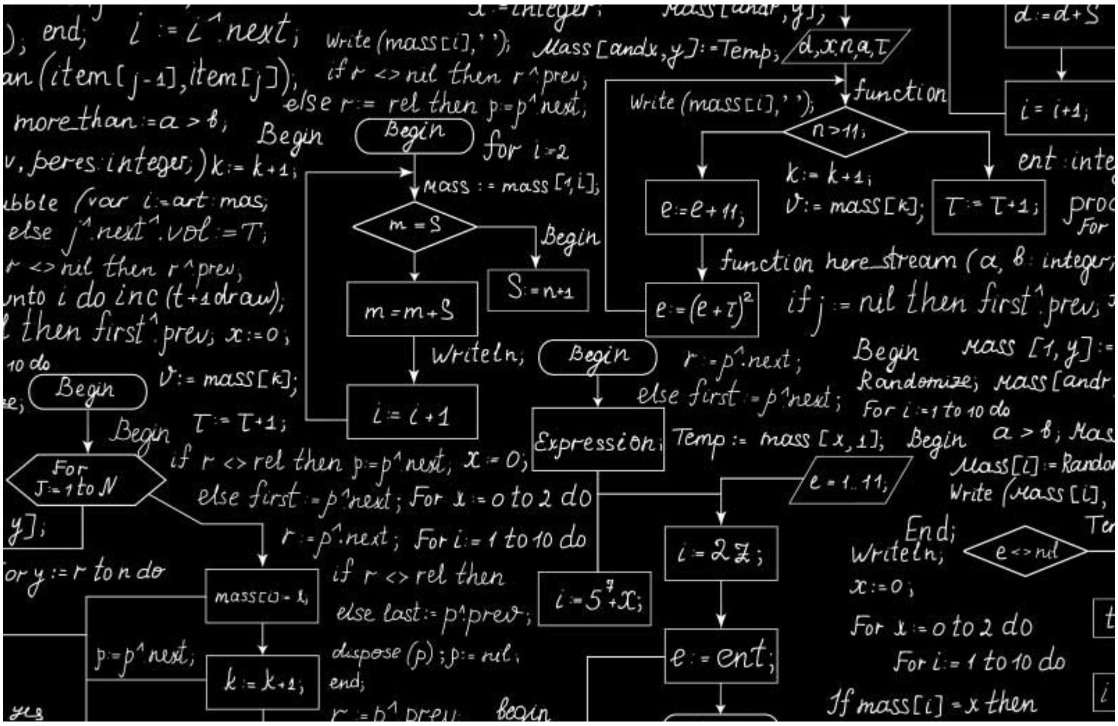
UNIVERSITÀ DEGLI STUDI DI TRIESTE

Enrico Lacchin

Enrico Lacchin

Algoritmi e Strutture dati

Appunti



Materia: Algoritmi e strutture dati
Docente: « Nome Docente »

Indice

1 Algoritmo

Un algoritmo è un procedimento di calcolo teoricamente meccanizzabile. Rappresenta la *sequenza di passi computazionali che prende in ingresso un dato insieme di valori e produce un insieme di valori in uscita.*
Un algoritmo viene usato come strumento per risolvere un ben definito problema computazionale.
Un algoritmo può essere:

- **Corretto**: termina con insieme corretto di valori in uscita per ogni istanza del problema
- **Incorretto**: termina o con un insieme non corretto di valori in uscita per alcune istanze del problema oppure non termina.

Ogni algoritmo ha una propria **efficienza** ossia il numero di operazioni richieste per produrre dati in uscita (considerando quelli in entrata) → con dimensioni notevoli la differenza di efficienza è marcata.
Le **proprietà** di un'algoritmo sono:

- Non ambiguità
- Finitezza
- Terminazione
- Effettività
- Atomicità (operazioni minime)

Un **array** è una lista ad accesso diretto (non sequenziale) casuale (dove vogliamo).
L'**istanza** di un problema è l'insieme dato di valori in ingresso.
La **struttura dati** è la modalità di memorizzazione e organizzazione di dati per garantire un miglior accesso e manipolazione. Una struttura dati è la *heap* la quale organizza array per verificare la proprietà detta priorità (alternativa: albero binario quasi completo).

2 Analisi

L'analisi va a stimare le risorse richieste in memoria, la banda e il tempo di calcolo.
Il **modello del calcolatore** da usare è:

- Singolo processore (Single Core)
- Memoria ad accesso casuale
- No gerarchie di memoria

Le **assunzioni** sono istruzioni varie con periodo costante.
Il **tempo di esecuzione** è la somma dei tempi di esecuzione (costanti per ipotesi) di ciascuna linea dello pseudocodice moltiplicata per il numero di esecuzione della stessa → espresso come dimensione dei valori in ingresso.
Tipologie di assunzioni:

- Ordinamento: lunghezza della sequenza da ordinare → divide et impera come metodo di risoluzione efficiente
  - suddivisione del problema in sottoproblemi più semplici
  - risoluzione ricorsiva dei sottoproblemi
  - combinazione delle soluzioni per soluzione del problema
- Operazioni sui grafi: numero di vertici e archi presenti

3 Complessità

3.1 Classi di complessità

$$\begin{array}{lll} f(n) = O(g(n)) & \text{equivale a} & a \leq b \\ f(n) = \Omega(g(n)) & \text{equivale a} & a \geq b \\ f(n) = \Theta(g(n)) & \text{equivale a} & a = b \\ f(n) = o(g(n)) & \text{equivale a} & a < b \\ f(n) = \omega(g(n)) & \text{equivale a} & a > b \end{array}$$

Caso peggiore (worst case): Si svolgono tutte le iterazioni possibili all'interno dell'algoritmo
Caso medio (medium case): É a metà tra il caso migliore e il caso peggiore
Caso migliore (better case): Si svolgono il minor numero di iterazioni possibili all'interno dell'algoritmo

Esempi:

Algoritmo	Tempo di esecuzione nel caso peggiore	Tempo di esecuzione atteso/nel caso medio
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \cdot log(n))$	$\Theta(n \cdot log(n))$
Heapsort	$O(n \cdot log(n))$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot log(n))$ (atteso)
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (caso medio)

4 Grafo

Un grafo è una configurazione formata da un insieme di punti e linee con unione di coppie di nodi → definita relazione qualsiasi tipo.
Un grafo può essere:

- **Semplice**: coppie di vertici non ordinate (simmetria) e distinte
  - Nodi o Vertici (v): singoli punti → senza figli: foglie
  - Archi/Edge (E): linee
  - Grado/Degree: numero di archi che partono dal vertice
  - Memoria: Theta di $(v + E)$
  - Complessità lista di adiacenza: $\Theta 2(|V| + |E|)$
    matrice di adiacenza: $\Theta(|V|^2)$
- **Multigrafo**: stesso arco può comparire più volte
  - Molteplicità dell'arco: numero di volte che si ripete
  - Cappi o Loops: contribuisce due volte al grado
  - Complessità: $\Theta(|V|^2)$

5 Ricorrenza

La ricorrenza è un equazione o disequazione per descrivere la funzione in termini del suo valore sugli ingressi con dimensione inferiore.
Il tempo di esecuzione viene spesso espresso come ricorrenza
Metodi di risoluzione:

- Sostituzione
  - fare una previsione della soluzione
  - verificare l'ipotesi tramite induzione
- Albero della ricorrenza: si crea un albero ricorsivo (generare buone soluzioni di tentativi)
- Esperto: basata sul teorema dell'esperto (algoritmi ricorsivi)

5.1 Master Theorem (per gli algoritmi ricorsivi)

Siano $a \geq 1,\ b > 1,\ T(n) = aT\left(\frac{n}{b}\right) + f(n),\ \varphi(n) = n^{log_b(a)+\ldots\ \epsilon>0}$
Se

i. $$\lim_{n\to+\infty} \frac{f(n)}{\varphi(n)} = 0 \longrightarrow f(n) \in O(n^{log_b(a)-\epsilon})$$

ii. $$\lim_{n\to+\infty} \frac{f(n)}{\varphi(n)} = l \in \mathbb{R} \longrightarrow f(n) \in \Theta(n^{log_b(a)})$$

iii. $$\lim_{n\to+\infty} \frac{f(n)}{\varphi(n)} = +\infty \longrightarrow f(n) \in \Omega\left(n^{log_b(a)+\epsilon}\right) \wedge af(\frac{n}{b}) < cf(n)\ per\ qualche\ c < 1$$

Allora

i. $T(n) \in \Theta(n^{lob_b(a)}) \to T(n) \in \Theta(\varphi(n))$

ii. $T(n) \in \Theta\left(n^{lob_b(a)} \cdot log(n)\right) \to T(n) \in \Theta(\varphi(n) \cdot log(n))$

iii. $T(n) \in \Theta(f(n)) \to T(n) \in \Theta(f(n))$

6 Struttura dati

Array: struttura dati ad accesso libero → staticità (bisogna sapere n a priori)
List/Lista concatenata: ad accesso sequenziale. L'*insert* è sempre possibile mentre il *delete* solo se il contenuto ≥ 1 → dinamicità.
L'inizio viene definito come head e la fine come tail.
Ci sono due tipologie:

- FIFO (First In First Out): elementi aggiunti da una parte e estratti dall'altra → code (queue)
- LIFO (Last In First Out): elementi aggiunti e tolti dalla stessa parte → pile (stack)

Un **record** è la raccorta di dati o numeri → key: valore da ordinare
altri valori: dati satelliti

7 Algoritmi di ordinamento

7.1 Bubble Sort

Ogni coppia di elementi adiacenti comparata se, non in ordine, vengono invertiti.
Complessità: $\Theta(n)$ nel caso migliore $\rightarrow$ controlli inutili se gli elementi sono già in ordine.

7.2 Exhaustive Search

Va a verificare ogni soluzione teoricamente possibile fino a quella corretta.
Risulta essere estremamente lento ma corretto e va sfruttato solo con pochissimi elementi.
Complessità: $\Theta(2^n)$

7.3 Heapsort

L'algoritmo va a ordinare dall'alto verso il basso.
L'**altezza di un nodo** è la lunghezza degli archi del percorso più semplice verso una foglia. L'**altezza heap** rappresenta l'altezza della radice.
La dimensione del problema varia a seconda del tipo di problema:

- Originario: n
- Sottoalbero con più elementi: m
- Altezza del nodo oggetto del problema originario: h

7.4 Mergesort

L'algoritmo va a suddividere n elementi in sottosequenze di $\frac{n}{2}$ elementi della stessa dimensione (approccio divide et impera).
Non fa ordinamento in loco, fa un ordinamento ricorsivo delle sottosequenze fino a una sottosequenza di lunghezza 1 (ordinata).
Utilizza le funzioni

- Merge: combinazione di due sottosequenze ordinate per produrre intera sequenza ordinata $\rightarrow$ Complessità: $\Theta(n)$
- Mergesort: ordinamento di una sequenza suddividendola in due, ogni sottosequenza ordinata (con ricorsione) tramite merge

8 Ordinamento per confronti

Teorema : Qualsiasi algoritmo per confronti ha $\Omega(n \cdot log(n))$ nel caso peggiore

8.1 Bucket Sort

É un algoritmo molto frequente. Ipotesi di elementi distribuiti uniformemente $\rightarrow$ n elementi da ordinare.
Procedimento:

- intervallo $[0, 1)$ diviso in n intervalli/contenitori (bucket) di lunghezza uguale
- ciascun valore dell'array inserito nel bucket a cui appartiene
- valori all'interno del bucket ordinati
- concatenazione dei valori contenuti nei bucket con insertion sort

8.2 Insertion Sort

L'algoritmo va a considerare un valore alla volta e lo si mette nella posizione corretta tra i valori che lo precedono (approccio incrementale).
Fa un ordinamento in loco (sul posto). Valori ordinati dell'array con al più un numero costante di essi all'esterno.
É invariante cioè per ciascuna iterazione, gli elementi originali rimangono gli stessi, ma in ordine crescente

- Vero prima del ciclo
- Se vero per prima iterazione, vero per l'iterazione successiva
- Prova che algoritmo corretto quando il ciclo termina

Complessità: $\Theta(n)$ nel caso migliore (better sort)

8.3 Counting Sort (tre indici)

Non viene fatto un ordinamento in loco. I numeri da ordinare sono compresi tra 0 e k con $k \ll n$ (k molto minore di n).
Procedimento:

- Due array, con con n elementi (A) e uno vuoto (B) $\rightarrow$ A parte da 1 ($A = 1, 0, 0, 2, 2, 1$)
- Altro array riempito tutto di zeri (C) $\rightarrow$ la prima casella parte da 0

- Si considerano gli elementi di A $\rightarrow$ caselle di C corrispondenti agli elementi di A con numeri in cui compaiono $(C = 2, 2, 2)$
- Si sommano le celle di C con quelle precedenti $(C = 2, 2, 2 \rightarrow C = 2, 4, 6)$
- Array B di lunghezza n come A $\rightarrow$ la prima casella parte da 1
- Controllo di A dall'ultima alla prima casella
- Si confrontano i numeri anche tramite il numero della casella $\rightarrow$ si scala C dopo aver messo il numero di B

Diventa disastroso con n^2. É un algoritmo corretto e stabile (numeri con stesso valore nell'array nello stesso ordine sia in input che in output).
Complessità dei cicli for: $\Theta(k), (n), (k), (n) \rightarrow$ totale: $\Theta(k + n)$

9 Algoritmi di vista

Algoritmi per cubo $n - dimensionale$

- Ogni vertice con n archi
- Anche con loops e molteplicità, complessità invariata

9.1 Teorema (Handshake Theorem)

$$\sum_{v \in V} deg(v) = 2|\xi| \Rightarrow \text{La somma dei gradi è sempre pari}$$

Nota: vero anche perché i loops valgono 2

10 Visita di un grafo

La visita di un grafo significa esaminare una volta tutti i nodi del grafo.
Le difficoltà possono essere due:

- Cicli (marcare nodi visitati)
- Nodi isolati: la visita termina dopo aver considerato tutte le componenti isolate di un grafo

10.1 In ampiezza (BFS, Bread-First-Search)

Conta la distanza (numero minimo di archi) dal vertice sorgente agli altri punti (se la distanza è infinita, non ci sono cammini disponibili).
Per capire se i vertici sono scoperti o meno si utilizza una classificazione per colori:

- Bianco (W): inizialmente tutti bianchi
- Grigio (G): incontrato per la prima volta
- Nero (B): tutti adiacenti di un nodo grigio visitati → un nodo nero ha solo adiacenti grigi

Grafo dei predecessori: albero Complessità: $O(V + E)$, se denso $O(E)$

10.2 In profondità (DFS, Depth-First-Search)

Procedimento:

- Si esplorano i nodi degli archi uscenti
- Backtrack: nodi degli archi uscenti esplorati, si torna indietro per esplorare quelli del predecessore
- Tutti i vertici raggiunti da una sorgente iniziale → se alcuni vertici sono inesplorati, uno viene scelto come nuova sorgente e riparte la ricerca

Grafo dei predecessori: foresta

10.3 Topological Sort (DAG, Direct Acyclic Graph)

Viene fatto un ordinamento lineare di tutti i vertici di un grafo orientato senza cicli → i vertici si definiscono ordinati se ogni nodo viene prima dei nodi collegati ai suoi archi uscenti.
Non vi è um ordinamento notale → la soluzione non è necessariamente unica
Procedimento:

- Si considera una situazione

- Si parte da un punto arbitrario
- Si segue la direzione degli archi uscenti
- Finisti gli archi uscenti, si torna al punto considerato
- Si ripete dal secondo punto, finché non rimangono più punti
- Metterli in fila, considerandoli al contrario rispetto a come sono stati ordinati (numero più grande)
- Ordine rispettato

11 Altri algoritmi

11.1 Algoritmo di Dijkstra

L'algoritmo presenza un costo per arco = grafi pesanti mediante numeri → necessità di archi non negativi (altrimenti Bellman - Ford)
Percorso con costo minimo dal vertice sorgente agli altri. Non serve molteplicità degli archi → viene eliminato quello dal costo maggiore.
Procedimento:

- Si prende in considerazione un grafo arbitrario
- Si considera un vertice come sorgente
- Si prendono singolarmente i costi delle distanze dal vertice sorgente agli altri → si considerano anche i punti per cui si deve per forza passare (se dal vertice sorgente v_2, bisogna passare per v_1, si considera anche v_1)
- Si eliminano i percorsi più costosi, via via che si trovano
- L'unico che rimane è il percorso meno costoso

11.2 Algoritmo di Euclide

L'algoritmo trova il MCD tra due numeri, se ricorsivo richiama se stesso.

12 Distanza di edit

Si parte da una stringa e la si trasforma in un'altra → trasformazioni di costo 1 (tranne gratis che non costa nulla)

C	Cancellazione	$\downarrow$
I	Inserimento	$\rightarrow$
S	Sostituzione	$\searrow$
G	Riporto gratis	$\searrow$

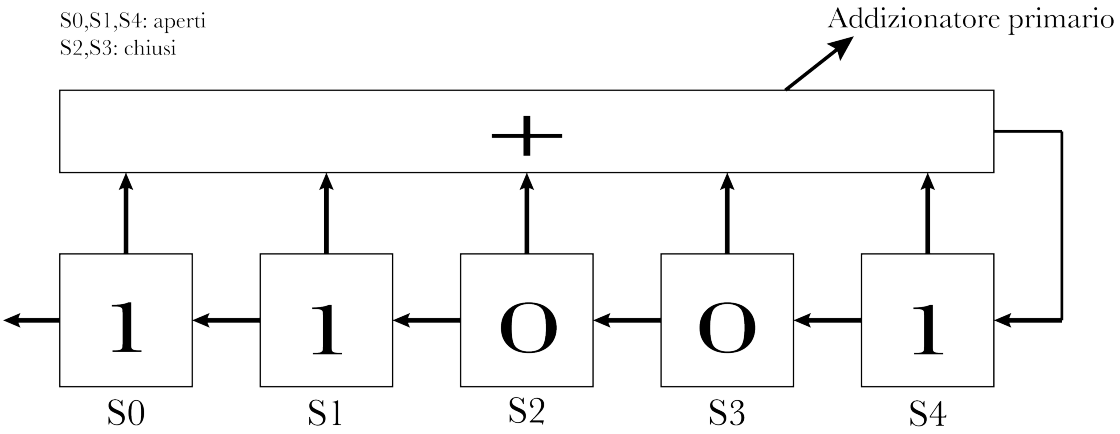
$I \leq C \leq S$

Procedimento:

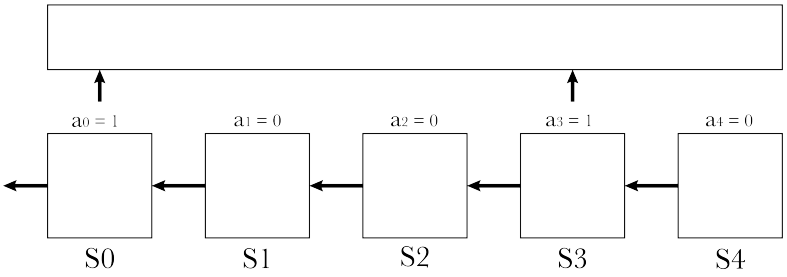
- Si crea una matrice → in alto la parola a cui si vuole arrivare, a sinistra la parola da sui si parte
- Si tiene uno spazio vuoto da un puntino
- Si procede a costruire la matrice con numeri
- Si considera il percorso meno costoso
- Si fanno le varie trasformazioni e si considerano al contrario

Complessità: $\Theta(n^2)$

13 Generazione di Bit pseudocasuali



$$p(x) \hat{=} a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + x^n$$



$$p(x) = 1 + x^3 + x^5$$

Registro a scorrimento (verso sinistra) → considerati gli spazi dove c'è 1
Seme: Combinazione casuale di 0 e 1 → proibito: $0 \ldots 0$ dato che la somma fa sempre 0
Produzione ciclica di periodo massimo: $2^n - 1$ $n-ple$ binarie con accesso "casuale" (0 uscenti uguali e 1 entranti)

- n: lunghezza del registro
- 1: seme proibito
  → funzione di autocorrelazione: bilanciamento del numero di $n-ple$ che finiscono con $00, 01, 10, 11$ [$(n-2)-ple$ binarie per ciascuno]

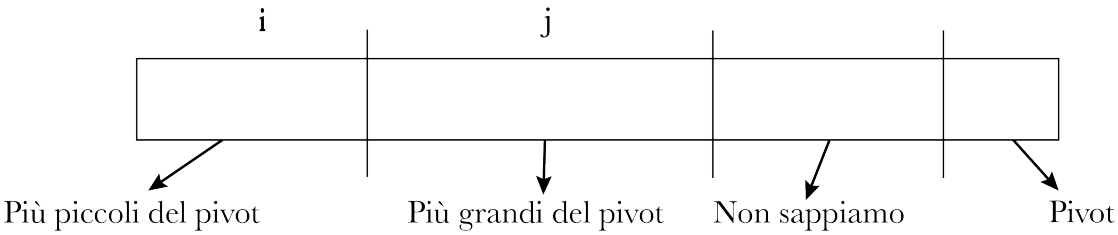
Shift register circa potenziatore di casualità (data del seme) → input determinato

13.1 Heapify

Il suo scopo è quello di riorganizzare heap per mantenere la proprietà della priorità.
Albero: grafo non diretto, connesso e aciclico → grafo di un nodo: numero di sottoalberi del nodo uguale al numero di figli del nodo

Altezza o profondità dell'albero: distanza radice - foglia
Percorrenza: dalla radice verso le foglie (dall'alto verso il basso)
$|E| = |V| - 1$
Complessità: $\Theta(n)$

13.2 Quicksort



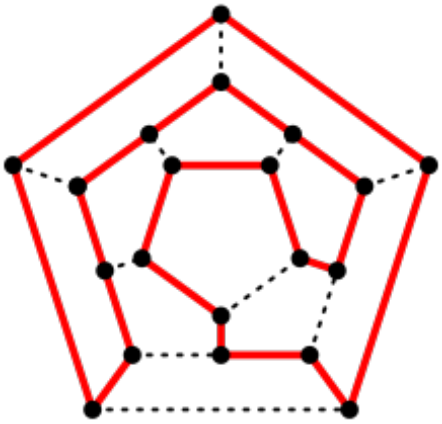
i (wall): non è detto che si sposti
j (current element): si sposta di una posizione alla volta
Pivot: elemento considerato
Viene utilizzato l'approccio del divide et impera.
Partition (prima parte di quicksort)

- Si fissa il pivot
- Si inizializza il muro

Quicksort → si mettono i valori come nell'immagine
Randompartition → il primo dei più grandi e pivot scambiati

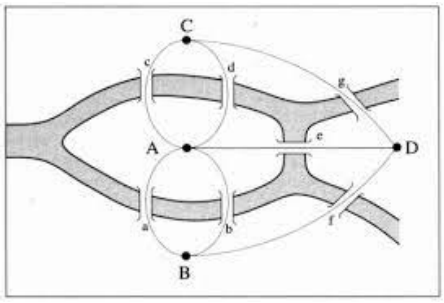
14 Problemi famosi

14.1 Hamilton



Icosaedro: Qualsiasi poliedro a 20 facce
Nel solido platonico le facce sono separate da spigoli (edge).
Si può, in due dimensioni, toccare tutti i vertici una sola volta (closed path)? La risposta è NO, non esiste un algoritmo che risolva il problema ($P \neq NP$)

14.2 Eulero



Königsberg è una città collegata da 7 ponti.
Problema: Fare una passeggiata passando per ciascun ponte una sola volta
Soluzione: Non è possibile, dato che il numero dei gradi di ciascuna zona è dispari.

14.3 Sacco o Knapsack Problem

Il problema di Knapsack è un problema di ottimizzazione.
Il ladro che ruba deve prendere in considerazione:

- H: altezza del sacco/zaino

- h: altezza dell'oggetto che vuole rubare
- v: valore dell'oggetto che vuole rubare

Greedy: il ladro ruba fino a quando ha spazio, anche se potrebbe prendere oggetti senza valore.
Complessità: $\Theta(n^2)$ nel caso peggiore
Greedy con preordinamento: Il ladro mette all'inizio gli oggetti più preziosi e quelli vengono rubati.
Complessità: $\Theta(n^2)$ nel caso peggiore

15 Altro

15.1 LCS

$$X =< x_1, \ldots, x_m > \quad e \quad Y =< y_1, \ldots, y_n >$$

1. Caratterizzazione: 2^m sottosequenze di $X \rightarrow$ tempo esponenziale (inutilizzabile con lunghe sequenze)
2. Soluzione ricorsiva
3. Calcolo della lunghezza con il metodo bottom-up
   $\Theta(m \cdot n)$ sottoproblemi

15.2 Problemi

- **P**: risolvibili in un tempo polinomiale $\rightarrow$ nel caso peggiore $O(n^k)$ con k costante
- **NP**: non risolvibili in un tempo polinomiale (e.g. problema di Hamilton) $\rightarrow$ C / completo: particolarmente difficili da risolvere. Se si può risolvere un elemento di NPC in tempo polinomiale, lo si può fare con qualsiasi elemento (ancora non trovato)

Si ipotizza $P \neq NP$ oppure $P \subset NP$, ma non dimostrato