**Appunti PA (progettazione di algoritmi)**

Un algoritmo è un procedimento effettivo che, tramite (tanti) passi elementali (operazioni matematiche o logiche semplici) risolve il problema dato in un tempo finito. “Algoritmo” deriva dal matematico Muhammad ibn musa al-khwarizmi, che a differenza dei matematici del suo tempo (780-850) si interessava dell’effettualità e dell’applicazione della matematica alla vita reale (è la stessa persona che ha dato nome all’algebra)

Tra gli algoritmi che hanno *cambiato il mondo* negli ultimi anni nell’informatica possiamo ricordare:

1. PageRank di Google (ricerca WEB)
2. Algoritmi crittografici a chiave pubblica
3. Algoritmi per la trasmissione affidabile dei dati
4. Algoritmi per la compressione dati
5. Algoritmi di apprendimento automatico

L’obiettivo dell’informatica è la formalizzazione del problema per proporre una soluzione, conoscendo e basandosi su algoritmi esistenti per la creazione di nuovi algoritmi più efficienti

Verranno usati dei Meta-Algoritmi per la formulazione di un algoritmo solutivo per un problema

**ESERCIZIO 1**

Dato un array di numeri interi (positivi o negativi) e sia V(i, j) la somma tra gli indici i e j, e V(a) la somma massima tra gli indici

Input= sequenza a di numeri

Output= massimo della somma tra gli indici i e j

Una possibile soluzione è testare qualsiasi somma tra tutte le possibili combinazioni di indici (0 <= i <= j <= n-1), la complessità di questo algoritmo però è inefficiente e viene evidenziato dall’analisi dell’algoritmo che dimostra che nel caso peggiore si avranno n3 operazioni elementari -> O(n3)

Una raffinazione di questo algoritmo dimostra come è possibile eliminare un for innestato, in quanto la somma tra gli indici i e j veniva calcolata ogni volta quando avrebbe potuto essere conservata dal calcolo precedente. Riducendo la complessità computazionale ad O(n2)

Applicando la tecnica divide et impera

Dividiamo l’array in due sezioni, e risolviamo in ognuna delle due sezioni la miglior somma tra tutte le combinazioni degli indici, verifichiamo poi anche indici compresi in entrambe le suddivisioni. Nel primo caso cerchiamo ricorsivamente nelle due metà dell’array, mentre nel secondo caso partendo dalla metà sommiamo, fino ad arrivare all’inizio o alla fine dell’array, ogni elemento alla somma precedente (formata dalla somma dal centro fino all’indice precedente)

L’analisi di questo algoritmo dimostra che:

sia T(n) il numero di operazioni dell’algoritmo, abbiamo che nella chiamata ricorsiva dell’algoritmo, la complessità sarà T(n/2). dato che le chiamate ricorsive sono 2 avremo 2T(n/2) mentre la ricerca a cavallo tra la metà dell’array richiede (al più) 2\*(n/2) quindi O(n). L’equazione di ricorrenza risultante sarà:

T(n)= 2T(n/2)+O(n). Ovvero O(n log n)

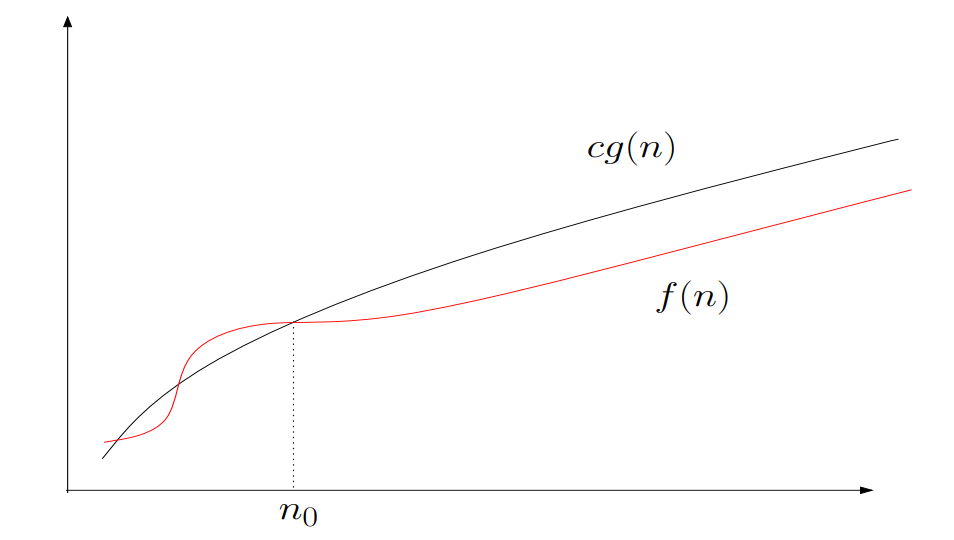
1. **Notazioni asintotiche**

**1.1 - Notazione O per il confronto di funzioni**

Siano f e g due funzioni su N:

Diremo che f(n) = O(g(n)) **se e solo se** la prima funzione (f) è sempre più piccola della seconda funzione moltiplicata per un numero intero c qualsiasi da un certo n0 in poi:

f(n) <= cg(n) per ogni n>=n0

Graficamente:

Es.

2n3+7n2√n + 4 = O(n3)

Per essere uguale ad O(n3)

F(n) deve essere <= di cn3

Quindi

2n3+7n2√n + 4 <= 2n3+7n2\*n + 4

Perché √n è sempre minore di n

2n3+7n2\*n + 4 <= 2n3+7n3+ n3

Perché n3 è sempre maggiore di 4 per ogni n>1

Allora:

2n3+7n3+ n3 = 10n3 quindi c=10

Generalizzando

Sia k un intero

Aknk+ Ak-1nk-1+ Ak-2nk-2+ Ak-3nk-3+…+ A1n+ a0=O(nk)

Ovvero qualsiasi sia f(n) la O è di grado massimo il grado massimo di f, ovvero f non crescerà mai oltre il grado massimo k di O(nk)

Quindi

Aknk+ Ak-1nk-1+ Ak-2nk-2+ Ak-3nk-3+…+ A1n+ a0=O(nk)

Aknk+ Ak-1nk-1+ Ak-2nk-2+ Ak-3nk-3+…+ A1n+ a0<=cnk

Aknk+ Ak-1nk-1+ Ak-2nk-2+ Ak-3nk-3+…+ A1n+ a0 <= |Ak|nk+| Ak-1|nk-1+ |Ak-2|nk-2+ |Ak-3|nk-3+…+ |A1|n+ |a0|

Perché ogni numero A è sempre minore od uguale al suo valore assoluto

|Ak|nk+| Ak-1|nk-1+ |Ak-2|nk-2+ |Ak-3|nk-3+…+ |A1|n+ |a0| <= |Ak|nk+| Ak-1|nk+ |Ak-2|nk+ |Ak-3|nk+…+ |A1|nk+ |a0|nk

**1.2 – Notazione Ω per il confronto di funzioni**

Date due funzioni f e g. f(n)= Ω(g(n)) significa che:

Esiste c>0, n0 appartenente ad N0 tale che f(n) >= cg(n) per ogni n>n0

Es. Sia f(n)=n2-2n e g(n)= n2

n2-2n = Ω(n2)

Occorre provare che esiste una c > 0, ed n0 tale che n2-2n = cn2. Per ogni n > n0

Quindi: n2- 2n >= cn2

Mettendo in equivalenza n2:

n2- cn2 >= 2n

mettendo in evidenza:

n2(1-c) >= 2n

Dividendo per n

n(1-c) >= 2

Dividendo per 1-c:

n >= 2/(1-c)

Vale quindi:

f(n) = Ω(g(n)) ⇔ ∃c, n0 : f(n) ≥ cg(n), ∀n ≥ n0

(Esistono c ed n0 tale che f(n) > cg(n) per ogni n maggiore di quell’n0)

⇔ ∃c, n0 : g(n) ≤ (1/c) f(n), ∀n ≥ n0

Che significa:

⇔ g(n) = O(f(n)).

Quindi trovando omega troviamo anche O

Es.

√n = Ω(log n))

n= Ω(√n)

nk+1=Ω(nk)

n! = Ω(2n ), nn = Ω(2n).

**1.3 – Notazione Θ per il confronto di funzioni**

Siano f e g due funzioni

Dire che f(n) = Θ(g(n)) significa dire che:

f(n) = O(g(n)) & f(n) = Ω(g(n)) -> ∃c1, c2, n0 : c1g(n) ≤ f(n) ≤ c2g(n), ∀n ≥ n0.

ovvero che f e g crescono alla stessa velocità

Es.

4n2 + log n = Θ(n2)

Innanzitutto: 4n2 + log n = Ω(n2 ) in quanto 4n2 + log n ≥ n2 .

Inoltre: 4n2 + log n = O(n2 ) in quanto 4n2 + log n ≤ 4n2 + n ≤ 5n2 .

La notazione viene utilizzata per indicare precisamente la velocità di crescita di una funzione

Sia g(n) = n + 2n3 − 3n4 + 4n5

1. g(n) = Ω(n log n) è vera perché n5 banalmente cresce più velocemente di n log n
2. g(n) = Θ(5n6) è falsa perché g(n) al massimo è O(5n6) perché non cresce almeno 5n6
3. g(n) = O(n10) è vera perché banalmente g(n) non cresce più di n10
4. g(n) = Ω(n5) è vera perché g(n) cresce almeno di n5

Nell’analisi degli algoritmi si definisce tempo lineare quando il tempo di esecuzione dell’algoritmo è al massimo un fattore costante per la dimensione dell’input

Tempo logaritmico= Il tempo di esecuzione dell’algoritmo `e al pi`u un fattore costante per il logaritmo della dimensione dell’input

Tempo quadrico= L’algoritmo esamina tutte le coppie di dati elementi

Tempo cubico = O(n 3 ) L’algoritmo esamina tutte le triple di dati elementi

Insieme indipendente= insieme di punti non uniti da un segmento

**2 - Analisi degli algoritmi**

Analizzare un algoritmo significa valutare le risorse utilizzate da un algoritmo, in termini di tempo e di spazio per la memorizzazione.

Generalmente la qualità di un algoritmo si basa sul suo tempo di esecuzione, questa pratica non è sempre precisa, perciò, vengono utilizzate le **notazioni asintotiche** (ignorando quindi le costanti moltiplicative in quanto sono trascurabili) per definire i tempi di esecuzione in modo approssimativo ma “preciso”.

Il tempo di esecuzione di un programma dipende da vari fattori:

* Quelli non controllabili:
  1. Abilità del programmatore
  2. Linguaggio utilizzato
  3. Linguaggio macchina del computer che lo esegue
  4. Dal codice generato dal compilatore
* Quelle controllabili:
  1. Dalla taglia dei problemi (l’input)
  2. Dal modo in cui è posto l’input (es. sequenze ordinate o non ordinate)
  3. Dal numero di operazioni elementari eseguite dal’algoritmo

La complessità di un algoritmo dipende esclusivamente da:

* Il suo numero di operazioni elementari:
  1. Operazioni matematiche
  2. Operazioni di confronto
  3. Operazioni logiche
  4. Seguire un puntatore
  5. …

Poiché un’operazione elementare utilizza una quantità di tempo costante, convenzionalmente la definiamo come un'unica unità di tempo comune a tutte le operazioni, tutte queste operazioni elementari hanno come costo O(1)

**2.1 - Taglia dell’input:**

Per la taglia dell’input si possono definire due criteri:

* Costo logaritmico
  1. La taglia dell’input è il numero di bits necessari per rappresentarlo
* Costo uniforme
  1. La taglia dell’input è il numero di elementi che lo costituiscono (es. la taglia di un albero è il numero dei suoi nodi)

Concettualmente si considera che un elemento della taglia dell’oggetto da calcolare sia della misura di una parola di memoria

**2.2 – Forma dell’input:**

Come già detto la complessità di un algoritmo (e la sua “bontà”) può dipendere anche dalla forma dell’input (es. array ordinato o non ordinato).

Per un calcolo universale e preciso bisogna considerare principalmente il caso più sfavorevole che l’algoritmo può accettare. Sia quindi T(n) **il tempo massimo** che l’algoritmo richiede per calcolare tutti i possibili input di taglia n. Questa premura è utile per ricavare un limite superiore all’algoritmo, quindi, **l’algoritmo non può mai richiedere più tempo di quello calcolato**, generalizzandone la “bontà”.

**2.3 – Costo delle operazioni**

Le operazioni semplici come quelle matematiche hanno costo O(1),

mentre nel caso di istruzioni IF…ELSE il costo sarà O(1)+O(blocco operazioni scelte dall’if),

per i cicli come while o for la complessità è la somma di tutte le operazioni del blocco di istruzioni ripetute tot volte fin quando continua l’esecuzione

**Una somma infinita è uguale a 1/(1-a)**

**Algoritmi ricorsivi**

È un algoritmo caratterizzato dalle chiamate a se stesso con un input piccolo del suo input originale.

Per valutare un algoritmo ricorsivo occorre:

* Determinare la taglia dell’input n
* Determinare per quale valore n0 vale la base della ricorsione
* Determinare il valore T(n0) del tempo di esecuzione nella base della ricorsione (generalmente è una costante c)

**Tempo di esecuzione**

Generalmente allora il tempo di un algoritmo ricorsivo T(n) è uguale:

* T(n) = c nel caso l’input corrisponde con la base della ricorsione (e questa ha tempo di esecuzione costante)
* T(n) = aT(f(n)) + g(n)
  1. a è una costante che indica il numero di chiamate ricorsive eseguire
  2. T(f(n)) è il tempo impiegato dall’algoritmo per eseguire tutte le chiamate ricorsive
  3. f(n) è il tempo svolto dalla chiamata ricorsiva
  4. g(n) è il tempo richiesto dall’algoritmo al di fuori dalla ricorsione.

**Risoluzione equazioni di ricorrenza**

Immagine che contiene testo

Descrizione generata automaticamenteData la ricorrenza (scegliendo arbitrariamente a, c e d):

Immagine che contiene testo

Descrizione generata automaticamenteLe sue risoluzioni sono:

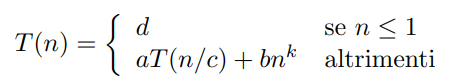
Mentre sia la ricorrenza:

Immagine che contiene testo

Descrizione generata automaticamenteLe sue risoluzioni sono:

**3 - Algoritmi Divide-Et-Impera**

La tecnica Divide-et-Impera generalmente segue una struttura di chiamate ricorsive, e sono organizzate come segue:

Algoritmo DivideEtImpera(input)

if(input abbastanza piccolo da essere risolvo in modo elementare) return risolvi(input) //fase impera

else {

l’input viene suddiviso in n istanze più semplici (input1,input2…inputN)

S1=AlgoritmoDivideEtImpera(input1);

S2=AlgoritmoDivideEtImpera(input2);

…

Sk=AlgoritmoDivideEtImpera(inputN);

Componi la soluzione date le sottosoluzioni calcolate s1,s2…sk

Return Stot;

}

La complessità di questi algoritmi allora è definibile per via di una relazione di ricorrenza T(n), che come detto nel paragrafo precedente, può avere soluzioni:

Es. **Ricerca Binaria in un array ordinato**

Input: una coppia(a,k) dove a è un array di numeri ordinati in senso crescente, k è un numero arbitrario

Output: un intero i, 0<=i<=n-1, se k=a[i], “non c’è altrimenti”

Algoritmo:

s indica l’indice di partenza e d l’indice di fine

RicercaBinaria(a, k, s, d) {

IF(s==d){

IF(k==a[s]){

RETURN (s);

} ELSE {

RETURN “non c’è”  
 }

}

c=(s+d)/2

IF(k<=a[c]) {

RETURN(RicercaBinaria(a, k, s, c);

} ELSE {

RETURN(RicercaBinaria(a, k, c+1, d);

}

}

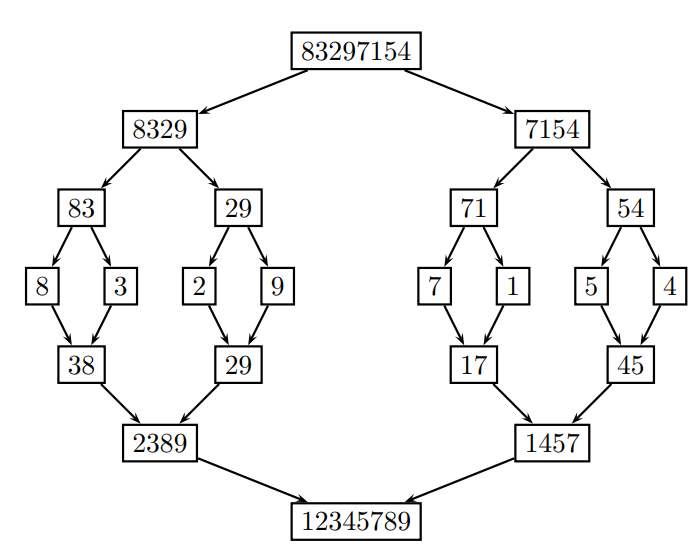
ANALISI DELLA COMPLESSITÀ:

Sia T(n) la complessità, si ha che:

T(n/2) in quanto ad ogni chiamata dell’algoritmo può avvenire una sola chiamata ricorsiva sulla metà dell’input (n/2). +c indica poi le operazioni elementari necessarie per risolvere i casi base. La complessità è O(n log n)

Esempio di algoritmo ricorsivo: MergeSort

Il merge sort è un algoritmo che prende in input un array di elementi e li ordina in modo ricorsivo: questo attraverso una suddivisione dato l’elemento mediano e, ricorsivamente, ogni suddivisione viene divisa a sua volta fino a che non resta l’elemento singolo che ha risoluzione immediata:

L’array viene diviso tramite la sua metà in due sottoarray (8329 e 7154). A loro volta questi vengono divisi nelle loro metà. La ricomposizione poi tipo degli elementi 8 e 3 in 38 avviene tramite la fase di **impera** che confronta i frutti della ricorsione, in questo caso ordinando gli array prodotti dalla suddivisione.

L’analisi dell’algoritmo di comparazione degli array è pari ad Θ(n)

Mentre per l’algoritmo di MergeSort effettua due chiamate ricorsive (su una taglia n/2), e una chiamata all’algoritmo di comparazione, quindi:

T(n) = 2T(n/2) + Θ(n) -> in questo caso allora la complessità è O(n log n)

Non esiste concettualmente un algoritmo più efficiente di questo. Dimostrare questa affermazione è molto più difficile che dimostrare che un algoritmo è possibile. In quanto bisogna dimostrare che questo algoritmo sia migliore per qualsiasi algoritmo esistente e qualsiasi algoritmo possibile, mentre nel caso opposto basterebbe presentare un controesempio che vale da antitesti all’affermazione.

**Dimostriamo che non esiste algoritmo migliore del MergeSort:**

Sia A un **qualsiasi** algoritmo di ordinamento, capace di ordinare arbitrariamente delle sequenze a=a[0]…a[n].

Il numero di possibili ordinamenti di a (permutazioni) è pari ad n!

Però nel caso dell’ordinamento esiste una sola permutazione (tra tutte le n!) valida.

Quindi l’algoritmo A dev’essere in grado di trovare quell’unica permutazione, per fare ciò l’algoritmo esegue dei confronti: siano i e j due indici di a, l’algoritmo confronterà a[i] e a[j]. Siano C e D due insiemi così definiti: C={ a[i], a[j] | a[i]<=a[j] } e D={ a[i], a[j] | a[i]>=a[j] } quindi C e D sono gli insiemi di tutte le possibili permutazioni secondo la relazione d’ordine definita.

Palesemente |C| + |D| = n!

Quindi ne segue che |C| >= n!/2 oppure |D| >= n!/2

Allora assumiamo che:

* |C| >= ½ \* n!

Per via del metodo di analisi degli algoritmi, consideriamo il caso peggiore di questo algoritmo A

Consideriamo quindi il secondo confronto a[k] e a[s], a loro volta, come nel primo caso esistono due insiemi E ed F che contengono tutte le possibili permutazioni che rispettano a[k] >= a[s] e viceversa.

Da ciò |E| + |F| = |C|.

L’algoritmo A, quindi, itererà i confronti finché non avrà trovato che a[m] = a[h] (un solo elemento) quindi l’ordinamento sarà certo

Il numero di confronti sarà tale che n!/2i <= 1 -> Osserviamo che i >= log n!

Per provare quest’affermazione è necessario provare che log n! cresce ALMENO n log n:

log n! > log (n/2)n/2 -> n/2 log n/2 -> n/2 \*(log n – log 2) -> n/2 \* (log n -1) -> n/2 \* log n – n/2 >= n log n

Allora l’affermazione è provata

**Altre applicazioni del metodo Divide-et-Impera (QuickSort)**

Es. Problema algoritmico: in un array a definiamo il rango di un generico elemento x come il numero di elementi (nell’array) minori o uguali di x. Con il minor rango possibile = 1 (poiché ogni elemento è minore o uguale a sé stesso)

Input: array a=a[0]…a[n-1], intero k

Output: rango di k

Una possibile soluzione è ordinare l’array e restituire il numero di elementi prima di k. Applichiamo quindi un algoritmo di ordinamento con complessità O(n log n)

Questa soluzione può essere implementata controllando il caso in cui:

* K è una costante c (c=1, c=2 …) basta quindi cercare il minimo se c=1, il secondo minimo se c=2 ecc. allora Θ(n)

Sia x un elemento (scelto arbitrariamente) dell’array a, dividiamo in due sottoarray A1 e A2, in A1 andranno tutti gli elementi minori di x e in A2 quelli maggiori.

Quindi se:

* |A1| = k-1 -> Allora l’elemento x corrisponde con l’elemento di grado k
* |A1| >= k -> Allora l’elemento si trova in A1
* |A1| < k-1 -> Allora l’elemento si trova in A2

Nel primo caso ovviamente il problema è risolto.

Nel secondo e terzo caso bisognerà ricorrere nei rispettivi sottoarray. Spostando a sinistra tutti gli elementi più piccoli di x e a destra i più grandi.

Questo avverrà attraverso l’uso di un ***pivot*** usato per scorrere, verso sinistra e verso destra, l’array. Scambiando l’elemento *errato* in entrambi i sottoarray

In questo caso la complessità dell’algoritmo dipende esclusivamente dalla scelta del pivot. Nel caso di x minimo o massimo, avremmo rispettivamente, O(n) oppure O(n2)

Studiando quindi il caso peggiore dimostriamo solo l’inefficienza dell’algoritmo. Quindi consideriamo il **caso medio** (che ha complessità O(n/2)).

Scegliendo il pivot a caso, statisticamente parlando, la probabilità di avere un elemento di rango r è 1/n (n dimensione array).

La soluzione delle equazioni di ricorrenza attraverso l’induzione è necessaria per dimostrarne la complessità anche per taglie di input diverse (col metodo di dimostrazione precedente veniva provato un solo input n)

Partendo dall’ipotesi induttiva in cui T(k) <= ck

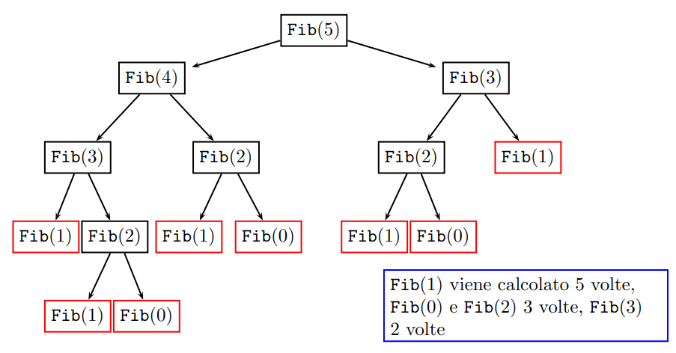
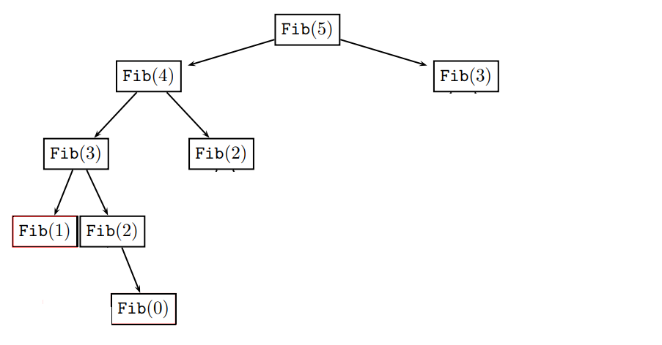
**4 - Programmazione Dinamica**

È un’implementazione ed evoluzione della tecnica divide-et-impera, che era basato su una struttura:

1. Risolvi il problema nel caso sia sufficientemente piccolo
2. Dividi il problema in sottoproblemi di dimensione inferiore
3. Risolvi ricorsivamente i sottoproblemi
4. Combina le soluzioni dei sottoproblemi per una soluzione generale

Il problema di efficienza del Divide-et-impera nasce quando dal passo 2 nascono due sottoproblemi **identici** (o simili/della stessa classe di problemi), infatti questi verranno risolti entrambi utilizzando il doppio del tempo. Questo **non avviene** nella **programmazione dinamica**, che, riconosce i problemi simili e li risolve una sola volta

Per esempio, nel calcolo di un n numero nella sequenza di fibonacci, l’algoritmo divide-et-impera ripeterà la somme già calcolate arrivando ad una complessità O(2n). Nella programmazione dinamica, ad ogni chiamata ricorsiva dell’algoritmo verrà memorizzato in un array il risultato di quella chiamata (usando spazio in più ma riducendo di molto le operazioni) da cui controlliamo i valori già presento ad ogni ricorsione dell’algoritmo



Diventa ->

Per via di ricorsioni già calcolate (come Fib(1), Fib(0), Fib(2) e Fib(3) ), le operazioni risultano ridondanti, le eliminiamo per via di un controllo che verifica se una determinata ricorsione n sia stata già calcolata. Arrivando ad una complessità Θ(n), ottenendo un miglioramento esponenziale.

La soluzione in programmazione dinamica può anche essere scritta in termini iterativi piuttosto che strettamente ricorsivi (in questo caso l’iterazione può essere più rapida della ricorisone)

La tecnica utilizzata in questa soluzione si chiama **Memorization**

La **programmazione dinamica** quindi si basa su:

* **Ricorsione**
* **Memorizzazione**

Quindi ogni algoritmo in programmazione dinamica viene formulato attraverso due passi:

1. Definizione del problema in termini ricorsivi
2. Soluzione globale al problema in modo ricorsivo, controllando le soluzioni già ottenute evitando di sprecare tempo nel ricalcolarle

La soluzione di un problema avviene in modo ***bottom-up***

Per esempio:

**Input**: Valore monetario V, ed un insieme v[1]…v[n] dove v[1] è il massimo e v[n] = 1

**Output**: il minimo numero di monete il cui valore sia uguale a V

Indichiamo con ai >= 0 il numero di monete v[i] che usiamo. Si vuole limitare al minimo il numero di monete così definite:

a­­­­­­­­­­­­­­­1 + a2 + … + an ->

Sia V=26 il vettore delle monete pari a {1,2,3,4} ed il vettore dei valori delle monete sarà:

v[1] = 10, v[2] = 5, v[3] = 2, v[4] = 1

Allora: 10\*1 + 5\*2 + 3\*2 + 4\*1 = 26

Creiamo l’algoritmo in PD(Programmazione Dinamica):

**Passo 1. Formulazione del problema in termini ricorsivi**

Innanzitutto serve identificare quali sono i sottoproblemi del problema principale, questi sono tutti quelli con valore 0 <= j <= i che possono essere rappresentabili con i valori v[1] > v[2] > … > v[n]=1

Sia quindi C(i,j) il minimo numero di monete per rappresentare la cifra j <= V, con le monete di valore v[i] > v[i+1] > … v[n] = 1

Sia V=12. Il valore delle monete v[1] = 10, v[2] = 6, v[3] = 1.

Formuliamo una matrice che ha per numero di righe i valori delle monete (3 righe = i) e tante colonne quanto è grande V (12 colonne = j)

Immagine che contiene tavolo

Descrizione generata automaticamente

C(i,j) quindi rappresenta il minimo numero di monete per esprimere j <= V.

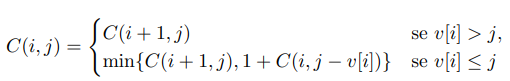
Esistono 2 casi:

1. La moneta v[i] non appare, in questo caso possiamo dedurre che la soluzione sarà data da v[i+1], allora C(i,j) = C(i+1, j)
2. La moneta v[i] appare. Allora, C(i, j) = 1 + k dove 1 rappresenta l’uso della moneta v[i] e k il resto delle monete usate. Allora **k = C(i, j-v[i])**, seguendo la dimostrazione per assurdo:
   1. sia k> C(i,j-v[i]), ci troveremmo che 1 + C(i, j-v[i]) < C(i,j). Questo è un assurdo poiché, per definizione, C(i,j) è il minimo.
   2. Questo avviene anche nel caso k < C(i, j-v[i]), poiché, sempre per definizione, C(i,j) è il minimo

Poiché non conosciamo se il valore v[i] apparirà o meno, ricorriamo in entrambi i sottoproblemi:

1. C(i+1, j)
2. C(i, j-v[i])

Quindi vale che:



Nei casi base varrà:

Immagine che contiene grafico

Descrizione generata automaticamente

Passo 2. **Creazione di una tabella per verificare le soluzioni già ottenute**

Immagine che contiene testo

Descrizione generata automaticamente

Ottenendo complessità (nel caso migliore O(1)) O(nV) con V la taglia del numero in input da dividere in monete.

Questo problema rappresenta il primo **problema di ottimizzazione**, poiché questi sono caratterizzati dalla possibile associazioni di output diversi ad input diversi. Ogni input viene quindi definito da un numero detto **costo** e tra le possibili soluzioni si sceglie quella che **ottimizza il costo** (questo può significare minimizzare o massimizzare, in base all’output cercato)

Un esempio di problema di ottimizzazione potrebbe essere: **massimizzare il punteggio all’esame**

All’input di n domande D1,D2,D3, …Dn, che valgono rispettivamente P[1], P[2], …, P[n]. Ogni domanda Di richiede t[i] tempo. Il tempo a disposizione è T. si vuole progettare un algoritmo che dati gli array p[1] … p[n] e t[1] … t[n] restituisca il massimo punteggio possibile nel tempo massimo T.

Sia P[i,j] il massimo punteggio possibile rispondendo ad i domande (D1,D2, … Di … Dn) con tempo massimo j minuti (0 <= j <= T). La soluzione migliore del problema è P[n,T]

Il passo base è definito da una sola domanda:

Immagine che contiene grafico

Descrizione generata automaticamente

In generale, per i=2, …, n e j= 0, …, T

Si possono verificare due casi:

1. Rispondiamo a Di:
   1. Ci guadagneremo p[i] punti e ci fa spendere t[i] minuti. Allora per il resto dell’esame bisogna cercare la migliore soluzione alle domande D1, D2, …, Di-1 con tempo j-t[i]
2. Non rispondiamo a Di:
   1. In questo caso otterremo 0 punti dalla domanda Di, tempo uguale a j di partenza, risponderemo quindi alle altre D1, D2, …, Di-1 domande, ottenendo P[i-1,j] punti

La formula di ricorrenza sarà: Immagine che contiene grafico

Descrizione generata automaticamente

L’algoritmo utilizzerà una tabella i\*j per controllare i risultati già calcolati, ottenendo una complessità di Θ(nT)

**Altri esempi in Programmazione Dinamica**

**Input**: insieme A = {A1,A2, … ,An} di attività di valore v(A1), v(A2), v(An). Ogni attività può essere svolta nell’intervallo di tempo compreso tra si ed fi (con si < fi )

Siano Ai e Aj due attività, queste possono essere eseguite solo nel caso che l’intervallo di tempo della prima non si accavalli con quello della seconda (ovvero la loro intersezione è un insieme vuoto) queste si dicono compatibili

**Output**: Un sottoinsieme S di A a due a due compatibili, di valore la massima possibile sommatoria di tutti i v(A) in S.

Es:

Immagine che contiene schematico

Descrizione generata automaticamente

I possibili massimi si trovano considerando:

* v=4 e v=7 che sono compatibili a due a due (la loro somma è 11)
* V=2 v=3 v=7 (la loro somma è 12)

Generalizzando ordiniamo le attività in base al loro fi con f1<= f2 <= … <= fn. Mentre, per qualsiasi attività j, sia p(j) il più grande indice i tale che l’attività i sia compatibile con j (nel caso non esista sarà p(j) = 0)

Definiamo Ω come un insieme delle soluzioni (ottime) di A, ed n l’ultima attività di A si verificheranno due casi:

1. n appartiene all’insieme Ω:
   1. Ω-{n} è una soluzione ottima relativa all’insieme { 1,2,…,p(n) } (dove p(n) rappresenta l’ultima attività compatibile con n), questo si dimostra (per assurdo) attraverso la definizione stessa di Ω, poiché si starebbe affermando che in Ω-{n} è presente una soluzione ottima migliore di Ω stesso, quando poi Ω rappresenta la miglior soluzione possibile ottimale
2. n non appartiene all’insieme Ω:
   1. Ω è chiaramente una soluzione ottima per l’insieme delle attività {1, 2, …, n-1}

Formulando la soluzione in termini ricorsivi:

* Si considera (genericamente) l’attività j dove j è compreso tra la prima attività e l’ultima (n).
* Definiamo OPT(j) il valore di Ω(j), anche se come soluzione cerchiamo OPT(n)
* Si distinguono due casi:
  1. J appartiene a Ω(j).
     + Allora Ωj non può contenere le attività p(j) +1 … j-1. Inoltre Ωj – {j} è una soluzione ottima per le attività {1, 2, … p(j) }, ovvero il valore di OPT(j):
       - OPT(j) = v(j) + OPT( p(j) )
  2. J non appartiene a Ωj.
     + Allora sappiamo che:
       - OPT(j) = OPT(j-1)

Allora OPT(j) = max{ v(j) + OPT( p(j) ), OPT(j-1) }, ovvero in termini di ricorsione:

Immagine che contiene testo

Descrizione generata automaticamente

Ovvero:

T(n) = T(n-2) + T(n-1) + d (come fibonacci ha una complessità esponenziale)

Applicando la memorization, utilizziamo un array di memorizzazione in cui vengono memorizzati tutti i valori p(j) già calcolati evitando di calcolarli di nuovo si ottiene una complessità O(n)

**Altri esempi di Programmazione dinamica**

Siano a=a[1]…a[m] e b=b[1]…b[n] due sequenze, bisogna trovare la più lunga sottosequenza (non necessariamente continua) comune tra a e b. denominiamo il problema con **LCS** (longest common sequence). Questo problema ha applicazioni in biologia molecolare, nell’analisi di testi (come con il comando *diff* su Unix).

Es:

a = A B C B D A B b = B D C A B A

la sottosequenza comune può essere: B C A

Oppure: B C A B

O anche: B C B A

Le possibili sottosequenze allora sono 2n

**PRIMO PASSO** per risolvere il problema dobbiamo definire in modo ricorsivo il problema:

Definiamo c[i,j] la più lunga sequenza comune considerando i primi i indici di a e i primi j indici di b

**Passo base**

Banalmente: c[i,0] = 0 = c[0,j] poiché in entrambi i casi andiamo a considerare 0 elementi (prima in b e poi in a)

**Passo ricorsivo**

Andiamo a definire i vari casi del problema:

1. a[i] ≠ b[j]:

Allora risulta che a[i] e b[j] non possono essere contenuti **entrambi** nella più lunga sottosequenza (dato che non sono comuni, ma uno dei due può comparire in **mutua esclusione**), da qui ricaviamo quindi altri 2 sottoproblemi:

* + 1. a[i] non appartiene alla più lunga sottosequenza che sarà formata da:

a[1]…a[i-1] e b[1]…b[j] nel caso ottimo

* + 1. b[j] non appartiene alla più lunga sottosequenza che sarà formata da:

a[1]…a[i] e b[1]…b[j-1] nel caso ottimo

Il costo di questo caso allora è c[i,j] = max{ c[i-1, j] , c[i, j-1] }

1. a[i] = b[j]

Sia d[1]…d[k] la più lunga sottosequenza di a e b, allora banalmente c[i,j] = k

Consideriamo quindi la soluzione ottima d per i e j, questa conterrà anche la soluzione ottima per i-1 e j-1 con d[1]…d[k-1].

Questo si dimostra per assurdo:

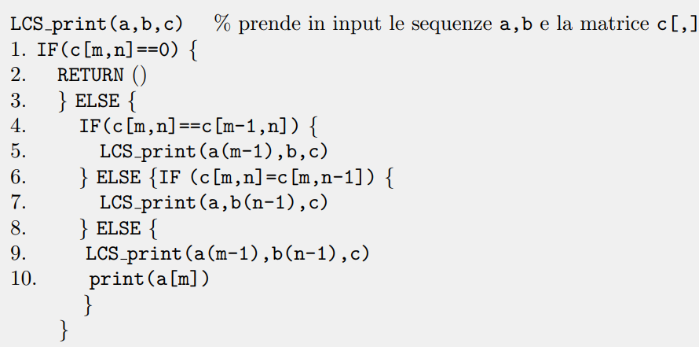
Supponendo che esista una sequenza comune x tra a[1]…a[i-1] e b[1]…[j-1] di lunghezza maggiore di k, aggiungeremo a[i]=b[j] alla fine della sequenza x, questa avrebbe lunghezza maggiore o pari a k+1. Questo però contraddice l’ipotesi del problema in cui c[i,j] = k è la sequenza comune più lunga possibile.

Il costo in questo caso è di c[i,j] = c[i-1, j-1] + 1

Riassumendo in termini di equazioni di ricorrenza:

Immagine che contiene grafico

Descrizione generata automaticamente

Per risolvere il problema in tecnica di programmazione dinamica, inseriamo in una tabella c(i,j) la lunghezza delle sequenze comuni già calcolate:

L’algoritmo in programmazione dinamica ha complessità O(m+n).

Immagine che contiene tavolo

Descrizione generata automaticamenteEs tabella nel caso di a=GDVEGTA e b=GVCEKST, la tabella risultante sarà:

**Problema Distanza di Edit**

Siano s e t due sequenze s[1]…s[m] e t[1]…t[n], la distanza di edit tra s e t ( dist(s,t) ) è definita come il minimo numero di inserzioni, cancellazioni o sostituizioni per trasformare s in t

Le applicazioni di questo algoritmo sono:

* Correzione automatica di parole
* Linguistica (per cercare la distanza linguistica)
* Correzioni di errori in OCR (riconoscimento ottico dei caratteri)

Es:

s=albero t=labbro

albero –(cancella a)-> lbero –(inserisci a)-> labero –(sostituisci e con b)-> labbro

Quindi dist(s,t) = 3

**Formuliamo** quindi il **problema** in **termini ricorsivi**:

Siano s e t due sequenze s[1]…s[m] e t[1]…t[n]

**Passo base**

Notiamo che possiamo sempre trasformare s in t cancellando tutti i caratteri di s ed inserendo i caratteri di t, quindi dist(s,t) < n+m

**Passo ricorsivo**

Siano 1 <= i <= m e 1 <= j <= n due indici.

Consideriamo le 3 operazioni consentire:

1. **Sostituzione**

Possiamo sostituire s[i] con t[j], lasciando irrisolto il sottoproblema s[1]…s[i-1] e t[1]…t[j-1].

Quindi il totale delle operazioni sarà 1 + dist( s[1]…s[i-1], t[1]…t[j-1] )

1. **Cancellare s[i]**

Cancellando s[i] dobbiamo poi risolvere il sottoproblema s,t con s[1]…s[i-1] in t[1]…t[j].

Il costo totale delle operazioni quindi sarà 1+dist(s[1]…s[i-1], t[1]…t[j] )

1. **Inserire t[j]**

Inserendo il valore t[j] alla fine della sequenza s, lascerà da risolvere il sottoproblema con s[1]…s[i] e t[1]…t[j-1]

Il costo totale delle operazioni sarà 1+ dist(s[1]…s[i], t[1]…t[j-1])

1. **s[i] = t[j]**

Definiamo l’operazione diff(x,y) = 1 se gli elementi passati sono uguali, 0 altrimenti.

Il costo di dist(s[1]…s[i],t[1]…t[j] ) è quindi = min { dist(s[1]…s[i-1],t[1]…t[j-1] )+ diff(s[i], t[j],

dist(s[1]…s[i-1],t[1]…t[j] ) +1,

dist(s[1]…s[i],t[1]…t[j-1] ) +1 }

Il passo base sarà con i = j = 0 dove entrambe le sequenze risultano vuote, quindi:

dist(s[0], t[1]…t[j]) = j e dist(s[1]…s[i], t[0]) = i

L’algoritmo Divide-Et-Impera può essere migliorato seguendo due approcci:

1. Tecnica **Memorization**, aggiungendo all’algoritmo una tabella con tutti le soluzioni già calcolate dei sottoproblemi risolti, questo evitava ricorsioni inutili
   1. Preserva la peculiarità ricorsiva dell’algoritmo divide et impera originale
   2. Uno dei **contro** però ha un costo molto maggiore in spazio, che sia nello stack delle chiamate per via delle ricorsioni sia per la tabella utilizzata
2. Vengono risolti tutti i sottoproblemi in tecnica bottom-up in maniera iterativa
   1. Risulta molto efficiente in termini di spazio e termini di tempo, per via della risoluzione di **tutti** i sottoproblemi possibili (anche quelli che non si presenteranno o che non concorreranno alla soluzione del problema)

La programmazione dinamica può essere applicata **solo nel caso valga** questa **proprietà**:

*Sia S una soluzione ottima ad un problema di ottimizzazione P, allora tutte le componenti di S sono soluzioni ottime ai sottoproblemi di P*

Per esempio, questa proprietà non vale:

Immagine che contiene diagramma

Descrizione generata automaticamenteDato un grafo di 5 nodi, vogliamo calcolare la massima distanza percorribile (dove la massima distanza è quanti archi vengono percorsi):

Il cammino più lungo percorribile è a-b-c-d-e, la proprietà dice che a sua volta questa dovrebbe essere sempre soluzione ottima. In questo caso considerando la partenza da b, la soluzione migliore non è b-c-d-e bensì b-c-a-d-e. Questo va in contro alla proprietà necessaria. Al momento non esistono migliori algoritmi

**Tecnica Greedy**

Un algoritmo greedy utilizza una tecnica iterativa, appunto, iterando verso una soluzione (componendola di passo in passo).

Il funzionamento è il seguente:

1. Trova una soluzione per un sottoproblema di piccola taglia
2. Ad ogni passo (iterazione) prova ad aggiungere componenti alla soluzione precedente cercando di comporre la soluzione completa
3. Ad ogni passo compone varie possibili soluzioni, preferendo la migliore

**Esempio**

Input: Un server che serve ed n clienti impiegando ciascuno un tempo ti

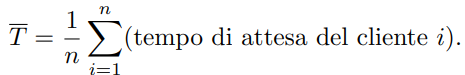
Output: L’ordine in cui servire i clienti in modo da minimizzare il tempo medio di attesa di ogni cliente

Supponiamo di avere 3 clienti, i tempi di servizio sono t1=5, t2=10, t3=3

Immagine che contiene testo

Descrizione generata automaticamenteSappiamo che esistono 3! = 6 possibili modi di servire questi clienti, riportati in questa tabella:

Notiamo che servendo i clienti in ordine crescente in base al loro tempo richiesto, definiamo quindi il problema come sommatoria di tutti i tempi di attesa dei clienti i con i che va da 1 ad n:



Poiché 1/n non dipende da i ed è un dato fisso ci concentriamo sul problema composto solo dalla sommatoria:

Immagine che contiene testo

Descrizione generata automaticamente

Vogliamo quindi aggiungere un generico cliente j

Definiamo con A la somma di tutti i tempi di attesa dei clienti i1…im allora il tempo di attesa di j sarà pari a:



L’algoritmo greedy avrà funzionamento:

1. Ordinamento dei clienti in modo che siano in ordine non decrescente
2. Servi i clienti nell’ordine stabilito

Denotiamo allora con P=p1,p2,…,pn una generica permutazione dei clienti in attesa, e sia s1=tp1, s2=tp2, … sn= tpn

L’i-esimo cliente, dell’insieme in ordine P, è si ed il tempo totale di attesa è:

T(P) = s1 + (s1+s2) + (s1+s2+s3) … -> ns1 + (n-1)s2 + (n-2)s3 … ->

Consideriamo la permutazione P come una permutazione in cui i clienti sono ordinati in base ai loro tempi di richiesti (s). si può dimostrare, prendendo due generici clienti a e b che se uno che viene servito prima ma con tempo di servizio maggiore, esiste miglioramento -> allora non è ottimale.

Un’applicazione di questo problema è la lettura e l’organizzazione dei file in un nastro magnetico, in quanto necessita di accesso sequenziale:

Sia L un array dei file presenti nel nastro, il tempo per la lettura di un file i è la sommatoria di tutti i tempi a lui precedenti, quello medio è la somma di tutti i tempi di tutti i file. Definiamo con pi(i) l’indice del file i, allora l’algoritmo con soluzione ottima si ha quanto esiste una permutazione di p(i) definita come nel problema precedente.

**Generalmente gli algoritmi a programmazione dinamica producono un algoritmo una soluzione ottima quando un algoritmo greedy produce una soluzione ottima (per problemi di ottimizzazione)**, anche se greedy più efficiente in termini di tempo, **non è tuttavia valido sempre l’opposto**

Un esempio di un algoritmo in cui greedy non produce soluzione ottima è quello dello zaino 0-1:

**Input**: Insieme A di n oggetti, con valori v[1],v[2],…,v[n] e pesi: w[1],w[2],…,w[n] ed una capacità massima W

**Output:** Un sottoinsieme S di A di **peso totale**= somma di tutti gli elementi <= W

e di **valore totale** = la massima somma di tutti i valori degli elementi presenti

Supponiamo:

* A={1,2,3}
* v[1]=60$, v[2]=100$, v[3]=120$
* w[1] = 10, w[2] = 20, w[3] = 30

allora 1 vale 6$ al kg, 2 vale 5$ al kg e 3 vale 4$ al kg

L’algoritmo greedy sceglierebbe 1 (perché ha il valore maggiore in base al peso) e poi 2 perché è il secondo con valore maggiore in base al peso, e poi basta perché ha finito il peso trasportabile, ottenendo un valore pari a 160$

In programmazione dinamica invece, consideriamo c[k,w] la soluzione ottima per un problema in cui si vuole il valore massimo in {1,…,k} con capacità di carico totale w

Esistono allora due casi:

1. Non prendo l’oggetto k

Allora la soluzione sarà di valore c[k-1,w]

1. Prendo l’oggetto k

La soluzione sarà c[k-1, w-w[k] ] + v[k]

Allora la soluzione al problema sarà:

C[k,w] = max{ c[k-1, w-w[k] ] + v[k], c[k-1,w] }

Greedy invece riesce a risolvere in modo ottimo un problema simile, in cui possiamo anche scomporre l’elemento da aggiungere nello zaino piuttosto che aggiungere l’intero oggetto

**Compressione dati (greedy)**

Sia X una generica stringa in un alfabeto (anche quello italiano), si vuole codificare in binario questa stringa utilizzando il numero minore possibile di bit, questo significa *comprimere X*. Questa necessità “aumenta” virtualmente la dimensione del disco in quanto ogni dato utilizzerà meno spazio in memoria, inoltre questo aumenta l’efficienza nella trasmissione dei dati.

Generalizziamo il problema:

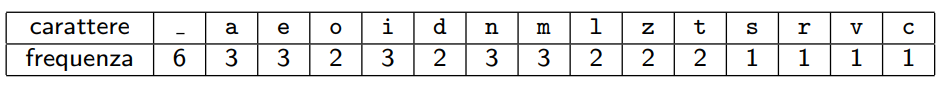
I metodi di codifica dei dati si configurano attraverso il sistema ASCII o Unicode, in cui a caratteri o “simboli” corrisponde una stringa di lunghezza fissa di x bit (Es 1100001 = A).

Nel nostro nuovo sistema di codifica “compatto” detto ***di Huffman***:

* ogni simbolo viene codificato in una sequenza binaria di lunghezza variabile, e, sia c una lettera da codificare, **f(c)= #occorrenze di c in X \* ogni carattere c in X**

Es codifica di Huffman:

nel\_mezzo\_del\_cammin\_di\_nostra\_vita viene scomposto in:

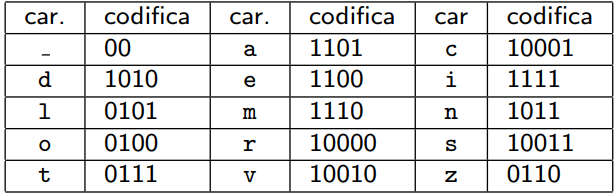


Compresa la codifica di un testo semplice, si può passare alla codifica di un intero testo concatenando le *parole codice* (ovvero le stringhe che rappresentano le parole codificate), e l’insieme di queste è detto **codice**

La codifica di huffman generalmente su testi abbastanza grandi riesce a “risparmiare” il 50% dello spazio usato

**Codice Prefisso**

Sono codici i cui ogni codifica di ogni lettera (parole codice) **non è prefisso** di nessun’altra parola codice:



In questo caso, per esempio, non esistono codici che inizino per: 00, 1010, 0101, 0100, 0111, 10001, 1111, 1011, 10011, 0110 oltre ai codici stessi che hanno questa codifica

**Decodifica**

La decodifica di un testo in codice avviene attraverso questa proprietà:

Es:

Stringa codificata: 101111000101001110110001100110010000101011000101

n = 1011

e = 1100

l = 0101

spazio = 00

m = 1110

z = 0110

o = 0100

d = 1010

e = 1100

Conoscendo ogni codifica di ogni lettera possiamo scomporre in:

1011 1100 0101 00 1110 1100 0110 0110 0100 00 1010 1100 0101

Poiché ogni codifica **non è** prefisso di nessun’altra codifica, è possibile riconoscere nella stringa codificata ogni singola lettera del codice

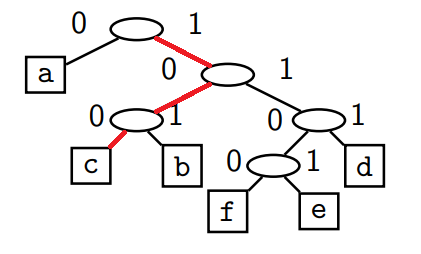
La **proprietà** di **codice prefisso** **è necessaria** per la **decodifica corretta** della stringa codificata in **huffman**, in casi contrari sarebbe necessario leggere tutto il testo codificato per “calcolare” quale sia la codifica corretta del primo carattere, così da decidere il secondo ecc…

Le codifiche prefisso possono essere rappresentate da **alberi binari** in cui le foglie rappresentano le lettere dell’alfabeto che compongono la stringa, mentre i percorsi possibili rappresentano le codifiche dei caratteri

Immagine che contiene diagramma

Descrizione generata automaticamenteEs:

Ogni bit rappresenta il “nodo” seguito ad ogni livello

Es: per codificare c si segue il primo nodo destro della radice (1) e si sceglie di percorrere poi il nodo sinistro (0) scegliendo ancora poi il nodo sinistro (0) che corrisponde con c:

Allora, di conseguenza, il livello è scelto in base a quanti bit sono necessari per rappresentarlo, infatti c (100) è al terzo livello in questo albero

La stringa può quindi essere utilizzata come “guida” nell’albero, infatti con una stringa 11100101:

Scorriamo 3 volte a destra (primi 3 uno) perché non si è finiti su una foglia durante il percorso, si finisce per arrivare sulla foglia d

Ricominciamo leggendo il numero successivo 0, e incontriamo la foglia a

Ricominciando per il bit successivo, succede lo stesso

Ricominciamo leggendo 1, andiamo nel nodo destro, poi leggendo 0 si va nel sinistro e infine si sceglie la foglia b (leggendo 1)

11100101 = daab

**Ritornando al problema di partenza:**

Sia X una data stringa in un alfabeto C di n caratteri, si vuole codificare X in una sequenza Y (più corta possibile) e siano f(c1), f(c2), … f(cn) le frequenze dei simboli nell’alfabeto C

Per codificare X utilizzeremo un albero T. La lunghezza di Y (|Y|) sarà data da:

|Y| = = B(T)

Ovvero, la somma di tutte le frequenza di c \* dimensione di c in bit

Immagine che contiene diagramma

Descrizione generata automaticamenteSia l’albero della stringa pari a:

Allora

|Y| = 45\*1 + 12\*3 + 13\*3 + 5\*4 + 9\*4 + 16\*3 = 224

Si dice **ottimo** l’albero con |Y| più piccolo possibile:

Andiamo quindi a considerare gli **alberi binari completi** (ovvero o 0 o 2 figli)

Esiste, necessariamente, almeno un albero in cui due foglie x,y ∈ C, x,y hanno profondità massima e x,y hanno frequenza minima

**GRAFI**

Un grafo è una coppia di insiemi (V, E):

* V = insieme dei vertici
* E = insieme degli archi tra coppie di vertici

Tipicamente gli archi rappresentano relazioni tra le entità (vertici).

La rappresentazione in grafi è molto utile per la concettualizzazione e astrazione di un problema o di una soluzione

Gli archi possono persino indicare precedenze o direzioni (difatti un albero, per esempio, è un grafo orientato, ovvero con direzione dalla radice alle foglie)

Gli archi hanno si dividono in:

* Simmetrici:
  1. Ovvero le entità sono doppiamente connesse (x connesso ad y e viceversa)
* Asimmetrici:
  1. Ovvero in cui l’arco è orientato e la relazione vale in un solo verso (x in relazione con y ma non viceversa)

Per gli archi **simmetrici** si utilizzano quindi **grafi non diretti**, ovvero non direzionati. In cui il grafo è generalmente indicato con G = (V,E), in cui e è un arco appartenente ad E, tra i vertici generici u e v in V per cui vale che {u,v} = {v,u}

Mentre per archi **asimmetrici** si utilizzano **grafi diretti**. La differenza consiste nella monodirezionalità della relazione, ovvero, generalmente il grafo si indica con G=(V,E), in cui sia e un arco qualsiasi appartenente ad E, orientato da u a v (vertici generici appartenenti a V), allora vale che {u,v} != {v,u}. questo non indica che non possa esistere la relazione {v,u} ma semplicemente che questa non è implicitamente inclusa in {u,v}

**Cammino in un grafo**:

* È una sequenza di nodi in cui ogni nodo è collegato al suo consecutivo, per esempio nel grafo:

Immagine che contiene orologio

Descrizione generata automaticamente

{1,3,7,8} è un cammino in quanto 1 -> 3 -> 7 -> 8

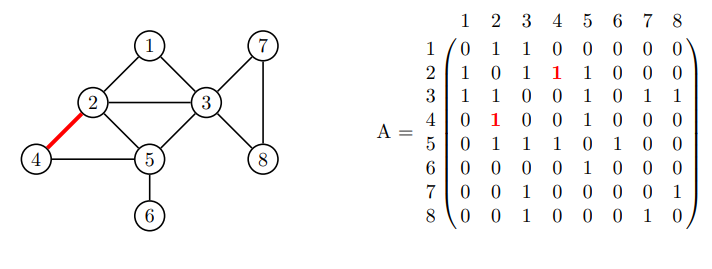
* Un cammino si definisce **semplice** se tutti i suoi vertici sono distinti
* Un grafo si definisce **connesso** se, dati due vertici i e j qualsiasi del grafo, tra questi c’è un **cammino**
* La **distanza** tra due vertici i e j si definisce come il minimo numero di vertici da “attraversare” o meglio, il più breve cammino tra i e j (ammesso che ci sia), nel caso poi non sia possibile arrivare da i a j allora si dice che la distanza è infinita

**Cicli nei grafi**:

* Si definisce ciclo un cammino **non diretto,** un cammino in cui il nodo iniziale i e il nodo finale j sono uguali, pur avendo tutti i nodi nel cammino diversi, ovvero tutti i nodi attraversati sono distinti ma “l’arrivo” è uguale alla “partenza”

**Un grafo ad un calcolatore si può indicare con vari metodi:**

1 – Matrice di adiacenza

Si utilizza una matrice binaria n\*n, in cui l’elemento A[i,j] è pari ad 1 se e solo se esiste l’arco tra i vertici i e j: es. se tra i vertici 3 e 4 esiste un arco **simmetrico**, allora A[3,4] = 1 e A[4,3] = 1. Per il resto la matrice è pari a 0

**Contro**:

* Spazio e tempo per verificare **tutti gli archi** richiesto Θ(n^2)

**Pro:**

* Controllare l’esistenza di un arco dati i e j richiede tempo costante Θ(1)

2 – Liste di adiacenza

Immagine che contiene diagramma

Descrizione generata automaticamenteVengono utilizzate n liste a puntatori, una per ogni nodo. Sia quindi i un generico nodo del grafo, la lista associata ad i contiene tutti i nodi j connessi ad i (ovvero i nodi della lista indicano tutti i nodi a cui è possibile creare un cammino partendo da i)

**Contro:**

* Verificare se esiste un arco dati due nodi i e j prende tempo Θ(numero\_nodi(i)), generalmente il numero nodi è indicato come **grado**

**Pro:**

* Spazio richiesto Θ(n+m), n = |V| e m=|E| (molto meglio di n^2)
* Identificare tutti gli archi (anche per ricostruire il grafo) richiede tempo Θ(n+m)

**Alberi**

Si definisce albero un grafo **non diretto** se esso è connesso e non contiene cicli e possiede n-1 archi, dove n = |V|

Immagine che contiene diagramma

Descrizione generata automaticamenteIn più si definiscono **Alberi radicati** degli alberi orientati da un nodo (in alto) detto radice verso quelli in bassi detti foglie:

(albero radicato in 1)

**Algoritmi di esplorazione(visite) dei grafi**

Un classico problema fondamentale dei grafi è verificare la connettività di questo (ovvero se è connesso o meno)

Oppure un altro problema è verificare se esiste un cammino tra due nodi s e t

Oppure anche trovare il cammino più breve tra due nodi s e t

**DFS, ovvero visita in profondità (Depth First Search) di un grafo G**

È una tecnica per la visita di un grafo qualsiasi, si basa sulla visita di ogni cammino possibile, partendo da un nodo vicino a quello radice, e verificando tutto il suo percorso fino alle foglie, e così per tutti gli altri nodi vicini.

**BFS, ovvero visita in ampiezza (Breadth First Search) di un grafo G**

Ovvero:

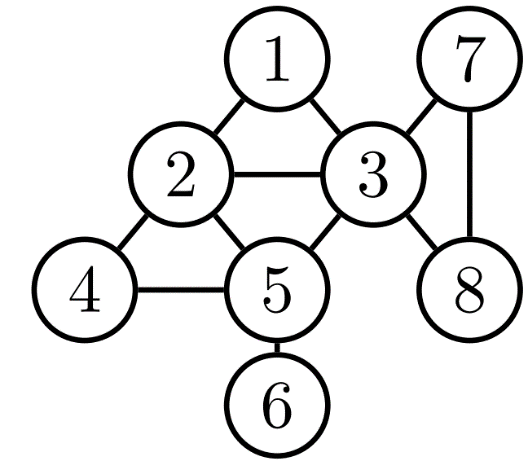
* Esplorare il **grafo G** per scoprire tutti i vertici raggiungibili dal **vertice** di partenza **s**
* Calcola la distanza da s ad ognuno dei vertici raggiungibili

Intuizione:

* Il grafo G viene esplorato scoprendo tutti i vertici a distanza k prima di cominciare a scoprire quelli a distanza k+1, poi k+2, …

Andiamo quindi a scoprire nella lista di adiacenza i vertici, in cui possono verificarsi i seguenti casi:

* Si scopre un nuovo nodo a distanza k:
  1. Lo si aggiunge ad un insieme (Lk) di tutti i vertici a distanza k
* Si scopre un nodo già noto
* Si scopre un nuovo nodo a distanza k+h:
  1. Lo si aggiunge ad un insieme di tutti i vertici a distanza k+h

Es: dato il grafo:

l’algoritmo considerando il nodo 1, scorrerà la sua lista e scoprirà i nodi 2 e 3.

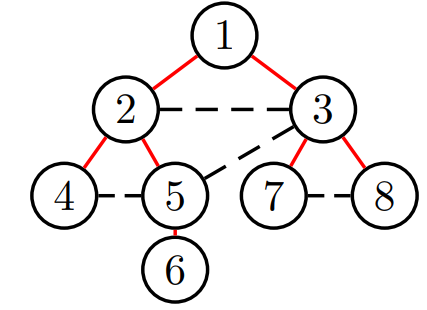
Scorrerà poi la lista del nodo 2 e troverà: 1,3,4,5 ma 1 e 3 già li conosce e aggiungeremo all’insieme i nuovi nodi 4 e 5

Scorrerà la lista del nodo 3 e troverà 1,2,5,7,8 ma 1,2,5 già sono noti e aggiungeremo i nuovi nodi 7 e 8

Scorrerà la lista del nodo 4 e troverà 2,5 che sono già noti

Scorrerà la lista del nodo 5 e troverà: 2,3,4,6 ma 2,3,4 sono giù noti e aggiungerà il nuovo nodo 6

Questo algoritmo poi può rappresentare l’esecuzione attraverso un albero radicato in 1:

Inserendo i nodi scoperti in base alla “prima volta” in cui divengono noti (o marcati come noti/inseriti nell’insieme dei vertici noti), infatti, i nodi 2 e 3 divengono noti dopo la scoperta nella lista del nodo 1 (distanza k+1). 4 e 5 divengono noti dopo la scoperta nella lista di 2 (distanza k+2), così i nodi 7 e 8 diventano noti e marcati per la prima volta nella lista del nodo 3 (distanza k+2) e infine il nodo 6 è noto per via della lista del nodo 5 (distanza k+3)

Questo algoritmo infine riesce a risolvere anche il problema della ricerca della connettività del grafo G per un qualsiasi nodo s

**DFS vs BFS**

La principale differenza è che il primo verifica prima tutto il percorso partendo da un nodo, mentre l’altro visita “livello per livello” tutti i nodi vicini a distanza k+h. Entrambi gli algoritmi presentano complessità O(|V|+|E|), con |V| il numero di vertici (o nodi) presenti nel grafo ed |E| il numero di archi. Variano però nella memoria utilizzata in quanto, la memoria massima richiesta da DFS dipende esclusivamente dall’altezza dell’albero. Mentre per BFS la memoria massima dipende dalla larghezza dell’albero

**Perciò** è preferibile utilizzare **DFS** se la soluzione che cerchiamo è molto lontana dal nodo sorgente, o se il grafo in questione è molto alto.

Analogamente è preferibile utilizzare **BFS** se la soluzione cercata non è troppo lontana del nodo sorgente, o se il grafo in questione non è tanto alto ma molto ampio

**Strutture Dati**

Si definiscono insiemi dinamici i tali insiemi che hanno bisogno di essere modificati “dinamicamente” nel tempo (venendo ampliati, ridotti o cambiati i contenuti). In un insieme siffatto sono possibili le seguenti operazioni:

* Interrogazioni
  1. Ritornano informazioni riguardanti l’insieme o i suoi contenuti senza cambiare la struttura dell’insieme
* Modifica
  1. Cambiano l’insieme o i loro contenuti

Vedremo le operazioni:

* ***Insert(S,x)***
  1. dove S è l’insieme ed x un elemento da aggiungere ad S
* ***Minimum(S)***
  1. Ritorna l’elemento di S con la chiave di minimo valore
* ***Extract-Min(S)***
  1. Ritorna l’elemento con chiave di minimo valore di S e lo cancella da S
* ***Diminuisci-Chiave(S,x,k) o Aumenta-Chiave(S,x,k)***
  1. Decrementa/Aumenta il valore della chiave dell’elemento puntato da x ad un nuovo valore k

Una struttura dati che supporta le operazioni di diminuisci chiave e aumenta chiave è: **(max)code a priorità**, ovvero code basate su priorità massima (viene preferito un elemento a priorità più alta). È una struttura importante per lo scheduling dei job nei calcolatori

Analogamente anche le **(min)code a priorità** (viene preferito un elemento con priorità più bassa) permettono le operazioni sopracitate, queste sono necessarie per applicazioni (event-driven) come l’invio e recezione di pacchetti via internet (in questo caso la chiave di priorità è il timestamp dei pacchetti, preferendo quelli con un timestamp minore = creati prima).

Per le (min)code a priorità può essere implementata con una lista a puntatori o un array ordinato:

Immagine che contiene tavolo

Descrizione generata automaticamente

Entrambi però risultano inefficienti in quanto complessità troppo alta (scorre tutta la struttura in alcune operazioni)

**Heap**

Si definisce Heap un **albero binario perfettamente bilanciato in cui l’ultimo livello può essere incompleto** (ma riempito da sinistra a destra).Vale che per tutti i nodi **v** che **key(v) >= key(parent(v))**

Un heap può essere implementato utilizzando:

1. Puntatori
2. Un array livello per livello riempito da sinistra a destra

Utilizzando un array che memorizza gli elementi dell’heap, ogni nodo in posizione **i** (nell’array) avrà i suoi figli in **posizione** **2i e 2i+1**; analogamente, il **padre** di un nodo qualsiasi in posizione **i** si trova in **posizione i/2**

Proprietà dell’heap:

Data la sua definizione, il **minimo** di un heap si trova alla radice ( la ricerca del minimo è theta(1) )

Si può provare che l’altezza di un heap con n nodi sia log(n):

Dimostrazione:

Si definisca h l’altezza di un Heap con il massimo numero di nodi, ed n il numero di nodi. È palese che n=1+2+4+…+2h = 2h+1 -1

Mentre si consideri l’altezza h di un Heap con il minimo numero di nodi, ed n il numero di nodi. Siano T1 e T2 i sottoalberi (rispettivamente sinistro e destro) di T.

È palese che n= 1+ #nodi T1 + #nodi T2 = 1 + 2h-1-1 + 2h-1-1 = 2h

Di conseguenza il numero di nodi n di un generico heap di altezza h soddisfa la relazione: 2h<= n <= 2h+1-1

Ovvero, (n+1)/2 <= 2h <= n, allora, h=Theta(log n)

**Operazioni sull’heap:**

* **Insert(S,x)**
  1. Inserisci x nell’heap come nuova foglia il più a sinistra possibile nell’ultimo livello
  2. Scambia iterativamente la posizione di x con quella del padre fin quando non venga rispettata la proprietà di base dell’heap, ovvero che **per ogni v, key(v) >= key(parent(v))**

**Algoritmo:**

* L’algoritmo viene implementato come segue:
* **Immagine che contiene testo, lettera

  Descrizione generata automaticamente**
* L’algoritmo inserisce (riga 1 e 2) k come nuova foglia e amplia la size dell’heap. Itera su tutto l’albero scambiando il nuovo elemento e il nodo padre
* **Complessità Theta(log n)**
* **Extraction-Min(S):**
  1. Restituisci l’elemento che si trova alla radice dell’heap (è il minimo)
  2. Sostituisci l’elemento che si trova più a destra e in profondità possibile con la radice
  3. Ripristina la proprietà fondamentale dell’heap, scambiando iterativamente un nodo con il figlio che ha valore chiave minore (ogni nodo viene confrontato con entrambi i figli e viene scambiato nel caso uno dei due sia minore, in tal caso si continua ad iterare su quel sottoalbero che aveva come radice il figlio con valore minore)

**Immagine che contiene testo

Descrizione generata automaticamenteAlgoritmo:**

Min Heapify è la funzione che controlla ogni nodo con i suoi figli (righe 2 a 6) e si preoccupa di scambiarli (riga 7 e 8) nel caso uno dei due sia minore della radice, nel caso esista, richiama poi la procedura sul figlio con valore minore della radice (riga 9)

* **Complessità Theta(log n)**
* **Minimum(S)**
* **Complessità Theta(1)** perché restituisce il valore del nodo radice
* **Aumenta-Chiave(S,i,k)**
* **Algoritmo**

Immagine che contiene testo

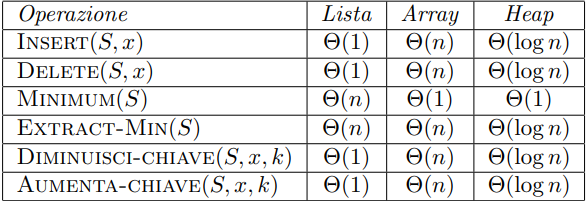
Descrizione generata automaticamente

* **Complessità Theta(log n)**
* **Diminuisci-Chiave(S,i,k)**
* **Algoritmo**

Immagine che contiene testo

Descrizione generata automaticamente

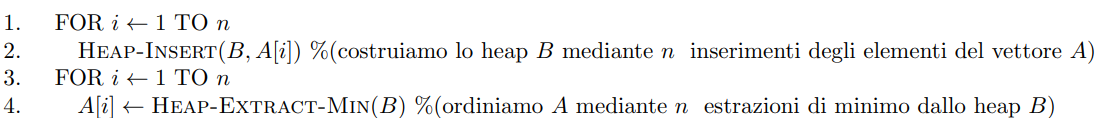
* **Complessità Theta(log n)**

L’heap risulta molto più efficiente nelle operazioni di un fattore esponenziale, qui di seguito la tabella che riassume le complessità delle operazioni nelle varie strutture dati: 

L’heap trova anche applicazione nell’ordinamento per una sequenza di n numeri

**HeapSort (con array di appoggio)**

**Algoritmo:**

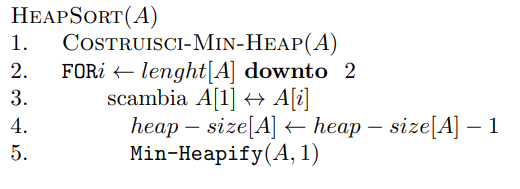
****

Consideriamo le operazioni di insert, eseguiremo n volte l’operazione di insert (finché l’heap raggiunge dimensione pari ad n prendendo da un array A), allora **O(n log n)**

Allora per ricomporre l’array eseguiremo n volte l’estrazione del minimo (finché l’heap non si svuota e l’array A non si riempie), **O(n log n)**

**HeapSort (senza array di appoggio)**

**Algoritmo:**

****

L’algoritmo funziona riempiendo l’heap con gli elementi della sequenza A. Estratto (ed inserito nell’array di destinazione) il nodo radice, lo scambia con l’ultimo nodo dell’heap, viene chiamato l’heapify per risolvere i problemi creati dalla sostituzione (vedi sopra) e viene ridotta la dimensione dell’heap di 1. Questo procedimento riduce man mano la dimensione dell’heap e contemporaneamente li inserisce nell’array risultato, questo andrà avanti finché l’heap non sarà vuoto.

**Complessità O(n log n)** in alcuni casi risulta addirittura più veloce del MergeSort

**Cammino più breve in un grafo**

**Storia e applicazioni dell’algoritmo:**

Questo algoritmo, **oggi**, è utilizzato da Google Maps per la definizione del percorso da percorrere da un nodo di partenza ad uno di destinazione

In **origine** il problema si presentò alla creazione di Internet, la soluzione fu formulata con il protocolllo **OSPF** (Open Shortest Path First), che è un protocollo dinamico per l’instradamento di pacchetti, tutt’ora usato in reti che utilizzano IP (Internet Protocol)

**Problema algoritmico:**

Consideriamo il seguente problema algoritmico:

**Input**: Grafo diretto G=(V,E), nodo sorgente s, funzione E -> R che misura la lunghezza l(e) di ogni arco con e appartenente E

**Output**: cammino di lunghezza minima s ad ogni nodo t nel grafo, con la lunghezza di un cammino la somma delle lunghezze degli archi

Allora, ci interessa trovare:

* Min{ l(s,v1) + l(v1,v2) + … + l(vk, t) } che verrà detta **distanza** tra il nodo s e il nodo t

**Banalmente** se tutte le **lunghezze** degli archi sono **uguali**, il problema algoritmico è risolvibile con una **BFS**, che trova l’albero (radicato in s) con le minime lunghezze (e quindi il minimo cammino)

Per la soluzione ottimale di questo problema utilizzeremo la tecnica greedy. Questo è possibile in quanto vale il **principio di** **ottimalità**, ovvero, una soluzione ottima al problema allora è soluzione ottima anche ad un eventuale sottoproblema, questo poiché se esistesse una soluzione migliore al sottoproblema, la soluzione originale, per definizione, non potrebbe essere ottima.

L’algoritmo funzionerà così:

* 1. Considerando il nodo s, tra tutti i nodi a lui collegati scegliamo quello a costo minore. Questo produrrà il minimo cammino tra s e il nodo x a lui collegato.
  2. Iteriamo su x lo stesso procedimento scegliendo l’arco di costo minimo tra tutti i nodi a lui collegati. Questo produrrà sì una soluzione ottima per i nodi x e y suo successivo, ma anche soluzione ottima tra s ed y.

**Algoritmo di Dijkstra**

Formalizziamo l’intuizione precedente. Consideriamo un nodo u nel grafo e preoccupiamoci di calcolare la distanza d[u] minima dalla sorgente.

* Mantieni un insieme S di nodi esplorati di cui è già determinata la distanza d[u] da s. per quelli non esplorati assumiamo che d[u] = infinito
* Banalmente inizializziamo S ponendo d[s] = 0
* Per ogni nodo v non appartenente ad S, d[v] = min { d[u] + l(e) } dove e = il cammino (u,v) con u appartenente ad S
* Sia quindi w il nodo con distanza minima. Aggiungiamo w in S e poniamo d[w] = d’[w]. Questo implica che w sia il nodo **non appartenente (ancora) ad S**, ma, **vicino ad s**, con distanza minima. In altre parole, è il nodo da seguire per il cammino minimo
* L’algoritmo termina quando non ci sono altri nodi raggiungibili da s con un cammino diretto

**Analisi dell’algoritmo:**

Proviamo che i valori d[u] calcolati sono le distanze minime di un cammino tra s ed u, ovvero:

* Ad ogni passo, per ogni v appartenente ad S, d[v] è la distanza minima tra il nodo v e la sorgente s

All’aggiunta di un nodo v ad S, l’algoritmo:

* 1. Sia t un nodo generico non appartenente ad S ma raggiungibile da s, abbiamo calcolato la distanza minima d[t] = min {d[u]+ l(e)}.
  2. Viene posto d[v] = d’[v] = d[u]+ l(u,v), ovvero la distanza minima fin’ora raggiunta è composta da quella precedentemente raggiunta (d[u]) + la distanza tra l’ultimo nodo u considerato e il nuovo nodo v
  3. Si può provare che l’algoritmo non sbaglia nella scelta del nodo v in quanto la distanza minima d[u] non sarebbe minima, andando in contro ad una contraddizione

L’algoritmo di Dijkstra, quindi, calcola la distanza minima tra due nodi s ed u. è possibile implementare anche la memorizzazione delle distanze dei singoli nodi del cammino.

L’algoritmo si presenta così:

Immagine che contiene testo

Descrizione generata automaticamente

Possiamo quindi considerare ad ogni passaggio dell’algoritmo, una MinCoda Q a priorità formata dai nodi V-S (ovvero tutti i nodi del grafo non ancora “visitati”) ordinati in base al loro d’[.]

Ad ogni passaggio si estrae da Q il nodo v con la minima d’[v] e lo aggiungiamo in S. Aggiornando Q nel caso:

* Sia w appartenente a Q tale che (v,w) non appartiene ad E (non esiste un collegamento diretto tra v e w). Q rimane quindi inalterato
* Sia w appartenente a Q, tale che (v,w) appartiene ad E (esiste un collegamento diretto tra v e w). Q viene alterato con un’operazione di **DecreaseKey**.

Allora per ogni generico arco (x,y), con x appartenente a Q, verrà chiamato DecreaseKey al più una volta (quando x è “favorito” da aggiungere ad S)

**Conclusioni**

L’algoritmo di dijkstra può essere implementato utilizzando un **grafo** con **n nodi** ed **m archi** richiedendo tempo O(m) in aggiunta al tempo per le n chiamate **ExtractMin** ed m chiamate **DecreaseKey**

Se si utilizza un min-heap le operazioni sopraelencate richiedono O(log n), raggiungendo un totale di m chiamate, ovvero O(m log n)

Consideriamo il seguente problema algoritmico:

Immaginiamo di avere n postazioni V=v1,…,vn e vogliamo costruire una rete che li connette tra loro spendendo il meno possibile, possiamo rappresentare le connessioni con archi tra la postazione i e quella j = (vi,vj) con un grafo G=(V,E)

**Input**: Grafo G=(V,E), costi c(e) > 0 per ogni arco e ∈ E

**Output:** Sottoinsieme di archi T incluso in E tale che: (V,T) sia connesso e la somma dei suoi costi sia il più piccolo possibile

È ovvio che esiste una soluzione T di minimo costo in cui il grafo (V,T) è un albero = grafo aciclico. In quanto in presenza di un ciclo sicuramente si avrebbero archi **superflui** che portano connessioni dove già sono presenti, aumentando di conseguenza il costo. Difatti:

* Se (V,T) non è un albero allora possiede un ciclo C {u,v}. è possibile eliminare il suddetto ciclo.

Allora è possibile trovare un grafo di costo minore con (V,T-{u,v}).

È palese allora che eliminando tutti i cicli la soluzione sarà di minimo costo

**Algoritmo di Kruskal**

Siano quindi e1, e2,… en gli archi del grafo, ordinati in modo che c(e1) <= c(e2) <= … <= c(en)

1. Allora, aggiungiamo iterativamente gli archi ei partendo da quelli che costano meno fino a quelli che costano di più.
2. Se eventualmente ei crea un ciclo in T, allora lo si salta e si passa ad e(i+1)
3. L’algoritmo termina quando sono stati aggiunti tutti gli archi possibili a T connettendo tutti i nodi V. ovvero quando (V,T) è un albero

**Analogamente all’algoritmo di Dijkstra**, possiamo risolvere il problema algoritmico:

**Algoritmo di Prim**

1. Iniziamo con un nodo radice s, ad ogni passo attaccheremo il nodo che costerà meno. Perciò, ad ogni istante memorizzeremo un insieme **S sottoinsieme di V**. quindi inizialmente S = { s }
2. Ad ogni istante aggiungiamo un nodo che non appartiene **ancora** ad S, che minimizza l’aumento del costo. Quindi sarà aggiunto il nodo v tale che il costo dell’arco dal vertice u, appartenente ad S, al nuovo vertice v sia minim006F
3. L’algoritmo termina quando abbiamo connesso tutti i vertici di V

**Dimostrazione**

Per provare che queste soluzioni al problema MST, si utilizza una dimostrazione per assurdo.

Supponiamo che i costi degli archi siano tutti diversi. Possiamo quindi ottenere un primo risultato:

Sia S un insieme di nodi **non vuoto** tale che S ⊂ V, e sia e = (u, v) l’arco di costo minimo con un estremo u ∈ S e v ∈ V – S (ovvero v non appartiene **ancora** ad S ma è incluso in V). Allora ogni MST contiene l’arco e.

Per dimostrare supponiamo che questa affermazione non sia vera. Allora sia T un albero che non contiene l’arco e.

Questo albero per avere una soluzione ottima dovrà necessariamente possedere un arco h=(x,y) (con x appartenente ad S e y appartenente ad V-S) diversi da (u,v) per collegare tutti i nodi del grafo

Supponiamo di aggiungere l’arco e (che per ipotesi e<h) all’albero T. Si creerà un ciclo. Rimuovendo l’arco h avremo un albero T’ che avrà costo minore rispetto a T. allora T non è ottimo e l’ipotesi è confermata.

Si può dimostrare che l’algoritmo di **Kruskal** produca un albero. Si procede considerando il grafo di partenza **connesso**, quindi, necessariamente per collegare tutti i nodi, prima o poi l’algoritmo incontrerà l’arco utile per collegare questo nodo. Allora necessariamente questo insieme all’eliminazione di archi che creano cicli ci permettono di dire che il risultato sarà un albero.

L’algoritmo di **Prim** invece, procede diversamente, si dimostra che il suo prodotto sia un albero: è ovvio che sarà così poiché l’algoritmo aggiungerà tutti i nodi possibili prima o poi scartando i possibili collegamenti che producono cicli. Inoltre, l’algoritmo necessita che il grafo sia già **connesso** dall’input. Da ciò allora il prodotto dell’algoritmo sarà sicuramente un albero radicato in s che viene posto all’inizio

L’algoritmo di Prim sarà implementato come l’algoritmo di Dijkstra, ovvero utilizzando un Heap, allora per ogni nodo v presente in V-S, si manterrà il valore minimo (denominato con a(v)) dell’arco utile per raggiungerlo. I nodi vengono quindi mantenuti in una coda a priorità organizzata in base ai vari a(v). Questi verranno estratti con **ExtractMin** e poi verrà aggiornata la coda con **DecreaseKey**

Immagine che contiene testo, lettera

Descrizione generata automaticamenteL’algoritmo di Prim ha questo algoritmo:

Il calcolo della complessità sarà pari:

* Le inizializzazioni richiedono O(|V|)
* Avverranno |V| chiamate ad ExtractMin (che ha complessità log |V|). Allora O(|V| log |V|)
* Avverranno |E| chiamate a DecreaseKey(|E| log |V|).

La complessità totale sarà O(|E| log |V|)

**Tecnica Backtracking**

È una metodologia utile per analizzare tutte le possibili soluzioni di un problema (es. tutte le possibili permutazioni, o sottoinsiemi o tutti i cammini)

Questo serve in casi in cui non è possibile conoscere a priori la soluzione ottima o quale soluzione la produca; infatti, la si può conoscere a posteriori considerando tutte le possibili soluzioni

A livello pratico questa tecnica si utilizza quando non è possibile applicare programmazione dinamica o le tecniche già conosciute

La complessità di questa tecnica è legata allo spazio campionario di tutte le possibili soluzioni (ovvero se questo è di ordine esponenziale, allora la complessità sarà tale)

Per evitare che una soluzione sia calcolata più volte, è necessario stabilire un ordine nel quale calcolarle.

Consideriamo, per esempio, in modo astratto il seguente **problema**:

* rappresentiamo con SOL=(a1,a2,…,an) un vettore di soluzioni in cui ogni elemento ai è selezionato da un insieme ordinato Si di possibili soluzioni per la posizione i

ad ogni passo della ricerca per ai, avremo costruito una soluzione parziale per i primi k elementi (k<=n) e da questa soluzione parziale costruiamo l’insieme S(k+1) dei possibili candidati per la posizione k+1. Da questo proviamo ad estendere SOL con un elemento ak+1 di Sk+1. Questo continua finché l’estensione genera una soluzione parziale.

Nel caso l’estensione k+1 sia vuoto (o non generi una soluzione parziale), si torna a k (backtracking) e sostituendo l’elemento ak con un nuovo elemento a’k

Questa soluzione costruisce **un albero** detto “delle soluzioni” dove ogni nodo è una soluzione parziale e ogni arco da un nodo generico x ad uno y indica un’estensione della soluzione parziale da x ad y

Seguendo questa logica, le foglie dell’albero sono le effettive soluzioni

Generalmente il backtracking allora, se consideriamo l’albero delle soluzioni, è rappresentato da una visita in profondità

È preferibile la visita in profondità rispetto a quella per ampiezza, per motivi di spazio, in quanto generalmente l’ampiezza cresce esponenzialmente rispetto all’altezza

Consideriamo il seguente problema algoritmico:

Dato un insieme S = {s1,s2, . . . ,sn} vogliamo generare tutti i 2n sottoinsiemi di S.

Esempio: S = {s1,s2,s3}, vogliamo generare:

∅ {s1} {s2} {s3} {s1,s2} {s1,s3} {s2,s3} {s1,s2,s3}

Quindi è perfettamente equivalente a generare i 2n vettori binari di lunghezza n

Immagine che contiene testo, Carattere, bianco, calligrafia

Descrizione generata automaticamenteUtilizzando il vettore caratteristico, l’insieme Sk dei candidati per la k-esima posizione è dato dagli elementi {0,1}. Lo pseudocodice per questa soluzione è:

Esempio **generazione di permutazioni**

Dati n elementi vogliamo generare tutte le possibili permutazioni degli elementi sottoinsiemi di S=s1,s2,…,sn

Es. dati s1,s2 vogliamo calcolare (s1,s2) (s2,s1)

Per il primo valore, ci sono n possibili scelte (s1,s2,…,sn). Fissato a1 come primo elemento, ci sono quindi n-1 possibili scelte per la seconda posizione. Sono possibili n! distinte permutazioni

Quindi l’insieme Sk di candidati per la k-esima posizione è dato dagli elementi che non compaiono tra i primi k-1 elementi

Immagine che contiene testo, Carattere, ricevuta, bianco

Descrizione generata automaticamenteL’algoritmo in pseudocodice è:

Consideriamo di essere interessati solo ad alcune soluzioni con determinate proprietà, dette soluzioni ammissibili.

L’albero, per come l’abbiamo definito fin’ora, sarà pieno di tutte le possibili soluzioni (anche quelle non ammissibili) e perciò sarà necessario fare una **potatura** dell’albero eliminando queste soluzioni superflue