrale sia nella teoria della complessità computazionale come nella calcolabilità, è una congettura largamente ritenuta valida ed è nota come *Tesi di Church-Turing*[3].

Tesi di Church-Turing
Una funzione calcolabile in un qualsiasi ragionevole modello di calcolo è calcolabile attraverso una MdT.

La MdT risulta allora semplice e potente allo stesso momento ed è lo strumento privilegiato per ragionare in maniera efficiente di calcolabilità.

Ora è necessario dare qualche definizione, utile per capire meglio il funzionamento della MdT e gettare le basi per comprendere al meglio il concetto di complessità e quindi di *classe* di complessità.

Definizione 1.1.2. *Siano M una MdT e Σ l'alfabeto su cui M opera. Si dice che M riconosce il* linguaggio L_M *se, dato in input x, essa termina in uno stato di accettazione (o di rifiuto) se e solo se $x \in L_M$ (o $x \notin L_M$).*

Nella definizione di MdT, essendo questa una macchina teorica, non abbiamo dato limiti su alcuna risorsa, visto che la calcolabilità ricerca solo se un dato

problema sia o meno risolvibile per via algoritmica. Per stabilire se un problema calcolabile sia effettivamente risolubile in maniera soddisfacente occorre accertarsi che la MdT abbia un costo (nel nostro caso in tempo e spazio) accettabile.

Definizione 1.1.3. *Una MdT* lavora in tempo $T_M(n)$ *se, per ogni stringa in input di lunghezza n, la computazione termina in al più $T_M(n)$ passi, e ha un* costo in spazio $S_M(n)$ *se la computazione utilizza in tutto non più di $S_M(n)$ celle di nastro distinte sui nastri di lavoro.*

Definizione 1.1.4. *Diciamo che M riconosce il linguaggio L in tempo $T_M(n)$ se M riconosce L e lavora in tempo $T_M(n)$ e che M riconosce il linguaggio L in spazio $S_M(n)$ se M riconosce L e lavora in spazio $S_M(n)$.*

Per quanto riguarda la complessità computazionale, si potrebbe essere portati a pensare, come prima, che le macchine che riescono a compiere una maggiore quantità di operazioni simultanee siano più indicate ad analisi in cui l'efficienza è fondamentale. Vediamo, al contrario, come anche in questo frangente la MdT sia essenziale per i nostri scopi grazie alla cosiddetta *Tesi del calcolo sequenziale.*

Definizione 1.1.5. *Siano A e B due modelli di calcolo. Diciamo che A e B sono* polinomialmente correlati *se e solo se è possibile simulare l'esecuzione di un passo nel modello A attraverso un numero di passi nel modello B che sia polinomiale nella dimensione del problema in esame, e viceversa.*

Risulta immediato allora, che, dato un insieme di modelli di calcolo polinomialmente correlati, è equivalente scegliere un modello rispetto ad un altro. La seguente congettura riesce a ridurre lo studio della complessità di un problema ad un unico modello di calcolo.

> **Tesi del calcolo sequenziale**
> *Tutti i ragionevoli modelli di calcolo sono polinomialmente correlati con la MdT, e pertanto sono equivalenti rispetto a tale criterio.*

Grazie a questa affermazione, assieme alla Tesi di Church-Turing vista in precedenza, possiamo scegliere un modello di calcolo, ovviamente la MdT, e sviluppare

una teoria su quello, senza preoccuparci di come sia correlato con altri eventuali modelli.

Nei paragrafi precedenti abbiamo introdotto la MdT ed esposto i motivi che portano, ancora oggi, ad utilizzarla come strumento per la calcolabilità e la complessità computazionale. Nelle prossime pagine daremo la definizione di classe di complessità e ne esamineremo alcune delle più importanti.

1.2 Classe di complessità P

Intuitivamente, una classe di complessità può essere vista come un insieme di problemi che richiedono, utilizzando lo stesso modello di calcolo, la stessa quantità di risorse. Più precisamente, occorre stabilire in partenza alcuni paramentri fondamentali:

- il modello di calcolo di riferimento;
- la modalità con cui si eseguono le computazioni (ad esempio deterministica, specificheremo in seguito);
- una risorsa disponibile in quantità limitata;
- una funzione costo $f : \mathbb{N} \longrightarrow \mathbb{N}$ che esprima il limite di tale risorsa.

Come abbiamo accennato, in questa trattazione faremo riferimento esclusivamente alla MdT come modello di calcolo per i motivi esposti sopra (Tesi di Church-Turing). Per quanto riguarda le risorse, la scelta ricade in maniera naturale sul tempo e sullo spazio.

Inoltre, nelle seguenti definizioni si utilizza la nozione di funzione propria. Si tratta di una classe ampia e naturale di funzioni da $\mathbb{N}$ a $\mathbb{N}$ che permette di evitare alcuni fenomeni patologici (e di scarso interesse in questa sede). Informalmente:

Definizione 1.2.1. *Una funzione $f : \mathbb{N} \longrightarrow \mathbb{N}$ è propria se:*

i. $f \nearrow$, *cioè* $f(n) \leq f(n+1) \quad \forall n \in \mathbb{N}$

ii. Esiste una MdT M_f che è in grado di calcolare $f(x)$ in tempo $O(|x| + f(|x|))$ e spazio $O(f(|x|))$

Introduciamo ora le classi $TIME$ e $SPACE$.

Definizione 1.2.2. *Sia f una funzione propria. La classe di complessità $TIME(f)$ è l'insieme dei linguaggi per cui, per ogni linguaggio L, esiste una MdT che riconosce L e che utilizza, per ogni input x, al più $f(|x|)$ unità di tempo, dove $|x|$ indica la lunghezza della stringa di input x. In altri termini:*

$$TIME(f) = \{L \text{ linguaggio} \mid \exists\, M \text{ MdT che riconosce } L \text{ in tempo } f(n)\} \quad (1.1)$$

Definizione 1.2.3. *Sia f una funzione propria. La classe di complessità $SPACE(f)$ è l'insieme dei linguaggi tali per cui, per ogni linguaggio L, esiste una MdT che riconosce L ed utilizza, per ogni input x, al più $f(|x|)$ celle della macchina, dove $|x|$ indica la lunghezza della stringa data in input x. Come prima possiamo scrivere:*

$$SPACE(f) = \{L \text{ linguaggio} \mid \exists\, M \text{ MdT che riconosce } L \text{ in spazio } f(n)\} \quad (1.2)$$

Dopo tutti questi preamboli è finalmente giunto il momento di gettare le basi per uno dei concetti cardine della complessità computazionale, la classe **P**.

Definizione 1.2.4. *La classe di complessità* **P** *è formata da tutti i problemi che possono essere risolti in tempo polinomiale, ossia:*

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME(n^k) \quad (1.3)$$

In maniera del tutto analoga si può definire **P** *come la classe dei linguaggi L per i quali esistono una MdT M ed un polinomio p tali che:*

i. $L = L_M$, ovvero M accetta in input ogni $x \in L$;

ii. $\forall x \in L$, M si arresta in uno stato finale di accettazione o rifiuto dopo un numero di passi $p(|x|)$.

Grazie alla Tesi del Calcolo Sequenziale siamo in grado di dire che **P** non dipende dal modello di calcolo scelto, ovvero l'uso della MdT risulta solo un'utile convenzione e non una necessità. In questo caso si dice che la classe **P** è *robusta*.

1.3 Non determinismo e classe NP

Ora che abbiamo introdotto la classe **P** dei problemi risolvibili in tempo polinomiale, dobbiamo parlare di un'altra classe fondamentale, la cosiddetta classe **NP**. Abbiamo già tutti gli strumenti per dare una prima definizione informale, ma corretta, di questa classe di complessità. Infatti, un problema appartenente alla classe **NP** ammette una procedura di verifica efficiente. In altri termini, anche non sapendo se questo problema è risolvibile in tempo polinomiale, avendo a disposizione una possibile soluzione, siamo in grado di stabilire se questa è effettivamente giusta o sbagliata.
Per permetterci di definire con più precisione questa classe dobbiamo prima dare la nozione di *non determinismo*. In parole povere, un calcolo non deterministico si serve di un agente esterno che guida le scelte e dà suggerimenti rispetto al problema. In maniera più rigorosa diciamo:

Definizione 1.3.1. *Una computazione si dice* non deterministica *se consiste in una successione di passi discreti, ciascuno dei quali è una transizione da uno stato ad un insieme di stati, invece che ad un singolo stato.*

Questo significa che una macchina non deterministica è capace di considerare simultaneamente tutti possibili passaggi di un certo calcolo. Nella pratica, a noi interessa la definizione di una una variante della MdT, ovvero la *macchina di Turing non deterministica*.

Definizione 1.3.2. *Una macchina di Turing non deterministica (in breve $MdTN$) è definita nello stesso modo di una normale MdT, con la sola modifica che riguarda la funzione di transizione δ, che, invece di restituire una singola configurazione successiva della macchina, restituisce un insieme di possibili configurazioni successive. Per essere precisi, allora, δ non è una funzione ma una relazione tra due*

insiemi di stati:

$$\delta \subseteq \left(Q \times \Gamma^{k-1}\right) \times \left(Q \times (\Gamma \times \{S, D, F\})^{k-1}\right) \tag{1.4}$$

È possibile vedere una $MdTN$ in un modo abbastanza intuitivo. Prendiamo una normale MdT e dotiamola di una nuova componente, il *modulo di ipotesi*. Questo è dotato di una testina di sola scrittura e ha il compito di scrivere una stringa sul nastro all'inizio della computazione. Ciò può essere visto come un "suggerimento" per procedere, se possibile, nella maniera corretta fino alla soluzione del problema. Dopo aver letto l'input non deterministico, la macchina procede normalmente in modo deterministico.

Definizione 1.3.3. *Una $MdTN$ M accetta una stringa di input x se e solo se almeno una delle computazioni originate da x termina nello stato di accettazione $q_S \in Q$. Analogamente, M riconosce il linguaggio L_M se*

$$L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\} \tag{1.5}$$

dove Σ^* è l'insieme di tutte le stringhe finite di simboli dell'alfabeto Σ della macchina M.

Il tempo che una $MdTN$ impiega ad accettare una stringa $x \in L_M$ è uguale alla lunghezza del cammino più breve tra tutti quelli che terminano in uno stato di accettazione $q_S \in Q$.

Definizione 1.3.4. *Una $MdTN$ M ha un* costo in tempo *espresso dalla funzione $T : \mathbb{N} \to \mathbb{N}$ se, dato l'input $x \in L_M$, M entra in uno stato finale di accettazione dopo al più $T(|x|)$ passi.*

Dopo quanto detto sopra, la definizione di classe **NP** è simile in tutto e per tutto a quella di **P**, considerando la $MdTN$ invece della semplice MdT.

Definizione 1.3.5. *Sia f una funzione propria. La classe di complessità $NTIME(f)$ è l'insieme dei linguaggi per cui, per ogni linguaggio L, esiste una $MdTN$ che*

consuma, per ogni input x accettato dalla macchina, al più $f(|x|)$ unità di tempo.

$$NTIME(f) = \{L \text{ linguaggio} \mid \exists\, M \; MdTN \text{ che riconosce } L \text{ in tempo } f(n)\} \quad (1.6)$$

Definizione 1.3.6. *La classe* **NP** *è formata da tutti i problemi la cui verifica deve essere portata a termine in tempo polinomiale. In altri termini:*

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME(n^k) \quad (1.7)$$

Allo stesso modo della classe **P***, si può definire* **NP** *come la classe dei linguaggi L per i quali esistono una $MdTN$ M e un polinomio p tali che:*

i. $L = L_M$, ovvero M accetta in input ogni $x \in L$;

ii. $\forall x \in L$, M si arresta in uno stato finale di accettazione dopo un numero di passi $p(|x|)$.

Prima di chiederci in che modo siano in relazione queste classi è opportuno soffermarci su un'ultima classe, quella dei problemi *complementari* ai problemi **NP**, la cosiddetta classe **coNP**.

Definizione 1.3.7. *Data la classe di complessità* **NP***, sia Σ^* l'insieme finito di stringhe dell'alfabeto Σ della $MdTN$ M. Definiamo la classe di complessità* **coNP** *nel modo seguente:*

$$\mathbf{coNP} = \{\Sigma^* - L | L \in \mathbf{NP}\} \quad (1.8)$$

Osserviamo che si può ragionare allo stesso modo anche per la classe **P**, ma, mentre risulta elementare l'uguaglianza $\mathbf{P} = \mathbf{coP}$, la stessa cosa non si può dire per **NP** e **coNP**. Al contrario, secondo molti l'uguaglianza di queste due ultime classi è molto improbabile e anzi è riconosciuta dai più la congettura:

$$\mathbf{NP} \neq \mathbf{coNP} \quad (1.9)$$

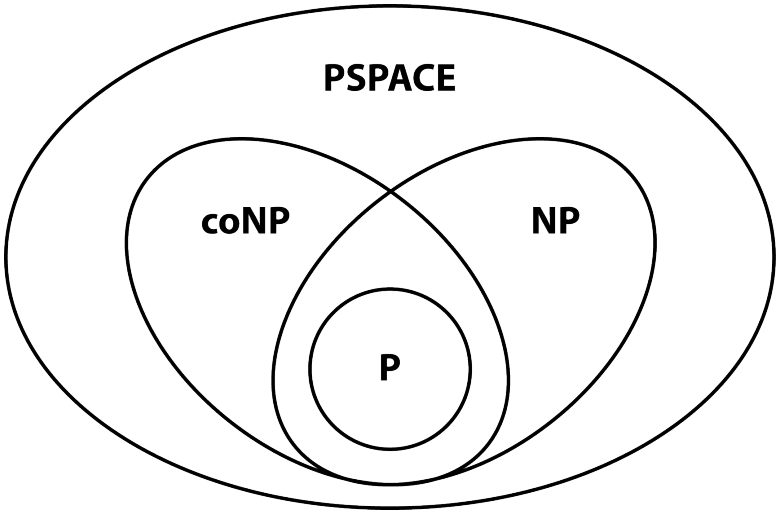

Figura 1.1: Diagramma delle classi di complessità introdotte

In realtà c'è un problema aperto ancora più dibattuto ed importante, quello che riguarda la relazione tra le classi **P** ed **NP**. Anche in questo caso la congettura ritenuta più probabile da molti esperti è proprio:

$$\mathbf{P} \neq \mathbf{NP} \tag{1.10}$$

tanto che questo quesito è uno dei celebri *Millennium Prize Problems*, i sette nuovi quesiti proposti dal *Clay Mathematics Institute*.

Capitolo 2

NP-completezza e problemi "intermedi"

Nel capitolo precedente sono state gettate le basi della complessità computazionale fino ad arrivare alle definizione delle classi **P** ed **NP**. Ora proseguiamo concentrandoci su come sia possibile classificare un problema specifico all'interno di queste introducendo la nozione fondamentale di **NP**-completezza.

2.1 La classe dei problemi NP-completi

Introduciamo dapprima l'importante definizione di riduzione.

Definizione 2.1.1 (Riduzione in tempo polinomiale)**.** *Dati i problemi L ed L' si dice che L è funzionalmente riducibile in tempo polinomiale a L' se esiste una funzione calcolabile in tempo polinomiale tale che*

$$x \in L \iff f(x) \in L' \qquad \forall x \in L \tag{2.1}$$

In tal caso la funzione f è detta trasformazione polinomiale *e si scrive $L \preceq_p L'$.*

Visto che stiamo lavorando con funzioni polinomiali, risultano banali le seguenti:

$$L \preceq_p L \tag{2.2}$$

$$L \preceq_p L' \wedge L' \preceq_p L'' \Longrightarrow L \preceq_p L'' \tag{2.3}$$

Allora, nel momento in cui abbiamo due problemi tali che

$$L \preceq_p L' \text{ e } L' \preceq_p L \tag{2.4}$$

si ha che L ed L' sono equivalenti rispetto a $\preceq_p$, e si scrive $L \equiv_p L'$, e quindi $\equiv_p$ è una relazione di equivalenza.

Anche se il concetto di riduzione è ampio e generale, a noi interessa il caso specifico delle classi **P** ed **NP** introdotte nello scorso capitolo. Le seguenti proprietà, che, in sintesi, dicono che **P** ed **NP** sono *chiuse per riduzione*, ci vengono in soccorso:

$$L \preceq_p L' \wedge L' \in \mathbf{P} \Longrightarrow L \in \mathbf{P} \tag{2.5}$$

$$L \preceq_p L' \wedge L' \in \mathbf{NP} \Longrightarrow L \in \mathbf{NP} \tag{2.6}$$

A questo punto abbiamo tutti gli strumenti necessari per definire la **NP**-completezza:

Definizione 2.1.2 (**NP**-completezza)**.** *Un problema L si dice completo per* **NP** *rispetto a $\preceq_p$ (o* **NP**-completo*) se:*

- $L \in \mathbf{NP}$
- $L' \preceq_p L \quad \forall L' \in \mathbf{NP}$

A priori nessuno ci garantisce che la classe **NPC** (problemi **NP**-completi) sia non vuota, in quanto bisogna trovare almeno un problema che vi appartenga. Il seguente teorema, dimostrato da Stephen Cook nel 1971, che per primo ha formalizzato il concetto di **NP**-completezza, ci viene incontro e stabilisce che SAT è **NP**-completo.

Teorema 2.1.1 (Teorema di Cook)**.** *Il problema* SAT *è* **NP***-completo.*

La dimostrazione di questo teorema sarebbe lunga ed al di fuori degli scopi di questa trattazione, mentre è indispensabile capire meglio di cosa stiamo parlando.

Una *formula booleana* è una formula logica formata da variabili, che possono assumere i valori TRUE e FALSE, e operatori AND, OR, NOT e (,). Una formula booleana si dice *soddisfacibile* se alle variabili possono essere assegnati dei valori per cui la formula dia come risultato TRUE.

> **SAT.**
> *Il problema di soddisfacibilità booleana (abbreviando* SAT*) determina se una certa formula booleana è soddisfacibile o meno.*

Il teorema 2.1.1 prova che la classe **NPC** sia non vuota. Non solo, grazie alla seguente proprietà delle riduzioni, simile a quelle esposte in precedenza, ci fornisce uno strumento per stabilire se un problema in **NP** sia completo.

Lemma 2.1.1. *Siano L ed L' due insiemi che appartengono a* **NP***. Se $L \in$* **NPC** *e $L \preceq_p L'$, allora $L' \in$* **NPC**

È chiara, allora, l'importanza che *SAT* ha avuto per lo sviluppo della teoria sulla **NP**-completezza: se si riesce a provare che *SAT* è riducibile in tempo polinomiale ad certo problema, allora questo, per 2.1.1, appartiene a **NPC**.

In questo modo si è riusciti a provare che molti problemi di **NP** stanno in **NPC**. Nella prossima sezione, che conclude il secondo capitolo, ci occuperemo di alcuni esempi di problemi molto importanti e stabiliremo a quali classi appartengano.

2.2 Alcuni esempi di problemi in NP

Per poter definire molti dei problemi **NP**-completi dobbiamo introdurre la nozione di grafo.

Definizione 2.2.1 (Grafo)**.** *Si dice* grafo non orientato *la coppia* $G = (V, E)$*, dove* V *è l'insieme dei vertici (o nodi) del grafo ed* E *è l'insieme degli archi che collegano due elementi di* V *(cioè* $E \subseteq V \times V$*). Dati due elementi* $u, v \in V$ *denotiamo l'arco che li unisce con* $[u, v] \in E$
Un sottografo $S = (V', E')$ *di* G *è tale se* $V' \subseteq V$ *e* $E' \subseteq E$*.*
Un grafo si dice completo *se* $\forall u, v \in V \quad \exists [u, v] \in E$*.*

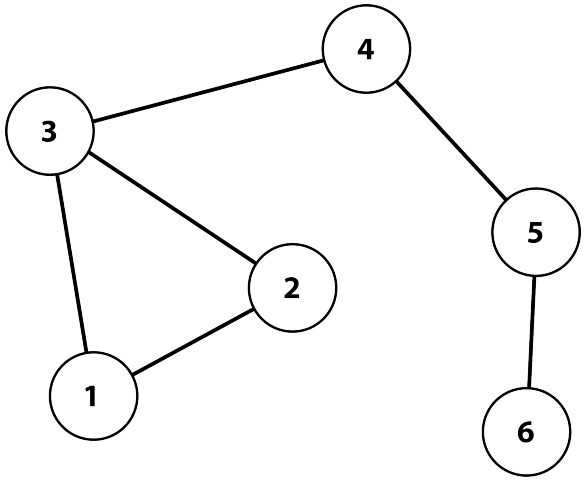

Figura 2.1: Esempio di grafo non orientato

Uno dei problemi principali della teoria dei grafi è il cosiddetto CLIQUE.

CLIQUE.
Sia G *un grafo non orientato e* k *un numero naturale.*
Il problema CLIQUE consiste nello stabilire se esiste un sottografo completo K_k *di* G *con cardinalità almeno* k*.*

Si può dimostrare che il problema CLIQUE è **NP**-completo. Per farlo bisognerebbe vedere innanzitutto che CLIQUE sta in **NP** e successivamente trovare una riduzione da un problema noto. Si vede, ad esempio, che, con un'opportuna riformulazione, SAT è polinomialmente riducibile a CLIQUE.

L'esistenza di problemi completi per **NP** depone a favore dell'ipotesi che $\mathbf{P} \neq \mathbf{NP}$. Infatti, in questo caso le classi **P** ed **NPC** sarebbero disgiunte. Se così non fosse, ovvero se $\mathbf{P} = \mathbf{NP}$, collassarebbe tutto su **P** e sarebbe garantita l'esistenza di algoritmi deterministici in costo polinomiale per tutti i problemi in **NP**. Per fare ciò sarebbe sufficiente provare l'esistenza di un algoritmo polinomiale per un solo problema ritenuto completo per **NP**. In questo modo, per definizione di **NPC**, si proverebbe che tutti i problemi **NP**-completi stanno in **P**.

Allora, nell'ipotesi $\mathbf{P} \neq \mathbf{NP}$, ha senso introdurre una classe di problemi cosiddetti *intermedi* (abbreviato in **NPI**), cioè che stanno esattamente in $\mathbf{NP} \setminus \mathbf{P}$. Questo è proprio l'enunciato del *Teorema di Ladner*.

Teorema 2.2.1 (Teorema di Ladner)**.** *Se* $\mathbf{P} \neq \mathbf{NP}$*, allora* $\mathbf{NP} \setminus \mathbf{P} \neq \varnothing$*.*

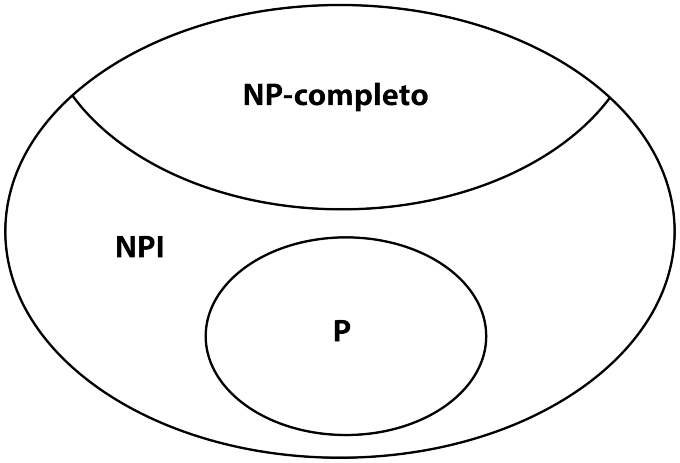

Figura 2.2: Classi di complessità nell'ipotesi $P \neq NP$

Considerando di nuovo i grafi, è possibile stabilire se questi (o parte di essi) abbiano la stessa struttura, ovvero se sono *isomorfi*.

Definizione 2.2.2 (Isomorfismo di grafi)**.** *Siano* $G = (V, E)$ *e* $G' = (V', E')$ *due grafi non orientati. Un* isomorfismo *tra questi è una funzione biettiva* $f : V \longrightarrow V'$ *tale che* $\forall u, v \in V \quad [u, v] \in V \iff [f(u), f(v)] \in V'$*.*

Un problema immediato è quello dell'isomorfismo tra grafi, che si potrebbe pensare di facile risoluzione, mentre in realtà così non è.

Isomorfismo di grafi (GI).
Siano G e G' due grafi non orientati.
Il problema GI stabilisce se esiste un isomorfismo tra i due grafi.

Finora, infatti, è stato possibile solamente provare che GI sta in **NP**, ma non si è stabilita né l'appartenenza a **P**, né la **NP**-completezza. Secondo il parere di molti esperti, questo problema è un ottimo candidato a stare in **NPI** senza possibilità di trovare un algoritmo che lo risolva in tempo polinomiale.

Questa ipotesi è sostenuta dal *Teorema di Ladner*, che, se la congettura $\mathbf{P} \neq \mathbf{NP}$ fosse vera, afferma che la classe **NPI** non può essere vuota.

Un problema vicino a GI è quello dell'isomorfismo di sottografi.

Isomorfismo di sottografi (SI).
Siano G e G' due grafi non orientati.
Risolvere SI significa rispondere alla domanda se esista un sottografo H' di G' isomorfo a G.

È ovvio come questa sia una generalizzazione del problema precedente, visto che GI prendeva in considerazione lo studio del caso in cui G sia isomorfo a G'. Sorprendentemente, anche se per quello la risposta rimane aperta, per quest'ultimo siamo in grado di provare che appartiene ai problemi completi per **NP**.

Teorema 2.2.2. $SI \in \mathbf{NPC}$

Dimostrazione. Per dimostrare il teorema è sufficiente ridurre SI ad un problema che sappiamo essere in **NPC**, nel nostro caso CLIQUE. Considerando quest'ultimo, prendiamo il sottografo K_k di cardinalità k e lo poniamo uguale ad H. Allora mostrare che esiste un sottografo H' di G è equivalente a trovare un sottografo K_k di cardinalità k in G. □

Fino a pochi anni fa, anche PRIMES, il problema di stabilire se un certo numero n sia o meno primo, faceva parte di questa classe intermedia, ma nel 2002, grazie all'algoritmo AKS, si è dimostrato che PRIMES sta in **P**.

Capitolo 3

Il problema della primalità: PRIMES

Ormai da millenni i numeri primi sono oggetto di studio e interesse da parte di innumerevoli persone. Dagli antichi uomini di scienza ai matematici moderni, la loro ricerca si è rivelata importantissima in tanti ambiti diversi, ma allo stesso tempo più complicata di quanto l'intuzione potrebbe suggerire. Basti pensare al cosiddetto *crivello di Eratostene*, un algoritmo che, dato un numero naturale, ci consente di trovare tutti i numeri primi compresi tra 2 e il numero stesso, o ancora al Teorema di Euclide che, già nel terzo secolo prima di Cristo, provava che i numeri primi sono infiniti.

Non è strano, allora, che anche in complessità lo studio dei numeri primi sia uno dei più trattati. In questo capitolo sarà introdotto il problema in maniera formale e si vedrà come sia evoluto nel corso degli anni fino ad arrivare al risultato fondamentale, conseguito solo nel 2002 dai matematici indiani Agrawal, Kayal e Saxena (abbreviato in AKS), che lo classifica nella classe di complessità **P**.

3.1 Introduzione a PRIMES

Come prima cosa scriviamo esplicitamente il problema.

PRIMES.
Sia $n \in \mathbb{N}$.
Il problema PRIMES decide se n sia un numero primo.

L'approccio più naïve per stabilire se n è primo sarebbe controllare se esistono numeri $a < n$ che dividono n. Il problema è che, così facendo, per numeri con centinaia di cifre questo metodo non è assolutamente efficiente. Anche osservando che in realtà è sufficiente controllare solo i numeri $a < \sqrt{n}$ (se $\exists\ a > \sqrt{n} : a \cdot b = n$ per un certo b, allora $b < \sqrt{n}$), la complessità risulta $O(\sqrt{n})$. Noi invece vogliamo trovare un algoritmo che si comporti come $O(\log n)$, poiché questo è proprio il numero di simboli necessari a rappresentare n in notazione posizionale[1].
È necessario, allora, utilizzare strumenti leggermente più complicati, ma molto più potenti, della teoria dei numeri. Il primo risultato, che ci servirà anche più avanti, garantisce una condizione necessaria affinché un numero sia primo.

Teorema 3.1.1 (Piccolo teorema di Fermat)**.** *Sia $p \in \mathbb{N}$ primo.*
Allora

$$\forall a \in \mathbb{Z}, \quad a^{p-1} \equiv 1 \ mod\ p. \tag{3.1}$$

Risulta equivalente la seguente formula

$$a^p \equiv a \ mod\ p \tag{3.2}$$

dove entrambi i membri sono moltiplicati per a.

Dimostrazione. Non è riduttivo supporre che $0 \leq a \leq p-1$, ovvero che $a \in \mathbb{Z}_p$. Nel caso sia $a = 0$ l'equivalenza è banalmente verificata (vedi 3.2). Allora possiamo considerare $a \in Z_p^* = \mathbb{Z}_p \setminus \{0\}$.
Consideriamo quindi il gruppo finito Z_p^* con l'operazione di moltiplicazione. Dall'algebra dei gruppi finiti sappiamo che l'insieme $\langle a \rangle$ è un sottogruppo ciclico di

[1]Qui ed per tutto il capitolo log rappresenta il logaritmo in base 2

Z_p^*. Allora, se poniamo k come ordine di a^2, il *Teorema di Lagrange* ci dice che k divide l'ordine di Z_p^*, che è $p-1$, quindi $p-1 = km$ con $m \in Z_p^*$ opportuno. Allora $a^{p-1} = a^{km} = (a^k)^m = 1^m = 1 \bmod p$. □

Quindi se un numero è primo vale la 3.1. Si potrebbe pensare di ragionare al contrario: se troviamo un numero che soddisfa questa proprietà vuol dire che è primo? Purtroppo non è così facile. Esistono certe coppie di p e a che soddisfano il 3.1.1 anche se p non è primo, ad esempio

$$a = 2,\ p = 341 = 11 \cdot 31, \quad 2^{341} = 2 \bmod 341. \tag{3.3}$$

Non solo, esistono dei p che soddisfano la 3.1 per ogni a pur essendo composti e quindi sfuggono sistematicamente ad ogni ricerca. Questi numeri vengono chiamati *numeri di Carmichael* o *pseudoprimi*.

Ricordando ora la definizione 1.3.7, possiamo affermare che PRIMES $\in$ **coNP**. Consideriamo il problema *opposto* di PRIMES

COMPOSTO.
Sia $n \in \mathbb{N}$.
Il problema COMPOSTO stabilisce se n *sia composto.*

Che il problema COMPOSTO stia in **NP** è banale, infatti un certificato di compostezza per n è dato da due numeri a, b verificando che $a \cdot b = n$. Ma se COMPOSTO $\in$ **NP**, allora PRIMO $\in$ **coNP**.

Il primo a provare che si ha anche PRIMES $\in$ **NP** fu Vaughan Pratt nel 1975. Egli affermò che

Teorema 3.1.2 (Teorema di Pratt)**.** *Sia* n *primo. Allora* n *ammette un certificato di primalità* C_n *della forma* $(n, x, p_1, \ldots, p_k, C_{p_1}, \ldots, C_{p_k})$ *con*

[2]Si dice ordine di un elemento a di un gruppo (e si scrive $o_p(n)$) il più piccolo k tale che $a^k = 1 \bmod p$

- $x \in \mathbb{N} : \ (n, x) = 1, x^{n-1} \equiv_n 1.$
- $p_1, \ldots, p_k$ *soddisfano* $n - 1 = \prod p_i$ *e* $x^{(n-1)/p_i} \not\equiv_n 1$, *per* $i = 1, \ldots, k$.
- C_{p_i} *è un certificato di primalità per ogni fattore* p_i *di* $n - 1$.

tale che $|C_n| \leq |\log p|$, *la cui correttezza si può verificare in al più* $O(\log^3 n)$ *operazioni.*

3.2 Le basi per AKS

Una delle idee più importanti nell'algoritmo AKS è anche una delle più semplici. Consideriamo il Triangolo di Tartaglia:

				1					n = 0
				1	1				n = 1
			1	**2**	1				**n = 2**
			1	**3**	**3**	1			**n = 3**
			1	4	6	4	1		n = 4
		1	**5**	**10**	**10**	**5**	1		**n = 5**
		1	6	15	20	15	6	1	n = 6
	1	**7**	**21**	**35**	**35**	**21**	**7**	1	**n = 7**
1	8	28	56	70	56	28	8	1	n = 8

Come evidenziato in figura, si nota un fatto curioso: se l'indice di riga n è primo, gli elementi di quella riga (a parte gli 1) che sono divisibili per n, mentre se n è composto questo non accade. Valendo anche il viceversa, questa proprietà del Triangolo di Tartaglia caratterizza la totalità dei numeri primi. Il problema è che anche questo test risulta inefficiente dal momento che per controllare se tutti gli elementi della riga n-esima (eccetto gli 1) sono divisibili per n dobbiamo compiere $n - 1$ divisioni [1].

Uno dei teoremi più importanti dell'articolo *Primes in* **P** è proprio una generalizzazione di questa osservazione e del teorema 3.1.1.

Teorema 3.2.1. *Siano $n \in \mathbb{N}$, $n \geq 2$ e $a \in \mathbb{Z}$ tali che $(a, n) = 1$. Allora n è primo se e solo se*

$$(X + a)^n \equiv X^n + a \ mod \ n. \tag{3.4}$$

Dimostrazione. Per $0 < i < n$ il coefficiente di x^i in $((X + a)^n - (X^n - a))$ risulta $\binom{n}{i} a^{n-i}$.
Nel caso n sia primo, si ha $\binom{n}{i} \equiv 0 \bmod n$ e quindi tutti i coefficienti sono 0.
Se n è composto, consideriamo un primo q che divida n e sia $q^k | n$. Allora q^k non divide $\binom{n}{q}$ ed è primo rispetto a a^{n-q} e quindi il coefficiente di X^q è diverso da 0 mod n. Perciò l'espressione $((X + a)^n - (X^n - a))$ non è identicamente nulla in $\mathbb{Z}_p$. □

Questo teorema ci fornisce un test per stabilire la primalità simile a quanto visto sopra: per provare che un dato n è primo basta scegliere un a e verificare che la 3.4 sia soddisfatta. Il problema è che questo richiede lo stesso tempo del Triangolo di Tartaglia, quindi non polinomiale.

L'idea centrale dell'algoritmo AKS è di velocizzare la verifica sui coefficienti binomiali riuscendo allo stesso tempo a provare che il numero scelto sia composto. Per fare questo AKS propone di lavorare non solo mod p, ma anche mod un polinomio $X^r - 1$, con r primo sufficientemente piccolo. Per esteso, deve essere soddisfatta la seguente

$$(X + a)^n = X^n + a \bmod X^r - 1, n \tag{3.5}$$

Nel calcolare entrambi i membri mod $X^r - 1$ si riduce il numero di verifiche a $r + 1$ invece di $n + 1$, quindi con un r opportuno è possibile andare a valutare effettivamente i resti.

Il problema, però, è che le formule 3.4 e 3.5 non sono equivalenti: mentre se n è primo, allora anche 3.5 ci dice che è primo per ogni valore di a ed r, è possibile che certi n composti soddisfino l'equazione per qualche valore di a ed r. Per fortuna, è possibile ristabilire la caratterizzazione con la scelta di *opportuni* valori di r. AKS studia il modo per scegliere questi r, verificando che il tempo di calcolo rimanga limitato polinomialmente.

3.3 L'algoritmo polinomiale

Andiamo ora a scrivere l'algoritmo AKS, ricordando che $o_r(n)$ indica l'ordine di a e $\varphi(n)$ è la *Funzione di Eulero*, che restituisce il numero di interi compresi tra 1 ed n che sono coprimi con n.

Input: numero naturale $n > 1$
Risultato: stabilire se n è primo o composto

1. Se ($n = a^b$ con $a \in \mathbb{N}$ e $b > 1$), restituisci COMPOSTO.
2. Trova il più piccolo r tale che $o_r(n) > log^2 n$.
3. Se $1 < (a, n) < n$ per qualche $a \leq r$, restituisci COMPOSTO.
4. Se $n \leq r$, restituisci PRIMO.
5. Da $a = 1$ a $\lfloor \sqrt{\varphi(r)} \log n \rfloor$ esegui
   se $((X + a)^n \neq X^n + a \bmod X^r - 1, n)$, restituisci COMPOSTO.
6. Restituisci PRIMO.

Teorema 3.3.1 (Teorema AKS)**.** *L'algoritmo scritto sopra restituisce* PRIMO *se e solo se n è un numero primo.*

Dimostrazione (Condizione sufficiente). Vogliamo dimostrare che se n è primo, allora l'algoritmo restituisce PRIMO. Ma se n è primo, i passaggi 1 e 3 non daranno mai come risultato COMPOSTO. Allo stesso modo, per il 3.2.1, anche il ciclo nel

passaggio 5 non potrà mai restituire COMPOSTO. Quindi l'algoritmo restituirà PRIMO o al punto 4 o al punto 6. □

La parte difficile è, ovviamente, il contrario. In quest'ultima parte ci occuperemo di dimostrare proprio la condizione necessaria.

Come prima cosa possiamo affermare che se l'algoritmo ritorna PRIMO in 4, allora n *deve* essere primo, altrimenti al passaggio 3 si sarebbe trovato un divisore di n. Nei paragrafi successivi studieremo, quindi, la correttezza del passaggio 6. I passaggi chiave sono il 2 e il 5: il 2 trova un r appropriato ed il 5 verifica l'equazione 3.5 per un certo numero di a.

Il seguente teorema fissa un maggiorante per r.

Teorema 3.3.2. *Esiste un* $r \leq \max\{3, \log^5 n\}$ *tale che* $o_r(n) > \log^2 n$.

(Per la dimostrazione vedere *Primes in* $\boldsymbol{P}$*, p. 4*).

Dal momento che $o_r(n) > 1$, deve esistere un numero primo p che divida n tale che $o_r(p) > 1$. Inoltre si ha anche che $p > r$, altrimenti uno tra i passaggi 3 e 4 stabilirebbe la primalità di n. Poiché $(n, r) = 1$, si ha $p, n \in Z_r^*$. Per i prossimi paragrafi consideriamo p ed r fissati e poniamo $l = |\varphi(r) \log n|$.

Per definire il secondo gruppo, invece, ci dobbiamo servire dei polinomi ciclotomici su campi finiti [5].

Il passaggio 5 dell'algoritmo verifica l equazioni e poiché non viene restituito COMPOSTO deve essere:

$$(X + a)^n = X^n + a \bmod X^r - 1, n \qquad \forall a : \ 0 \leq a \leq l.$$

Ciò implica:

$$(X + a)^n = X^n + a \bmod X^r - 1, p \qquad \forall a : \ 0 \leq a \leq l. \tag{3.6}$$

Dal teorema 3.2.1 abbiamo che:

$$(X + a)^p = X^p + a \bmod X^r - 1, p \qquad \forall a : \ 0 \leq a \leq l \tag{3.7}$$

e dalle equazioni 3 e 4 si ha

$$(X+a)^{\frac{n}{p}} = X^{\frac{n}{p}} + a \bmod X^r - 1, p \qquad \forall a: \ 0 \leq a \leq l. \tag{3.8}$$

Come possiamo vedere da 3.6 e 3.8, n ed $\dfrac{n}{p}$ si comportano come il numero primo p nelle precedenti equazioni. Questa proprietà prende il nome di *introspezione*.

Definizione 3.3.1. *Siano $m \in \mathbb{N}$ ed $f(X)$ un polinomio. Si dice che m è introspettivo per $f(X)$ se*

$$[f(X)]^m = f(X^m) \bmod X^r - 1, p \tag{3.9}$$

Si vede chiaramente dalle equazioni 3.7 e 3.8 che sia p sia $\dfrac{n}{p}$ sono introspettivi per $X + a$ con $0 \leq a \leq l$.

Il prossimo lemma mostra due importanti proprietà dei numeri introspettivi.

Lemma 3.3.1.

- *Se m ed m' sono numeri introspettivi per $f(X)$, allora lo è anche $m \cdot m'$.*
- *Se m è introspettivo per $f(X)$ e per $g(X)$, allora è introspettivo anche per $f(X) \cdot g(X)$.*

Una conseguenza diretta di questo lemma è che ogni numero dell'insieme

$$I = \left\{ \left(\frac{n}{p}\right)^i \cdot p^j \;\middle|\; i, j \geq 0 \right\}$$

è introspettivo per ogni polinomio nell'insieme

$$P = \left\{ \prod_{a=0}^{l} (X+a)^{e_a} \;\middle|\; e_a \geq 0 \right\}.$$

A questo punto abbiamo tutti gli strumenti per definire i due gruppi chiave per la dimostrazione. Il primo è l'insieme di tutti i residui modulo p dei numeri di I. Dato che abbiamo già visto che $(n, r) = (p, r) = 1$, si ha che questo gruppo, che

chiameremo G con cardinalità $|G| = t$, è sottogruppo di Z_r^*. Da quanto visto in 3.3.2, dato che G è generato da n e p modulo r, $t \geq \log^2 n$.

Per definire il secondo gruppo, invece, ci serviremo dei *gruppi ciclotomici su campi finiti*. Sia $Q_r(X)$ l'r-esimo polinomio ciclotomico su $\mathbb{F}_p$. Si dimostra che il polinomio $Q_r(X)$ divide $X^r - 1$ e spezza in fattori irriducibili di grado $o_r(p)$. Sia allora $h(X)$ uno di questi fattori. Poiché $o_r(p) > 1$, il grado di $h(X)$ sarà maggiore di 1. Il secondo gruppo (che denoteremo con H) è proprio l'insieme dei residui dei polinomi di P modulo $h(X)$ e p. Questo gruppo è generato dagli elementi $X, X+1, X+2, \ldots, X+l$ nel campo $F = F_p[X]/h(X)$ ed è un sottogruppo moltiplicativo di F.

A questo punto AKS riesce a trovare, grazie a due lemmi, un *lower bound* (minorante) ed un *upper bound* (maggiorante), quest'ultimo nell'ipotesi che n non sia una potenza di p, per la cardinalità di H:

$$\binom{t+l}{t-1} \;\leq\; |H| \;\leq\; n^{\sqrt{t}}$$

A questo punto, grazie a queste stime, è possibile concludere la dimostrazione di 3.3.1.

Dimostrazione (Condizione necessaria). Vogliamo provare che, se l'algoritmo restituisce il valore PRIMO, allora n è primo. Avendo posto $t = |H|$ ed $l = |\sqrt{\varphi(r)}\log n|$, la maggiorazione di prima ci dice:

$$\begin{aligned}
|H| &\geq \binom{t+l}{t-1} \\
&\geq \binom{l+1+|\sqrt{t}\log n|}{|\sqrt{t}\log n|} \text{ (poiché } t > \sqrt{t}\log n) \\
&\geq \binom{2|\sqrt{t}\log n|+1}{|\sqrt{t}\log n|} \text{ (poiché } l = |\sqrt{\varphi(r)}\log n| \geq |\sqrt{t}\log n|) \\
&> 2^{|\sqrt{t}\log n|+1} \text{ (poiché } |\sqrt{t}\log n| > |\log^2 n| \geq 1) \\
&\geq n^{\sqrt{t}}.
\end{aligned}$$

Grazie alla minorazione, invece, $|H| \leq n^{\sqrt{t}}$ se n non è una potenza di p. Quindi,

$n = p^k$ per qualche $k > 0$. Se $k > 1$, allora l'algoritmo restituirà COMPOSTO nel passaggio 1. Quindi $n = p$. □

A questo punto la dimostrazione della correttezza dell'algoritmo è completa. Nell'articolo *Primes in* $\boldsymbol{P}$ segue la dimostrazione della sua polinomialità, che sarà tralasciata per ragioni di sintesi. In particolare, il risultato finale è

Teorema 3.3.3. *La complessità asintotica sulla risorsa tempo dell'algoritmo AKS è* $O(\log^{15/2} n)$.

Abbiamo dimostrato, allora, che l'algoritmo AKS è sia corretto sia eseguibile in tempo polinomiale.

Bibliografia

[1] S. Aaronson (2003). *The Prime Facts: From Euclid to AKS*. `http://www.scottaaronson.com/writings/prime.pdf`

[2] M. Agrawal, N. Kayal, N. Saxena (2002). *PRIMES in* $\boldsymbol{P}$.

[3] A. Bernasconi, B. Codenotti (1998). *Introduzione alla complessità computazionale*. Springer, Milano, Italia.

[4] A. Bernasconi, B. Codenotti, G. Resta (1999). *Metodi matematici in complessità computazionale*. Springer, Milano, Italia.

[5] Wikipedia. *Cyclotomic polynomial*. `http://en.wikipedia.org/wiki/Cyclotomic_polynomial`